# oneAPI GPU 最適化ガイド

2023 年 11 月 7 日
リリース 2024.0

# 注意事項:

この日本語マニュアルは、インテル コーポレーションのウェブサイトで公開されている『oneAPI GPU Optimization Guide 2024.0』の日本語参考訳です。

インテル社の許可を得て iSUS (IA Software User Society) が翻訳版を作成した iSUS の著作物です。

原文は Intel Corporation の Copyright であり、日本語参考訳版にも適用されます。

† 開発コード名

# 目次

**はじめに** ..... **6**
パフォーマンスの移植性ではなく生産性を高めるパフォーマンス ..... 6
パフォーマンス上の考察 ..... 6
コードのプロファイルとチューニング ..... 7
ソースコードの例 ..... 7
**導入ガイド** ..... **8**
アムダールの法則 ..... 8
局所性の問題 ..... 8
ワークサイズ ..... 9
**並列処理** ..... **10**
並列プログラミング言語や API を使用 ..... 10
コンパイラー ..... 10
ライブラリー ..... 10
**インテル® Xe GPU アーキテクチャー** ..... **11**
Xe LP エグゼキューション・ユニット (EU) ..... 11
Xe LP デュアル・サブスライス ..... 12
Xe LP スライス ..... 12
Xe コア ..... 13
Xe スライス ..... 14
Xe スタック ..... 15
Xe HPC インテル® データセンター GPU マックス・シリーズ (2 スタック) ..... 15
Xe HPG GPU ..... 16
Xe インテル® データセンター GPU フレックス・シリーズ ..... 17
用語と構成のサマリー ..... 17
**GPU 上での汎用計算** ..... **19**
GPU 実行モデル概要 ..... 20
スレッドのマッピングと GPU 占有率 ..... 21
nd_range ..... 22
スレッドの同期 ..... 22
最大の占有率を得るように work-group を Xe コアへマップ ..... 22
work-group 内での work-item の同期の影響 ..... 24
work-group 内ローカルメモリーの影響 ..... 25
詳細例 ..... 25
インテル® GPU の占有率計算 ..... 33
カーネル ..... 34
sub-group と SIMD ベクトル化 ..... 34
条件チェックを排除 ..... 41
レジスターの使用とスピルの回避 ..... 43
レジスタープレッシャーの高いコードのインテル® データセンター GPU マックス・シリーズへの移植 ..... 51
スモール・レジスター・モードとラージ・レジスター・モード ..... 58
共有ローカルメモリー ..... 61
ポインターのエイリアシングと restrict ディレクティブ ..... 69
カーネル内のスレッド間での同期 ..... 72
work-group サイズを選択する考慮事項 ..... 80
プリフェッチ ..... 84
リダクション ..... 88

カーネルの起動 ..... 94
デバイス上で同時に複数のカーネルを実行 ..... 96
複数のキューへカーネルを送信 ..... 98
冗長なキューの構築を避ける ..... 101
SYCL* 結合行列拡張を使用したインテル® XMX のプログラミング ..... 105
カーネル内での I/O ..... 107
GPU オフロードにライブラリーを使用 ..... 110
パフォーマンス・ライブラリーを使用 ..... 110
SYCL* カーネルで共有ライブラリー関数を使用 ..... 112
oneAPI マス・カーネル・ライブラリー (oneMKL) を使用した効率良いフーリエ相関の実装 ..... 115
ホスト/デバイスメモリー、バッファー、および USM ..... 123
統合共有メモリーの割り当て ..... 123
USM とバッファーのパフォーマンスへの影響 ..... 125
ホストとデバイス間でデータ移動の往復を避ける ..... 128
ループ内でのバッファー宣言を避ける ..... 132
バッファー・アクセサー・モード ..... 134
ホスト/デバイスの調整 ..... 140
ホストとデバイス間のデバイス間の非同期および重複データ転送 ..... 140
複数のヘテロジニアス・デバイスを使用 ..... 144
各種デバイスで計算をオーバーラップ ..... 144
コンパイル ..... 146
ジャストインタイム・コンパイル ..... 146
事前コンパイル ..... 149
特殊化定数 ..... 150
浮動小数点計算における精度とパフォーマンスのトレードオフ ..... 155
OpenMP* オフロードのチューニング・ガイド ..... 166
OpenMP* ディレクティブ ..... 166
OpenMP* 実行モデル ..... 167
用語 ..... 168
OpenMP* アプリケーションのコンパイルと実行 ..... 168
oneMKL の計算を GPU へオフロードする ..... 172
OpenMP* アプリケーションのパフォーマンスを解析するツール ..... 205
OpenMP* オフロードの最良の事例 ..... 207
マルチ GPU、マルチスタック、およびマルチ C スライス・アーキテクチャーとプログラミング ..... 282
マルチスタック GPU アーキテクチャー ..... 282
GPU メモリーシステム ..... 284
暗黙のスケーリング ..... 288
明示的なスケーリング ..... 298
GPU クラスター向けのインテル® MPI ..... 318
用語 ..... 329
レベルゼロ ..... 332
即時コマンドリスト ..... 332
パフォーマンスのプロファイルと解析 ..... 334
タイマーを使用 ..... 334
インテル® VTune™ プロファイラー ..... 336
インテル® Advisor ..... 353
OpenCL* アプリケーション向けインテル® インターセプト・レイヤー ..... 374
GPU 向けのインテル® プロファイル・ツール・インターフェイス ..... 375
GPU デバイスの構成 ..... 376

はじめに ... 376
GPU ドライバーまたはプラグインのインストール (オプション) ... 377
GPU 周波数を固定することのパフォーマンスへの影響 ... 377
インテル® Xe リンクのオン/オフの切り替え ... 378
**GPU 上でのメディア・グラフィックス計算 ... 379**
メディア・パイプラインの最適化 ... 379
メディア・エンジン・ハードウェア ... 380
ハードウェア・アクセラレーション向けのメディア API オプション ... 382
メディア・パイプラインの並列処理 ... 382
メディア・パイプラインの相互運用とメモリー共有 ... 384
SYCL* によるぼかし (Blur) の例 ... 389
インテル® グラフィックス・パフォーマンス・アナライザーによるパフォーマンス解析 ... 390
概要 ... 390
**参考資料 ... 403**
**法務上の注意書き ... 404**

# はじめに

高性能なソフトウェアを設計するには、ソフトウェアを作成する際に「通常とは異なる考え方」をする必要があります。コードを実行するターゲット・ハードウェアと、そのハードウェアのパフォーマンス特性に十分な注意が必要です。皆さんの目的は、正しい結果を生成するようにコードを構造化することであると思いますが、それにはコードを実行するハードウェアの能力を最大限に引き出す方法を考慮しなければなりません。

oneAPI は、業界を超えたオープンな標準化をベースとした統一プログラミング・モデルであり、アクセラレーター・アーキテクチャー全体で共通の開発手法を提供します。アクセラレーターのユニークな機能は、プラットフォームのメイン CPU に追加装着できるということです。そして、アクセラレーターを利用する主な利点は、ソフトウェアをホストとアクセラレーターに分割して、アクセラレーターに最適化された計算領域を特殊化することで、ソフトウェアの能力を改善できることです。アクセラレーターのアーキテクチャーは、特定クラスの計算向けに計算ハードウェアを特殊化することでメリットを得ています。これにより、アクセラレーターのアーキテクチャーに特化したソフトウェアに最適な結果を供給できます。

このドキュメントの主なターゲットは GPU であり、それぞれの節では、最適化されたソリューションを開発するための多くのトピックに焦点を当てています。インテル® oneAPI ツールキットは、コードの最適化に必要な言語と開発ツールを提供します。これには、コンパイラー、デバッガー、プロファイラー、アナライザー、およびライブラリーが含まれます。

## パフォーマンスの移植性ではなく生産性を高めるパフォーマンス

このドキュメントは GPU に注目しますが、CPU やそのほかのアクセラレーターで実行されるアプリケーションが必要になることも考えられます。アクセラレーターのアーキテクチャーは特殊化されているため、最高のパフォーマンスを達成するにはコードも特殊化する必要があります。この特殊化には、アプリケーションからハードウェアへの最適なマッピングを見つけるためのコードの再構築とチューニングが含まれます。極端な場合には、適切な計算を実装するため、それぞれのアクセラレーターのアルゴリズムの再設計が必要になることもあります。oneAPI の素晴らしい点は、アクセラレーターで起動されたデバイス固有の機能を使用するため、それぞれのバリエーションを共通の言語で記述できることです。

## パフォーマンス上の考察

GPU を利用する際の最初のフェーズは、アプリケーションのどの領域がオフロードの恩恵を得られるか特定することです。これは通常、計算に対するメモリーアクセスの比率が適切で、GPU にマッピングする適切なデータ依存のパターンを持つ計算集約型のコードであることがほとんどです。GPU にはローカルメモリーが備わっており、通常、大規模な並列処理を実行できます。そのため、どの領域を実行するか決定する際に、コードが持つ特性を理解することが最も重要となります。

インテル® oneAPI ベース・ツールキットに含まれるインテル® Advisor ツールは、コードを解析して並列実行に最適な場所を特定するように設計されています。インテル® Advisor のプロファイラーは、各種アクセラレーターにマッピングされた際にコードがどのように実行されるか予測するため、関数内のデータ移動、メモリー・アクセス・パターン、および計算量を測定します。潜在的に最も利点があるコード領域をアクセラレートする最初の対象とすべきです。

GPU は、多くの場合、複数レベルで並列処理を活用します。これには、ホストとデバイス間のオーバーラップ、計算コア間の並列処理、計算とメモリーアクセスのオーバーラップ、同時パイプライン、およびベクトル計算が含まれます。これらすべてのレベルで並列処理を行うには、GPU アーキテクチャーとライブラリー、および言語の機能を十分に理解する必要があります。

すべての計算リソースを常にビジー状態に保つことも重要です。例えば、デバイスに 100 個の計算コアがあり、タスクが 1 つしかない場合、デバイスの 99% はアイドル状態であると言えます。デバイスの飽和状態を保つには、ハードウェアが実行中のタスクの完了時に次のワークをスケジュールできるよう、利用可能な計算リソースよりもかなり多めの独立したタスクを生成する必要があります。

そして、ホストとデバイス間の同期を最小限に抑えます。ホストはデバイスでカーネルを起動し、その完了を待機します。カーネルの起動にはオーバーヘッドが生じるため、カーネルの起動回数を最小限にするように計算を構成します。

また、ホストとデバイス間のデータ転送を最小化することも考慮しなければなりません。データは通常ホストで準備され、計算の入力としてデバイスへコピーされます。そして、計算が完了した後、結果をホストへ転送する必要があります。最高のパフォーマンスを得るには、デバイスの計算間で中間結果を保持することで転送を最小限に抑えます。また、計算とデータ移動をオーバーラップしてデータ転送の影響を減らし、計算コアがデータを待機する時間が生じないようにします。

データを高速メモリーに保持して適切なアクセスパターンを使用します。GPU アーキテクチャーには、アクセスコストが異なるさまざまなタイプのメモリーが搭載されています。レジスター、キャッシュ、またはスクラッチパッドは、ローカルメモリーよりもアクセスが容易ですが、容量は小さくなります。データがレジスター、キャッシュライン、またはメモリーページにロードされる場合、次のチャンクに移動する前にロードされたすべてのデータを利用するようなアクセスパターンを導入します。メモリーがバンク化されている場合、すべての計算コアが同じバンクに同時にアクセスするのを回避するようなストライドを適用します。

## コードのプロファイルとチューニング

高いパフォーマンスを達成できるコードを設計したら、次のステップでは、ターゲット・アクセラレーターでコードがどのように実行されるか観測します。コードに時間計測コードを追加して、トレースを収集、またはインテル® VTune™ プロファイラーなどのツールを使用して実行中のプログラムを監視します。収集した情報からハードウェアのボトルネックとアイドル状態を特定して、ハードウェアのピーク・パフォーマンスと比較し、最適化の対象とすべき最も重要なホットスポットを見つけます。

## ソースコードの例

本書では、実際のコード例を使用して最適化の手法を説明します。ここで使用するすべてのサンプルコードは、oneAPI-samples GitHub* リポジトリー (英語) から入手できます。サンプルをダウンロードして、以下の手順の指示にしたがって、環境を設定してください。

それぞれのサンプルは、最適化の方法を示すため、可能な限りシンプルに、短く、そして分かりやすくしています。サンプルコードは、ドキュメント全体で参照されます。

「百聞不如一見、百聞不如一為、故に、万聞不如一為」(百聞は一見にしかず、の元になった孔子の言葉です) ということわざがあります。また、本書をお読みになっているときにサンプルコードに遭遇したら、読み進めるのをいったんやめて実機でサンプルを検証することをお勧めします。

ともあれ、oneAPI GPU 最適化ガイドへようこそ !!

# 導入ガイド

このドキュメントを最後まで読む時間がない読者は、少なくとも、アクセラレーター向けにソフトウェアを最適化する次の 3 つの概念を理解してください。

## アムダールの法則

これは当然のように思われますが、アクセラレーターを利用する最初のステップと言えます。アムダールの法則によれば、アプリケーションがアクセラレーターを使用する時間の比率 (*Fp*) によってアクセラレーションのメリットが制限され、最大スピードアップは、1/(1 – *Fp*) で制限されます。例えば、アクセラレーターを 50% 使用する場合、無限数のアクセラレーターを追加しても最大 2 倍のスピードアップしか得られません。

これは、プログラムのソースコードではなく実行に関連するものである、ということに注意してください。並列カーネルは、ソースコード全体のごく一部である可能性がありますが、実行時間が集中している場所であれば、それで問題ないでしょう。

## 局所性の問題

多くの場合、アクセラレーターには互いに独立したアドレス空間を持つ特殊なメモリーが備えられています。そのため、アプリケーションは、適切なタイミングで適切なメモリーにデータを割り当てるか、移動する必要があります。

アクセラレーターのメモリーは階層的に配置されます。レジスターはキャッシュよりもアクセス効率が良く、キャッシュはメインメモリーよりも効率良いアクセスが可能です。データを実行ポイントに近づけることで効率は向上します。

コードを再構成してデータを実行ポイントに近づける方法はたくさんあります。これらについては以下で説明します。ここでは次の 3 つに注目します。

1. アクセラレーターにデータを割り当ててデータをコピーする場合、可能な限りデータの常駐を維持します。アプリケーションには多数のオフロード領域が含まれる場合がありますが、それらの領域間で共通のデータがある場合、最初のコピーのコストを償却するため、残りのカーネル呼び出しでデータを再利用するのは理にかなっています。
2. カーネルの実行中は連続したメモリーブロックにアクセスします。ハードウェアは連続するブロックをメモリー階層にフェッチするため、ブロック単位で転送のコストが生じます。ブロックの最初の要素を参照すると、残りの要素は低いコストでアクセルできるためできるだけそれらを使用します。
3. データの再利用率が高いブロックにコードを再構成します。2 次元行列では、次の要素ブロックにアクセスする前に、要素を含む 1 ブロックを処理するようにワークを調整できます。例えば、ステンシル操作では、前の行、現在の行、そして次の行にアクセスできます。ブロック内の要素をすべてアクセスすると、データが再利用されメモリーにデータを要求するコストを排除できます。

# ワークサイズ

データ並列アクセラレーターは、スループット・エンジンとして設計されており、エグゼキューション・ユニットを繰り返し複製することに特化しています。これにより、より多くの要素を同時に処理することができますが、データ並列アルゴリズムで高いパフォーマンスを得るには厄介であり、並列プロセッサーを完全に利用するのは困難な場合があります。例えば、512 個のエグゼキューション・ユニットがあり、それぞれのエグゼキューション・ユニットは 8 つのスレッドを実行し、スレッドには 16 要素のベクトルがあると仮定します。これに一致するようにスケジュールを行うには、最低でも 512 x 8 x 16 = 65536 の並列アクティビティーが必要となります。さらに、それぞれの並列アクティビティーが小さい場合、このワークをアクセラレーターに送信するコストを償却するには、また別の大きな要素が必要です。単一の大規模なアクセラレーターを完全に利用するには、計算を数百万の並列アクティビティーに分割することが求められることもあります。

# 並列処理

アクセラレーターはコードを並列に実行できる多くの独立した処理要素を備えているため、並列処理はアクセラレーターを効率良く利用するには不可欠です。並列コードを開発するには 3 つの方法があります。

## 並列プログラミング言語や API を使用

並列処理を表現するために利用できる並列プログラミング言語と API は各種存在します。oneAPI は、SYCL* フレームワークによる並列プログラム開発をサポートします。また、インテル® oneAPI では、これらのプログラムをさまざまなアクセラレーターで実行可能なバイナリーに変換する、コード生成ツールも多数用意されています。通常の作業工程では、シリアルプログラムを実行し、時間を要するコード領域 (ホットスポットと呼ばれる) を識別して、それらを並列カーネルに変換し、アクセラレーターに送信して実行します。

## コンパイラー

OpenMP* のようなディレクティブ・ベースの手法は、並列プログラムを開発するもう 1 つの方法です。ディレクティブ・ベースのアプローチでは、プログラマーはコード自身を明示的に変更することなく、コンパイラーに並列処理に関するヒントを提供します。このアプローチは、スクラッチから並列プログラムを開発するよりも簡単です。

## ライブラリー

oneAPI には、oneTBB、oneMKL、oneDNN、およびインテル® ビデオ・プロセシング・ライブラリー (インテル® VPL) など多くのライブラリーが含まれており、さまざまなアクセラレーター・アーキテクチャーで実行される汎用的な計算操作のバージョンが用意されています。アプリケーションのニーズに応じて、開発者はこれらのライブラリーの関数を直接呼び出すことで、ベースとなるアーキテクチャー向けの効率良い実装を選択できます。必要とする関数がライブラリー含まれている場合、これは並列プログラムを開発する最も簡単なアプローチと言えます。例えば、oneDNN はマシンラーニング・アプリケーション向けに最適化されたプリミティブを提供します。これらのライブラリーは、正当性とパフォーマンスの両面で徹底した検証が行われており、プログラムの信頼性を高めます。

# インテル® Xe GPU アーキテクチャー

インテル® Xe GPU ファミリーは、統合/低電力 (Xe LP) から、愛好家/高性能ゲーム (Xe HPG)、データセンター/AI (Xe HP)、およびハイパフォーマンス・コンピューティング (Xe HPC) 向けの一連のアーキテクチャーで構成されています。

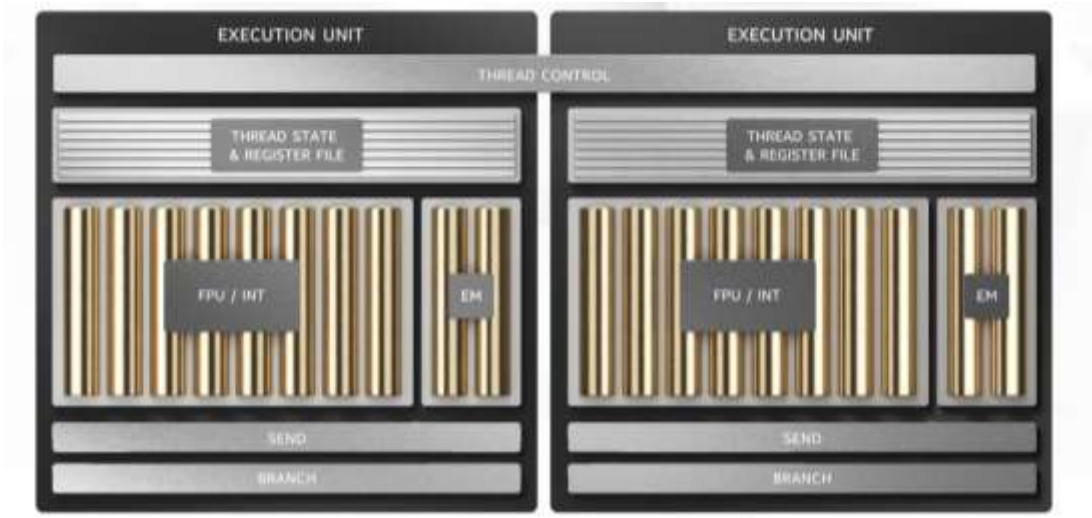
### インテル® Iris® Xe ファミリー

この節では、インテル® Xe GPU ファミリーのマイクロアーキテクチャーと設定パラメーターについて説明します。

## Xe LP エグゼキューション・ユニット (EU)

エグゼキューション・ユニット (EU) は、インテル® Xe LP GPU アーキテクチャーにおける最も小さなスレッドレベルの構成要素です。各 EU は 7 つのスレッドによる同時マルチスレッド (SMT) を実装します。主な計算ユニットは、SIMD8 FP/INT 操作をサポートする 8 要素の SIMD (Single Instruction Multiple Data) 演算ロジックユニット (ALU) と、SIMD2 拡張数学操作をサポートする 2 要素の SIMD ALU から構成されています。それぞれのハードウェア・スレッドは、32B 幅の 128 個の汎用レジスター (GRF) を持ちます。

### Xe LP-EU



Xe LP EU は、AI アプリケーション向けに FP16、INT16 および INT8 データタイプをサポートします。次の表では、Xe LP における各種データタイプの EU スループットを比較しています。

**インテル® Xe LP GPU 計算スループット率 (ops/clock/EU)**

| FP32 | FP16 | INT32 | INT16 | INT 8 |
|---|---|---|---|---|
| 8 | 16 | 8 | 16 | 32 (DP4A) |

# Xe LP デュアル・サブスライス

それぞれの Xe LP デュアル・サブスライス (DSS) は、16 個の EU アレイ、命令キャッシュ、ローカル・スレッド・ディスパッチャー、共有ローカルメモリー (SLM)、および 128B/サイクルのデータポートで構成されます。ハードウェアが SIMD16 を実行するため 2 つの EU をペアにするので、デュアル・サブスライスと呼ばれます。

SLM は、サブスライス内の EU からアクセス可能な 128KB の低レイテンシーおよび高帯域幅メモリー (HBM) です。SLM の重要な用途の 1 つは、サブスライス内で同時に実行されるワーク項目間でアトミックデータとシグナルを共有することです。そのため、カーネルのワークグループに同期処理が含まれる場合、ワークグループ内のすべてのワーク項目は単一のサブスライスに割り当てられ、同じ 128KB SLM に共有アクセスできるようにする必要があります。サブスライスの占有率と利用率を最大化するため、ワークグループのサイズを注意深く選択しなければなりません。一方、カーネルが SLM にアクセスしない場合、ワーク項目は複数のサブスライスにまたがってディスパッチされる可能性があります。

次の表は、サブスライスの計算能力をまとめたものです。

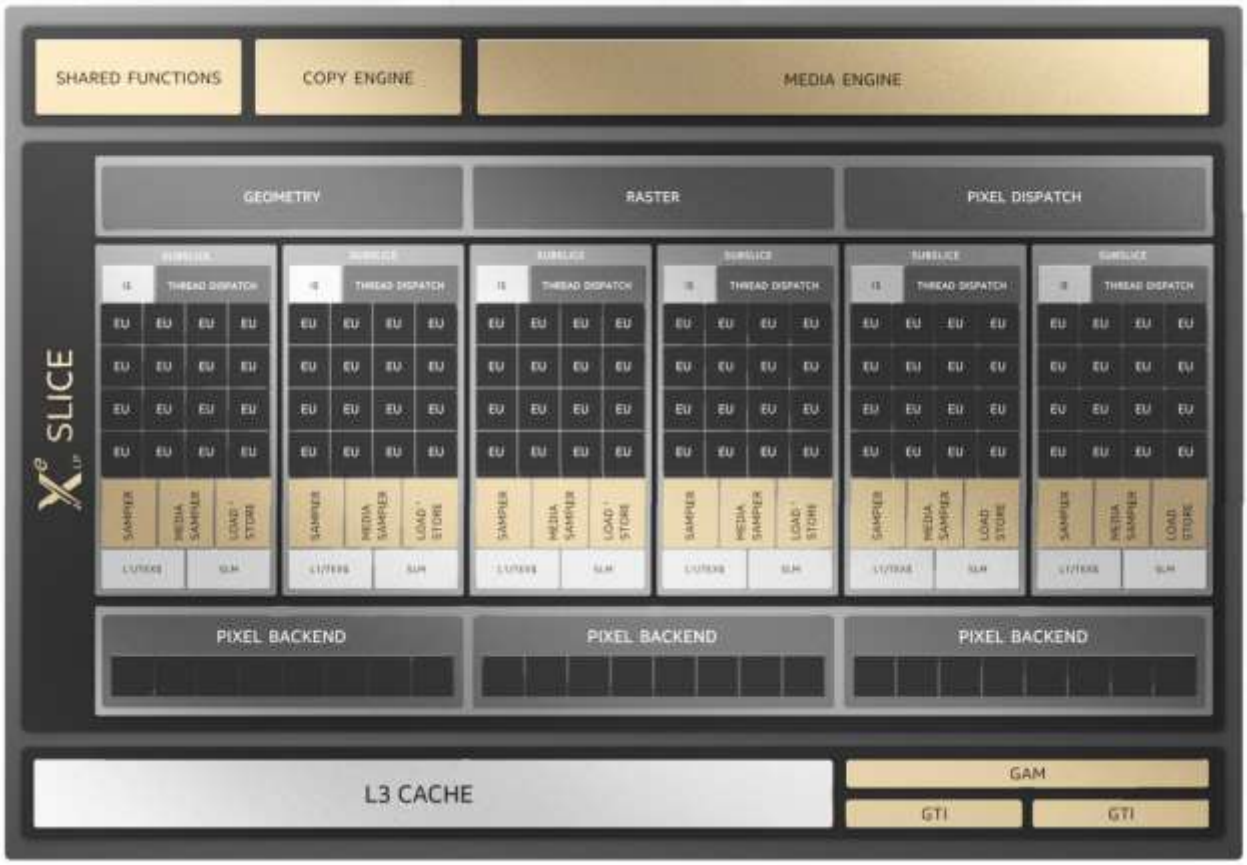
**インテル® Xe LP サブスライスの計算能力**

| EU 数 | スレッド数 | 操作 |
|---|---|---|
| 16 | 7 x 16 = 112 | 112 x 8 = 896 |

# Xe LP スライス

各 Xe LP スライスは 6 つのサブスライス (デュアル) で構成され、合計 96 個の EU、最大 16MB の L2 キャッシュ、L2 への 128B/サイクル帯域幅、および 128B/サイクルのメモリー帯域幅を持ちます。

### Xe LP スライス




## Xe コア

Xe LP や以前のインテル® GPU では、計算ユニットをエグゼキューション・ユニット (EU) と呼んでいましたが、Xe HPG と Xe HPC では Xe コアと呼称します。これは、Xe LP デュアル・サブスライスと同様です。

Xe コアは、ベクトルエンジンおよび行列エンジンと呼ばれるベクトル ALU と行列 ALU を搭載しています。

Xe コアは、8 つのベクトルエンジン、8 つの行列エンジン、および 512KB 大容量キャッシュ/SLM を搭載しています。これは、インテル® データセンター GPU マックス・シリーズの中核となります。各ベクトルエンジンは 512 ビット幅で、融合型 FMA による 16 個の FP32 SIMD 操作をサポートしています。8 つのベクトルエンジンを搭載する Xe コアは、512 FP16、256 FP32 および 128 FP64 操作/サイクルを実現します。

各行列エンジンは 4096 ビット幅です。8 つの行列エンジンを搭載する Xe コアは、8192 int8、4096 FP16/BF16 および 2048 FP32 操作/サイクルを実現します。また、Xe コアはメモリーシステムに対して 1024B/サイクルのロード/ストア帯域幅を提供します。

### Xe コア

## Xe スライス

Xe スライスには、合計 8MB L1 キャッシュを持つ 16 個の Xe コア、16 個のレイトレーシング・ユニット、1 個のハードウェア・コンテキストが含まれます。

### Xe スライス

# Xe スタック

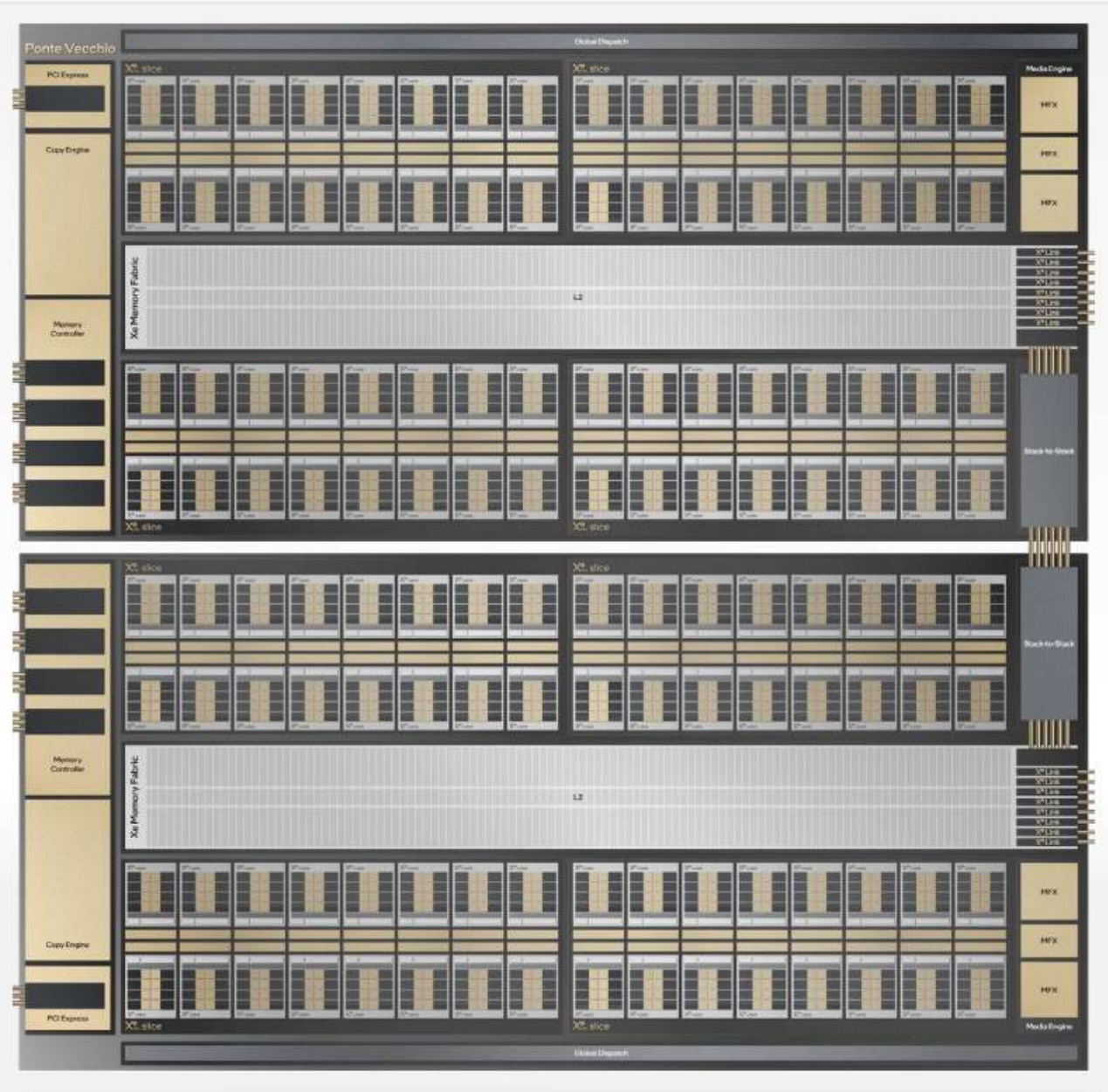
Xe スタックは最大 4 つの Xe スライスを含みます。64 個の Xe コア、64 個のレイトレーシング・ユニット、4 個のハードウェア・コンテキスト、4 個の HBM2e コントローラー、1 つのメディアエンジン、および 8 つの Xe リンク・ハイスピード・コヒーレント・ファブリックを搭載します。また、共有 L2 キャッシュも含みます。

### Xe スタック

# Xe HPC インテル® データセンター GPU マックス・シリーズ (2 スタック)

Xe HPC インテル® データセンター GPU マックス・シリーズ (2 スタック) は 2 つのスタックで構成されます。8 個のスライス、128 個の Xe コア、128 個のレイトレーシング・ユニット、8 個のハードウェア・コンテキスト、8 個の HBM2e および 16 個の Xe リンクが含まれます。

### Xe HPC 2 スタック




## Xe HPG GPU

Xe HPG は、Xe アーキテクチャーのエンスージアストまたはハイパフォーマンス・ゲーミング向けのバリエーションです。これは、グラフィックス・パフォーマンスに特化したマイクロアーキテクチャーであり、ハードウェア・アクセラレーションによるレイトレーシングをサポートします。

各 Xe コアには、16 個のベクトルエンジン、16 個の行列エンジンが搭載されています。これは、インテル® Arc™ GPU の中核となります。各ベクトルエンジンは 256 ビット幅で、融合型 FMA による 8 個の FP32 SIMD 操作をサポートしています。16 個のベクトルエンジンを搭載する Xe コアは、サイクルあたり 256 FP32 操作を実現します。各行列エンジンは 1024 ビット幅です。16 個の行列エンジンを搭載する Xe コアは、サイクルあたり 4096 int8 操作、およびサイクルあたり 2048 FP16/BF16 操作を実現します。また、Xe コアはメモリーシステムに対して 512B/サイクルのロード/ストア帯域幅を提供します。

Xe HPG GPU は、8 つの Xe HPG スライスで構成され、最大 4 つの Xe HPG コアがあり、合計 4096 FP32 ALU ユニット/シェーダーコアが含まれます。

## Xe インテル® データセンター GPU フレックス・シリーズ

インテル® データセンター GPU フレックス・シリーズには、2 つの構成があります。150W オプション構成では、PCIe* 4.0 カードに 32 個の Xe コアがあります。75W オプションには、16 Xe コア (GPU あたり 8 Xe コア) の 2 つの GPU があります。どちらの構成にも、4 つの Xe メディアエンジン、データセンター向けでは業界初の AV1 ハードウェア・エンコーダーおよびアクセラレーター、GDDR6 メモリー、レイトレーシング・ユニット、ビルトインのインテル® XMX AI アクセラレーションが付属しています。

インテル® データセンター GPU フレックス・シリーズは、Xe HPG GPU の派生品であり、インテル® データセンター GPU フレックス 170 は 4096 FP32 ALU ユニット/シェーダーコアを備えた合計 32 Xe コアの 8 つの Xe HPG スライスで構成されます。

データセンターにおけるクラウドゲーム、メディア・ストリーミング、およびビデオ分析アプリケーション向けのインテル® データセンター GPU フレックス・シリーズは、ハードウェア・アクセラレーションによる AV1 エンコーダーを備えているため、品質を損ねることなくビットレートを 30% 向上します。カードごとに 8 つの同時 4K ストリームまたは 30 以上の 1080p ストリームをサポートします。AI モデルは、インテル® データセンター GPU フレックス・シリーズの Xe コアを利用したデコードストリームに適用できます。

メディア・ストリーミングおよび配信ソフトウェア・スタックは、AV1 を含むすべての主要なコーデックのデコードとエンコードのアクセラレーションにインテル® ビデオ・プロセシング・ライブラリー (インテル® VPL) を採用しています。メディア配信事業者は、2 つの主要なメディア・フレームワークである FFMPEG* または GStreamer を選択でき、どちらもインテル® CPU とインテル® GPU 上でインテル® VPL によるアクセラレーションが可能です。

メディアストリームのデコードとエンコードをアクセラレートするインテル® VPL と並行して、oneAPI ディープ・ニューラル・ネットワーク・ライブラリー (oneDNN) は、TensorFlow* や PyTorch* フレームワークの推論モードをアクセラレートするのに有効な AI 最適化カーネル、または OpenVINO™ モデル・オプティマイザーや推論エンジンを提供し、推論をさらに加速してユーザーのワークロードの展開をスピードアップします。

## 用語と構成のサマリー

次に示すアーキテクチャー用語の変更の表は、従来の GPU 用語 (第 9 世代から第 12 世代インテル® Core™ アーキテクチャーで使用された) を、インテル® Xe GPU アーキテクチャー (12.7 世代以降) と比較して新しい用語に対応させたものです。

**アーキテクチャー用語の変更**

| 古い用語 | 新しい用語 | 一般的な用語 | 新しい略語 |
|---|---|---|---|
| エグゼキューション・ユニット (EU) | Xe ベクトルエンジン | ベクトルエンジン | XVE |
| シストリック/「EU の DPAS 部分」 | Xe 行列拡張 | 行列エンジン | XMX EU |
| サブスライス (SS) またはデュアル・サブスライス (DSS) | Xe コア | NA | XC |
| スライス | レンダースライス/計算 | スライス | SLC |
| タイル | スタック | スタック | STK |

次の表は、Xe ファミリー GPU 全体のハードウェア特性を示しています。

### Xe の構成

| アーキテクチャー | Xe LP (TGL†) | Xe HPG (インテル® Arc™ A770 グラフィックス) | Xe HPG (インテル® データセンター GPU フレックス 170) | Xe HPC (インテル® データセンター GPU マックス 1550) |
|---|---|---|---|---|
| スライス数 | 1 | 8 | 8 | 4 x 2 |
| XC (DSS/DSS) 数 | 6 | 32 | 32 | 64 x 2 |
| XVE (EU)/XC | 16 | 16 | 16 | 8 |
| XVE 数 | 96 | 512 | 512 | 512 x 2 |
| スレッド/XVE | 7 | 8 | 8 | 8 |
| スレッド数 | 672 | 4096 | 4096 | 4096 x 2 |
| FLOPs/clk - 単精度、MAD | 1536 | 8192 | 8192 | 16384 x 2 |
| FLOPs/clk - 倍精度、MAD | NA | NA | NA | 16384 x 2 |
| FLOPs/clk - FP16 DP4AS | NA | 65536 | 65536 | 262144 x 2 |
| GTI 帯域幅 バイト/非スライス - clk | r:128、w:128 | r:512、w:512 | r:512、w:512 | r:1024、w:1024 |
| LL キャッシュサイズ | 3.84MB | 16MB | 16MB | 最大 408MB |
| SLM サイズ | 6 x 128KB | 32 x 128KB | 16 x 128KB | 64 x 128KB |
| FMAD、SP (ops/XVE/clk) | 8 | 8 | 8 | 16 |
| SQRT、SP (ops/XVE/clk) | 2 | 2 | 2 | 4 |

# GPU 上での汎用計算

従来、GPU は画像や動画などコンピューター・グラフィックスの作成で使用されていました。近年の GPU は、超並列処理向けのエグゼキューション・ユニットも多数備わっているため、これまで CPU で実行されていたタスクの計算にも利用されています。これは一般に、GPU または GPGPU による汎用コンピューティングと呼ばれます。

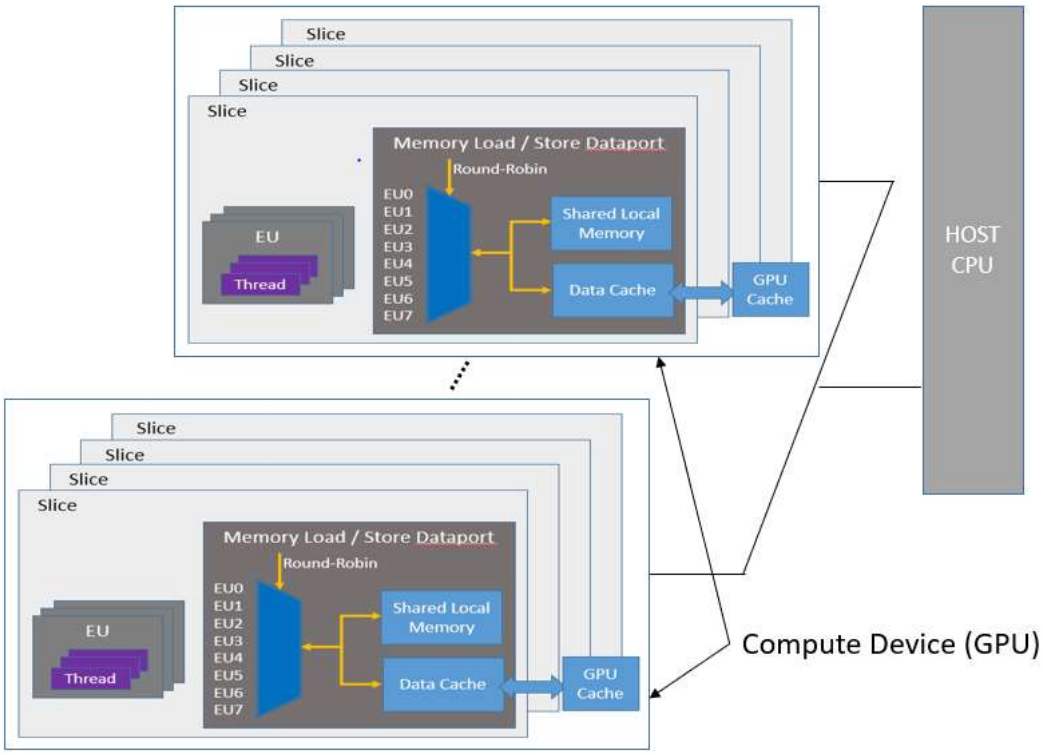
ハイパフォーマンス・コンピューティングやマシンラーニングのアプリケーションは、GPGPU から大きな恩恵を受けています。


- GPU 実行モデル概要
- スレッドのマッピングと GPU 占有率
- カーネル
- GPU オフロードにライブラリーを使用
- ホスト/デバイスメモリー、バッファー、および USM
- ホスト/デバイスの調整
- 複数のヘテロジニアス・デバイスを使用
- コンパイル
- OpenMP* オフロードのチューニング・ガイド
- マルチ GPU、マルチスタック、およびマルチ C スライス・アーキテクチャーとプログラミング
- レベルゼロ
- パフォーマンスのプロファイルと解析
- GPU デバイスの構成

# GPU 実行モデル概要

汎用 GPU (GPGPU) の計算モデルは、1 つ以上の計算デバイスに接続されたホストで構成されます。各計算デバイスは、エグゼキューション・ユニット (EU) またはベクトルエンジン (XVE) と呼ばれる多数の GPU 計算エンジン (CE) で構成されています。また、計算デバイスは、次の図に示すように、キャッシュ、共有ローカルメモリー (SLM)、高帯域幅メモリー (HBM) などを含む場合があります。アプリケーションは、ホスト上のソフトウェア (ホスト・フレームグラフごと) と、ホストから送出されるカーネルの組み合わせとして構築され、事前に定義された分岐ポイントで VE 上で実行されます。

## 汎用計算モデル




GPGPU 計算アーキテクチャーは、ホストプログラムと、ホストで設定されたコンテキスト内で実行されるカーネルセットの 2 つの異なる実行単位を含んでいます。ホストは、コマンドキューを介してこれらのカーネルと対話します。各デバイスは独自のコマンドキューを保持することができます。コマンドキューにコマンドが投入されると、コマンドの依存関係がチェックされ、計算ユニットクラスター内の VE で実行されます。コマンドの実行が完了すると、カーネルは「スレッドの終了」メッセージによってライフサイクルの終了を通知します。

GP 実行モデルでは、カーネルをどのようにスケジュールして実行するかを決定します。`kernel-enqueue` コマンドが実行のためカーネルを送出するときに、コマンドはインデックス空間または N 次元のレンジを定義します。kernel-instance は、カーネルそしてカーネルに関連する引数値、およびインデックス空間を定義するパラメーターで構成されます。計算デバイスが kernel-instance を実行する際に、カーネル関数は定義されたインデックス空間、または N 次元のレンジ内の各ポイントに対し実行されます。

実行中のカーネル関数は work-item (ワーク項目) と呼ばれ、その集合を work-group (ワークグループ) と呼びます。計算デバイスは、work-group によって work-item を管理します。それぞれの work-item は、グローバル ID、または work-group ID と work-group 内のローカル ID の組み合わせで識別されます。

work-group の概念は、基本的にグループ内の複数のユニット項目で同じカーネルを実行するものであり、データ並列計算の本質であると考えられます。VE は、work-item を SIMD ベクトル形式で構成することで SIMD ベクトル上で同じカーネルを実行できるため、該当するアプリケーションの計算を高速化できます。

デバイスは、各 work-group を任意の順番で計算できます。また、1 つの work-group 内の work-item は同時に実行されますが、実行の順番は保証されません。バリアのような高レベルの work-group の同期機能は、work-group 内の各 work-item に適用され、必要とされる同期ポイントを容易にします。このような work-group 機能は、work-group 内のすべての work-item が正確に同じ work-group 機能に到達するように定義されなければなりません。

同期はコマンドレベルでも発生する可能性があり、ホストのコマンドキューにあるコマンド間で同期が行われることがあります。このモードでは、あるコマンドが別のコマンド、または複数のコマンドの実行ポイントに依存関係を持つことがあります。

プログラム内部のメモリー順序の制約に関する同期として、アトミックやフェンスがあります。これらの同期タイプは、特定の work-item のメモリー操作をほかの work-item からどのように見えるかを制御します。これにより、データ並列計算モデルにおけるマイクロレベルの同期ポイントが提供されます。

インテルの GPU デバイスは、多数のベクトルエンジン (VE) を搭載し、それぞれの VE はマルチスレッドの SIMD プロセッサーです。コンパイラーは、ハードウェア・スレッド内で同時に実行される複数の work-item をマップする SIMD コードを生成します。カーネルの SIMD 幅は、ヒューリスティックによりコンパイラーが選択します。一般的な SIMD 幅は、SIMD-8、SIMD-16、SIMD-32 です。

特定の SIMD 幅に対して、スレッド内のすべてのカーネル・インスタンスが同じ命令を実行する場合、SIMD レーンを最大限に活用できます。カーネル・インスタンスの 1 つ以上が分岐先に分岐した場合、スレッドは分岐の 2 つの経路を実行し、マスクによって結果をマージします。VE の分岐ユニットは、このような分岐や分岐の入れ子を記録します。

## スレッドのマッピングと GPU 占有率

SYCL* 実行モデルは、インテル® アーキテクチャーを採用する GPU の抽象化ビューを提供します。SYCL* スレッド階層は、work-item の 1、2、または 3 次元のグリッドで構成されます。これらの work-item は、work-group と呼ばれる同一サイズのスレッドのグループにグループ化されます。そして、work-group のスレッドは、sub-group と呼ばれる同じサイズのベクトルのグループに分割されます。

**work-item (ワーク項目)**
work-item は、カーネルの並列実行のコレクションの 1 つを表します。

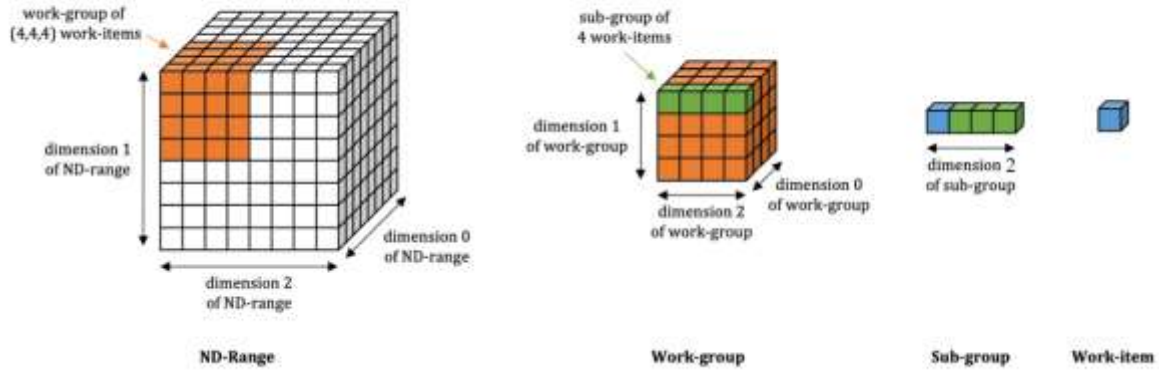
**sub-group (サブグループ)**
sub-group は、レングスが 8、16、32 の SIMD ベクトル、またはインテル® UHD グラフィックスを搭載する CPU のネイティブベクトル長の倍数として同時に処理される、短い範囲で連続した work-item を表します。

**work-group (ワークグループ)**
work-group は、スレッド階層内の 1、2、または 3 次元のスレッドのセットです。SYCL* における work-item 間の同期は、同一 work-group 内で work-item のバリアがある場合にのみ可能です。

## nd_range

`nd_range` は、スレッド階層を 1、2、または 3 次元の work-group のグリッドに分割します。これは、各 work-group のローカルレンジであるグローバルレンジとして表現されます。

### スレッド階層




上の図は、nd_range、work-group、sub-group、および work-item の関係を示しています。

## スレッドの同期

SYCL* は、カーネル関数内で呼び出すことができる 2 つの同期メカニズムを提供します。どちらも、同じ work-group 内の work-item に対し定義されます。SYCL* は、`nd_range` 全体のすべての work-item に対するカーネル内でのグローバル同期メカニズムを提供していません。

- **mem_fence** は、work-group 内のすべての work-item 間のグローバルおよびローカル・メモリー・アクセスに対しメモリーフェンスを挿入します。
- **barrier** はメモリーフェンスを挿入して、work-item がその場所に達するまで、work-group 内のすべての work-item の実行をブロックします。

## 最大の占有率を得るように work-group を Xe コアへマップ

この章の残りでは、GPU リソースの占有率を最大化するために必要な、適切な work-group サイズの選択方法について説明します。ここでは、実行ターゲットとして、Xe LP GPU を搭載するインテル® プロセッサー (Tiger Lake†、略称 TGL†) を使用します。この例では、デュアル・サブスライスには Xe コア (XC)、エグゼキューション・ユニットには Xe ベクトルエンジン (XVE) という新しい用語を使用しています。

以下の表に、Xe LP グラフィックス (TGL†) GPU のアーキテクチャー・パラメーターをまとめています。

### Xe LP (TGL†) GPU

| | VE 数 | スレッド数 | 操作 | 最大ワークグループ・サイズ |
|---|---|---|---|---|
| 各 Xe コア | 16 | 7 x 16 = 112 | 112 x 8 = 896 | 512 |
| 合計 | 16 x 6 = 96 | 112 x 6 = 672 | 896 x 6 = 5376 | 512 |

最大 work-group サイズは、ハードウェアと GPU ドライバーによって制限されます。サポートされる work-group の最大サイズは、`device::get_info<cl::sycl::info::device::max_work_group_size>()` で照会できます。

簡単なカーネルの例を見ていきましょう。

```
1.  auto command_group =
2.    [&](auto &cgh) {
3.      cgh.parallel_for(sycl::range<3>(64, 64, 64), // グローバルレンジ
4.        [=](item<3> it) {
5.        // (カーネルコード)
6.    })
7.  }
```

このカーネルには、64 x 64 x 64 の 3D レンジとして構造化された 262,144 個の work-item が含まれます。work-group と sub-group サイズの選択は、コンパイラーに任されます。GPU スライスで使用可能な 5,376 の並列操作を完全に利用するには、コンパイラーが適切な work-group サイズを選択しなければなりません。

最も重要な 2 つの GPU リソースは以下です。

- スレッド・コンテキスト: カーネルには、GPU のスレッド・コンテキストを活用するのに十分な数のスレッドが必要です。
- SIMD ユニットと SIMD レジスター: カーネルは、work-item をベクトル化し、SIMD レジスターを利用するように構成する必要があります。

SYCL* カーネルでは、プログラマーはカーネルを適切な work-group サイズ、sub-group サイズに構造化し、効率良いベクトル実行のため work-item を構成することで、ワークの分散を調整できます。効率良いベクトルカーネルの記述方法については、別の節で説明します。この章では work-group と sub-group のサイズの選択にのみ焦点を当てます。

スレッド・コンテキストは SIMD ベクトルよりも容易に利用できます。したがって、work-group 内のスレッド数を選択することから始めます。各 $X^e$ コアには 112 のスレッド・コンテキストがありますが、カーネルが 8 でベクトル化されている場合 (112 x 8 = 896 > 512)、すべてのスレッドを利用することができません。このことから、work-group 内のスレッドの最大数は 64 (512/8) であることが分かります。

SYCL* は work-group 内のスレッド数を直接設定するメカニズムを持っていませんが、work-group サイズと sub-group サイズからスレッド数を求めることができます。

**work-group サイズ = スレッド数 x sub-group サイズ**

カーネルに十分な数のレジスターがある限り、sub-group のサイズを増やすことができます。各 VE には 8 つの 128 ビット SIMD レジスターがあることに留意してください。sub-group サイズを増やす効果はループのアンロールに似ています。各 VE は、サイクルごとに 8 つの 32 ビット操作を実行しますが、work-group の相互作用ごとのワーク量は 2 倍もしくは 4 倍になります。SYCL* では、プログラマーは `intel::reqd_sub_group_size({8|16|32})` を使用して sub-group のサイズを明示的に指定し、コンパイラーの選択をオーバーライドできます。

次の表は、スレッド数の選択基準、インテル® Iris® Xe LP GPU のすべての GPU リソースを占有できる sub-group のサイズをまとめたものです。

**完全な占有率を達成する構成**

| 最大スレッド数 | 最小サブグループ・サイズ | 最大サブグループ・サイズ | 最大ワークグループ・サイズ | 条件 |
|---|---|---|---|---|
| 64 | 8 | 32 | 512 | スレッド数 × サブグループ・サイズ <= 512 |

一般に、大きな work-group サイズを選択すると、work-group をディスパッチする回数を減らすことができる利点があります。sub-group のサイズを大きくすると、work-group で必要なスレッド数を減らすことができますが、sub-group ごとの待機時間が長くなり、レジスターの負荷が増加します。

## work-group 内での work-item の同期の影響

work-item の同期を必要とするカーネルの例を示します。

```
1.  auto command_group =
2.    [&](auto &cgh) {
3.    cgh.parallel_for(nd_range(sycl::range(64, 64, 128), // グローバルレンジ
4.                              sycl::range(1, R, 128)    // ローカルレンジ
5.    ),
6.    [=](sycl::nd_item<3> item) {
7.        // (カーネルコード)
8.        // 内部初期化
9.        item.barrier(access::fence_space::global_space);
10.       // (カーネルコード)
11.   })
12. }
```

このカーネルは、work-group バリアの同期が必要なことを除けば前の例と似ています。work-item の同期は、同じ work-group 内の work-item 間でのみ利用できます。プログラマーは、`nd_range` と `nd_item` を使用して work-group のローカルレンジを選択する必要があります。work-group のすべての work-item は同じ Xe コアに割り当てられるため、Xe コアの占有率とカーネルのパフォーマンスに影響します。

このカーネルの work-group のローカルレンジは、`range(1, R, 128)` として与えられます。sub-group サイズが 8 であると仮定すると、変数 `R` の値が VE の占有率にどのように影響するか確認してみます。`R=1` の場合、ローカルグループのレンジは (1, 1, 128) で、work-group のサイズは 128 です。使用可能なスレッド・コンテキストは 112 ありますが、work-group に割り当てられた Xe コアには 16 スレッドしか含まれていません (占有率が非常に低い)。ただし、システムは 7 つの work-group を同じ Xe コアにディスパッチすることで、ディスパッチ数を増やすことと引き換えにスレッドの占有率を高めることができます。

`R>4` の場合、work-group のサイズはシステムでサポートされる最大 work-group サイズ 512 を超えるため、カーネルの起動に失敗します。`R=4` の場合、Xe コアは 57% しか占有されておらず (4/7)、3 つの未使用スレッド・コンテキストは別の work-group に対応するには十分でなく、利用可能な VE の 43% が浪費されています。ドライバーは、未使用の Xe コアに部分的な work-group をディスパッチできる可能性があることに注意してください。ただし、カーネルにはバリアがあるため、部分的にディスパッチされた work-item は残りの work-group がディスパッチされるまでバリアを通過できません。多くの場合、カーネルは部分的なディスパッチによるパフォーマンスの恩恵を受けることはありません。したがって、work-group のサイズを適切に選択することが、この問題の回避につながります。

次の表は、グループサイズ、スレッド数、Xe コアの利用率、および占有率間のトレードオフをまとめたものです。

**各種構成での利用**

| work-item 数 | グループサイズ | スレッド数 | Xe コア 利用率 | Xe コア 占有率 |
|---|---|---|---|---|
| 64 x 64 x 128 = 524288 | (R=1) 128 | 16 | 16 / 112 = 14% | 100% (7 work-group) |
| 64 x 64 x 128 = 524288 | (R=2) 128 x 2 | 2 x 16 = 32 | 32 / 112 = 28.6% | 86% (3 work-group) |
| 64 x 64 x 128 = 524288 | (R=3) 128 x 4 | 3 x 16 = 48 | 49 / 112 = 42.9% | 86% (2 work-group) |
| 64 x 64 x 128 = 524288 | (R=4) 128 x 4 | 4 x 16 = 64 | 64 / 112 = 57% | 最大 57% |
| 64 x 64 x 128 = 524288 | (R>4) 640+ | | | 起動の失敗 |

## work-group 内ローカルメモリーの影響

カーネルが work-group にローカルメモリーを割り当てる例を見てみましょう:

```
1.  auto command_group =
2.    [&](auto &cgh) {
3.      // ワーク項目間で共有されるローカルメモリー変数
4.      sycl::accessor<int, 1, sycl::access::mode::read_write,
5.        sycl::access::target::local>
6.        myLocal(sycl::range(R), cgh);
7.      cgh.parallel_for(nd_range(sycl::range<3>(64, 64, 128), // グローバルレンジ
8.        sycl::range<3>(1, R, 128)     // ローカルレンジ
9.      ),
10.     [=](ngroup<3> myGroup) {
11.       // (work-group コード)
12.       myLocal[myGroup.get_local_id()[1]] = ...
13.   })
14. }
```

work-group のローカル変数は work-item 間で共有されるため、Xe コアの SLM に割り当てることができます。したがって、この work-group は、グループ内の同期と同様に単一の Xe コアに割り当てる必要があります。さらに、ローカル変数が Xe コア SLM の 128KB 容量内に収まるよう、さまざまなグループサイズでローカル変数のサイズを比較して検討する必要があります。

## 詳細例

このセクションの最後では、単純なベクトル加算の例のバリエーションからハードウェアの占有率を検証してみます。Xe LP (TGL†) GPU で指定されるリソース・パラメーターで、ベースとなるハードウェアとしてインテル® Iris® Xe グラフィックス (TGL†) を使用します。

```
1.  auto d_selector = sycl::default_selector_v;
2.
3.  // この例の配列タイプとデータサイズ
4.  constexpr size_t array_size = 3 * 5 * 7 * (1 << 17);
5.  typedef std::array<int, array_size> IntArray;
6.
7.  #define mysize (1 << 17)
8.
9.  int VectorAdd1(sycl::queue &q, const IntArray &a, const IntArray &b,
10.                 IntArray &sum, int iter) {
```

```
11.   sycl::range num_items{a.size()};
12.
13.   sycl::buffer a_buf(a);
14.   sycl::buffer b_buf(b);
15.   sycl::buffer sum_buf(sum.data(), num_items);
16.
17.   auto start = std::chrono::steady_clock::now();
18.   auto e = q.submit([&](auto &h) {
19.     // 入力アクセサー
20.     sycl::accessor a_acc(a_buf, h, sycl::read_only);
21.     sycl::accessor b_acc(b_buf, h, sycl::read_only);
22.     // 出力アクセサー
23.     sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
24.
25.     h.parallel_for(num_items, [=](auto i) {
26.       for (int j = 0; j < iter; j++)
27.         sum_acc[i] = a_acc[i] + b_acc[i];
28.     });
29.   });
30.   q.wait();
31.   auto end = std::chrono::steady_clock::now();
32.   std::cout << "VectorAdd1 completed on device - took " << (end - start).count()
33.             << " u-secs\n";
34.   return ((end - start).count());
35. } // VectorAdd1 の終わり
36.
37. template <int groups>
38. int VectorAdd2(sycl::queue &q, const IntArray &a, const IntArray &b,
39.                IntArray &sum, int iter) {
40.   sycl::range num_items{a.size()};
41.
42.   sycl::buffer a_buf(a);
43.   sycl::buffer b_buf(b);
44.   sycl::buffer sum_buf(sum.data(), num_items);
45.   size_t num_groups = groups;
46.   size_t wg_size = 512;
47.   // size_t wg_size = 512 の代わりに最大 wg_size 512 を取得
48.   auto start = std::chrono::steady_clock::now();
49.   q.submit([&](auto &h) {
50.     // 入力アクセサー
51.     sycl::accessor a_acc(a_buf, h, sycl::read_only);
52.     sycl::accessor b_acc(b_buf, h, sycl::read_only);
53.     // 出力アクセサー
54.     sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
55.
56.     h.parallel_for(
57.         sycl::nd_range<1>(num_groups * wg_size, wg_size),
58.         [=](sycl::nd_item<1> index) [[intel::reqd_sub_group_size(32)]] {
59.           size_t grp_id = index.get_group()[0];
60.           size_t loc_id = index.get_local_id();
61.           size_t start = grp_id * mysize;
62.           size_t end = start + mysize;
63.           for (int j = 0; j < iter; j++)
64.             for (size_t i = start + loc_id; i < end; i += wg_size) {
65.               sum_acc[i] = a_acc[i] + b_acc[i];
66.             }
67.         });
68.   });
69.   q.wait();
70.   auto end = std::chrono::steady_clock::now();
71.   std::cout << "VectorAdd2<" << groups << "> completed on device - took "
72.             << (end - start).count() << " u-secs\n";
73.   return ((end - start).count());
74. } // VectorAdd2 の終わり
75.
76. void InitializeArray(IntArray &a) {
```

```
77.   for (size_t i = 0; i < a.size(); i++)
78.     a[i] = i;
79. }
80.
81. void Initialize(IntArray &a) {
82.   for (size_t i = 0; i < a.size(); i++)
83.     a[i] = 0;
84. }
85. IntArray a, b, sum;
86.
87. int main() {
88.
89.   sycl::queue q(d_selector);
90.
91.   InitializeArray(a);
92.   InitializeArray(b);
93.
94.   std::cout << "Running on device: "
95.             << q.get_device().get_info<sycl::info::device::name>() << "\n";
96.   std::cout << "Vector size: " << a.size() << "\n";
97.
98.   // 結果をチェック
99.   Initialize(sum);
100.  VectorAdd1(q, a, b, sum, 1);
101.
102.  for (int i = 0; i < mysize; i++)
103.    if (sum[i] != 2 * i) {
104.      std::cout << "add1 Did not match\n";
105.    }
106.
107.  Initialize(sum);
108.  VectorAdd2<1>(q, a, b, sum, 1);
109.  for (int i = 0; i < mysize; i++)
110.    if (sum[i] != 2 * i) {
111.      std::cout << "add2 Did not match\n";
112.    }
113.
114.  // time the kernels
115.  Initialize(sum);
116.  int t = VectorAdd1(q, a, b, sum, 1000);
117.  Initialize(sum);
118.  t = VectorAdd2<1>(q, a, b, sum, 1000);
119.  t = VectorAdd2<2>(q, a, b, sum, 1000);
120.  t = VectorAdd2<3>(q, a, b, sum, 1000);
121.  t = VectorAdd2<4>(q, a, b, sum, 1000);
122.  t = VectorAdd2<5>(q, a, b, sum, 1000);
123.  t = VectorAdd2<6>(q, a, b, sum, 1000);
124.  t = VectorAdd2<7>(q, a, b, sum, 1000);
125.  t = VectorAdd2<8>(q, a, b, sum, 1000);
126.  t = VectorAdd2<12>(q, a, b, sum, 1000);
127.  t = VectorAdd2<16>(q, a, b, sum, 1000);
128.  t = VectorAdd2<20>(q, a, b, sum, 1000);
129.  t = VectorAdd2<24>(q, a, b, sum, 1000);
130.  t = VectorAdd2<28>(q, a, b, sum, 1000);
131.  t = VectorAdd2<32>(q, a, b, sum, 1000);
132.  return 0;
133.} // codeblock の終わり
```

上記の VectorAdd1 により、コンパイラーは work-group サイズと SIMD 幅を選択します。この場合、カーネルのレジスター・プレッシャーが低いことから、コンパイラーは work-group サイズ 512 と SIMD 幅 32 を選択します。

```
int VectorAdd2(sycl::queue &q, const IntArray &a, const IntArray &b,
               IntArray &sum, int iter) {
  sycl::range num_items{a.size()};
  sycl::buffer a_buf(a);
  sycl::buffer b_buf(b);
  sycl::buffer sum_buf(sum.data(), num_items);
  size_t num_groups = groups;
  size_t wg_size = 512;
  // size_t wg_size = 512 の代わりに最大 wg_size 512 を取得

  auto start = std::chrono::steady_clock::now();
  q.submit([&](auto &h) {
    // 入力アクセサー
    sycl::accessor a_acc(a_buf, h, sycl::read_only);
    sycl::accessor b_acc(b_buf, h, sycl::read_only);
    // 出力アクセサー
    sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
    h.parallel_for(
      sycl::nd_range<1>(num_groups * wg_size, wg_size),
      [=](sycl::nd_item<1> index) [[intel::reqd_sub_group_size(32)]] {
        size_t grp_id = index.get_group()[0];
        size_t loc_id = index.get_local_id();
        size_t start = grp_id * mysize;
        size_t end = start + mysize;
        for (int j = 0; j < iter; j++)
          for (size_t i = start + loc_id; i < end; i += wg_size) {
            sum_acc[i] = a_acc[i] + b_acc[i];
          }
    });
  });
  q.wait();
  auto end = std::chrono::steady_clock::now();
  std::cout << "VectorAdd2<" << groups << "> completed on device - took "
            << (end - start).count() << " u-secs\n";
  return ((end - start).count());
} // VectorAdd2 の終わり
```

上記の VectorAdd2 の例では、関数パラメーター・グループとして、work-group サイズ 512、SIMD 幅 32、および可変数の work-group を明示的に指定しています。

スレッド数を GPU で利用可能なスレッド・コンテキスト数で割ると、GPU ハードウェアの占有率を推測することができます。次の表は、さまざまな引数の組み合わせで前述の 2 つのカーネルに、インテル® Iris® Xᵉ アーキテクチャー (TGL†) パラメーターを指定して GPU ハードウェアの占有率を計算しています。

### 占有率

| プログラム占有率 | work-group 数 | work-item 数 | work-group サイズ | SIMD | work-group のスレッド数 | スレッド数 | 占有率 |
|---|---|---|---|---|---|---|---|
| VectorAdd1 | 53760 | 27.5M | 512 | 32 | 16 | 860K | 100% |
| VectorAdd2<1> | 1 | 512 | 512 | 32 | 16 | 16 | 16/672 = 2.4% |
| VectorAdd2<2> | 2 | 1024 | 512 | 32 | 16 | 32 | 32/672 = 4.8% |
| VectorAdd2<3> | 3 | 1536 | 512 | 32 | 16 | 48 | 48/672 = 7.1% |
| VectorAdd2<4> | 4 | 2048 | 512 | 32 | 16 | 64 | 64/672 = 9.5% |
| VectorAdd2<5> | 5 | 2560 | 512 | 32 | 16 | 80 | 80/672 = 11.9% |
| VectorAdd2<6> | 6 | 3072 | 512 | 32 | 16 | 96 | 96/672 = 14.3% |
| VectorAdd2<7> | 7 | 3584 | 512 | 32 | 16 | 112 | 112/672 = 16.7% |
| VectorAdd2<8> | 8 | 4096 | 512 | 32 | 16 | 128 | 128/672 = 19% |
| VectorAdd2<12> | 12 | 6144 | 512 | 32 | 16 | 192 | 192/672 = 28.6% |
| VectorAdd2<16> | 16 | 8192 | 512 | 32 | 16 | 256 | 256/672 = 38.1% |
| VectorAdd2<20> | 20 | 10240 | 512 | 32 | 16 | 320 | 320/672 = 47.7% |
| VectorAdd2<24> | 24 | 12288 | 512 | 32 | 16 | 384 | 384/672 = 57.1% |
| VectorAdd2<28> | 28 | 14336 | 512 | 32 | 16 | 448 | 448/672 = 66.7% |
| VectorAdd2<32> | 32 | 16384 | 512 | 32 | 16 | 512 | 512/672 = 76.2% |
| VectorAdd2<36> | 36 | 18432 | 512 | 32 | 16 | 576 | 576/672 = 85.7% |
| VectorAdd2<40> | 40 | 20480 | 512 | 32 | 16 | 640 | 640/672 = 95.2% |
| VectorAdd2<42> | 42 | 21504 | 512 | 32 | 16 | 672 | 672/672 = 100% |
| VectorAdd2<44> | 44 | 22528 | 512 | 32 | 16 | 704 | 100% 次に 4.7% |
| VectorAdd2<48> | 48 | 24576 | 512 | 32 | 16 | 768 | 100% 次に 14.3% |

次のインテル® VTune™ プロファイラーの `VectorAdd2` のグラフで、さまざまな work-group サイズの予測精度を確認します。グリッドビューでは実行全体の平均が示されるため、グリッドビューの数値は予測と異なることがあります。

### インテル® VTune™ プロファイラーで示される VectorAdd2 の占有率

| Computing Task | Work Size | | Computing Task | | | | | Data Transferred | | EU Array | | | EU Threads Occupancy |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Global | Local | Total Time | Average Time | Instance Count | SIMD Width | S... | Size | Total, GB/sec | Active | Stalled | Idle | |
| [Outside any task] | | | 0s | | | | | | | 0.0% | 0.2% | 99.7% | 0.2% |
| VectorAdd2<(int)1>(int, cl::sycl::queue&, std::a... | 512 | 512 | 0.046s | 0.046s | 1 | 32 | | 0 B | 0.000 | 4.9% | 11.6% | 83.5% | 2.4% |
| VectorAdd2<(int)2>(int, cl::sycl::queue&, std::a... | 1024 | 512 | 0.056s | 0.056s | 1 | 32 | | 0 B | 0.000 | 7.7% | 8.9% | 83.4% | 4.4% |
| VectorAdd2<(int)3>(int, cl::sycl::queue&, std::a... | 1536 | 512 | 0.060s | 0.060s | 1 | 32 | | 0 B | 0.000 | 7.9% | 8.6% | 83.4% | 7.1% |
| VectorAdd2<(int)4>(int, cl::sycl::queue&, std::a... | 2048 | 512 | 0.087s | 0.087s | 1 | 32 | | 0 B | 0.000 | 9.9% | 22.2% | 67.9% | 9.3% |
| VectorAdd2<(int)5>(int, cl::sycl::queue&, std::a... | 2560 | 512 | 0.089s | 0.089s | 1 | 32 | | 0 B | 0.000 | 12.1% | 20.4% | 67.5% | 11.6% |
| VectorAdd2<(int)6>(int, cl::sycl::queue&, std::a... | 3072 | 512 | 0.094s | 0.094s | 1 | 32 | | 0 B | 0.000 | 13.5% | 18.6% | 67.9% | 13.8% |
| VectorAdd2<(int)7>(int, cl::sycl::queue&, std::a... | 3584 | 512 | 0.101s | 0.101s | 1 | 32 | | 0 B | 0.000 | 14.9% | 33.4% | 51.8% | 16.0% |
| VectorAdd2<(int)8>(int, cl::sycl::queue&, std::a... | 4096 | 512 | 0.111s | 0.111s | 1 | 32 | | 0 B | 0.000 | 15.6% | 32.7% | 51.7% | 18.2% |
| VectorAdd2<(int)12>(int, cl::sycl::queue&, std::... | 6144 | 512 | 0.243s | 0.243s | 1 | 32 | | 0 B | 0.000 | 10.7% | 55.6% | 33.8% | 28.1% |
| VectorAdd2<(int)16>(int, cl::sycl::queue&, std::... | 8192 | 512 | 0.416s | 0.416s | 1 | 32 | | 0 B | 0.000 | 8.4% | 70.7% | 20.9% | 36.2% |
| VectorAdd2<(int)20>(int, cl::sycl::queue&, std::... | 10240 | 512 | 0.599s | 0.599s | 1 | 32 | | 0 B | 0.000 | 7.5% | 86.9% | 5.7% | 45.4% |
| VectorAdd2<(int)24>(int, cl::sycl::queue&, std::... | 12288 | 512 | 0.842s | 0.842s | 1 | 32 | | 0 B | 0.000 | 6.5% | 88.4% | 5.0% | 54.4% |
| VectorAdd2<(int)28>(int, cl::sycl::queue&, std::... | 14336 | 512 | 0.944s | 0.944s | 1 | 32 | | 0 B | 0.000 | 6.8% | 83.4% | 9.8% | 60.9% |
| VectorAdd2<(int)32>(int, cl::sycl::queue&, std::... | 16384 | 512 | 1.177s | 1.177s | 1 | 32 | | 0 B | 0.000 | 6.3% | 85.6% | 8.1% | 69.7% |
| VectorAdd2<(int)36>(int, cl::sycl::queue&, std::... | 18432 | 512 | 1.336s | 1.336s | 1 | 32 | | 0 B | 0.000 | 6.3% | 89.8% | 3.9% | 80.1% |
| VectorAdd2<(int)40>(int, cl::sycl::queue&, std::... | 20480 | 512 | 1.467s | 1.467s | 1 | 32 | | 0 B | 0.000 | 6.4% | 92.6% | 1.0% | 87.4% |
| VectorAdd2<(int)48>(int, cl::sycl::queue&, std::... | 24576 | 512 | 1.783s | 1.783s | 1 | 32 | | 0 B | 0.000 | 6.2% | 93.8% | 0.0% | 88.3% |
| VectorAdd2<(int)44>(int, cl::sycl::queue&, std::... | 22528 | 512 | 1.648s | 1.648s | 1 | 32 | | 0 B | 0.000 | 6.3% | 92.8% | 1.0% | 89.3% |
| VectorAdd2<(int)42>(int, cl::sycl::queue&, std::... | 21504 | 512 | 1.568s | 1.568s | 1 | 32 | | 0 B | 0.000 | 6.3% | 93.7% | 0.0% | 90.8% |

次のタイムライン・ビューは一定期間の占有率を示しています。占有率メトリックは、カーネル実行の大部分で正確ですが、各スレッドが実行を終了する時間が異なるため後半に向かって減少していきます。

### VectorAdd2 のタイムライン・ビュー

次に示す `VectorAdd3` カーネルは上記のカーネルに似ていますが、2 つの点で異なります。

1. テンプレート・パラメーターとして、work-group 数、work-group サイズ、および sub-group サイズをインスタンス化できます。これにより、スレッドの占有率に対する sub-group と work-group 数の影響を調査することができます。
2. 長時間のワークがエグゼキューション・ユニットに常駐するようにすることで、カーネル内のワーク量が劇的に増加します。

```
1.  template <int groups, int wg_size, int sg_size>
2.            int VectorAdd3(sycl::queue &q, const IntArray &a, const IntArray &b,
3.    IntArray &sum, int iter) {
4.    sycl::range num_items{a.size()};
5.    sycl::buffer a_buf(a);
6.    sycl::buffer b_buf(b);
7.    sycl::buffer sum_buf(sum.data(), num_items);
8.    size_t num_groups = groups;
9.    auto start = std::chrono::steady_clock::now();
10.   q.submit([&](auto &h) {
11.     // 入力アクセサー
12.     sycl::accessor a_acc(a_buf, h, sycl::read_only);
13.     sycl::accessor b_acc(b_buf, h, sycl::read_only);
14.     // 出力アクセサー
15.     sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
16.     h.parallel_for(
17.       sycl::nd_range<1>(num_groups * wg_size, wg_size),
18.       [=](sycl::nd_item<1> index) [[intel::reqd_sub_group_size(sg_size)]] {
19.       size_t grp_id = index.get_group()[0];
20.       size_t loc_id = index.get_local_id();
21.       size_t start = grp_id * mysize;
22.       size_t end = start + mysize;
23.       for (int j = 0; j < iter; j++)
24.         for (size_t i = start + loc_id; i < end; i += wg_size) {
25.           sum_acc[i] = a_acc[i] + b_acc[i];
26.         }
27.     });
28.   });
29.   q.wait();
30.   auto end = std::chrono::steady_clock::now();
31.   std::cout << "VectorAdd3<" << groups << "> completed on device - took "
32.             << (end - start).count() << " u-secs\n";
33.   return ((end - start).count());
34. } // VectorAdd3 の終わり
```

`VectorAdd4` カーネルは、カーネル実行の開始時と終了時にバリア同期があることを除いて、前述の `VectorAdd3` カーネルに類似しています。このバリアは機能的には必要ありませんが、ハードウェアでスレッドをスケジュールするのに大きく影響します。

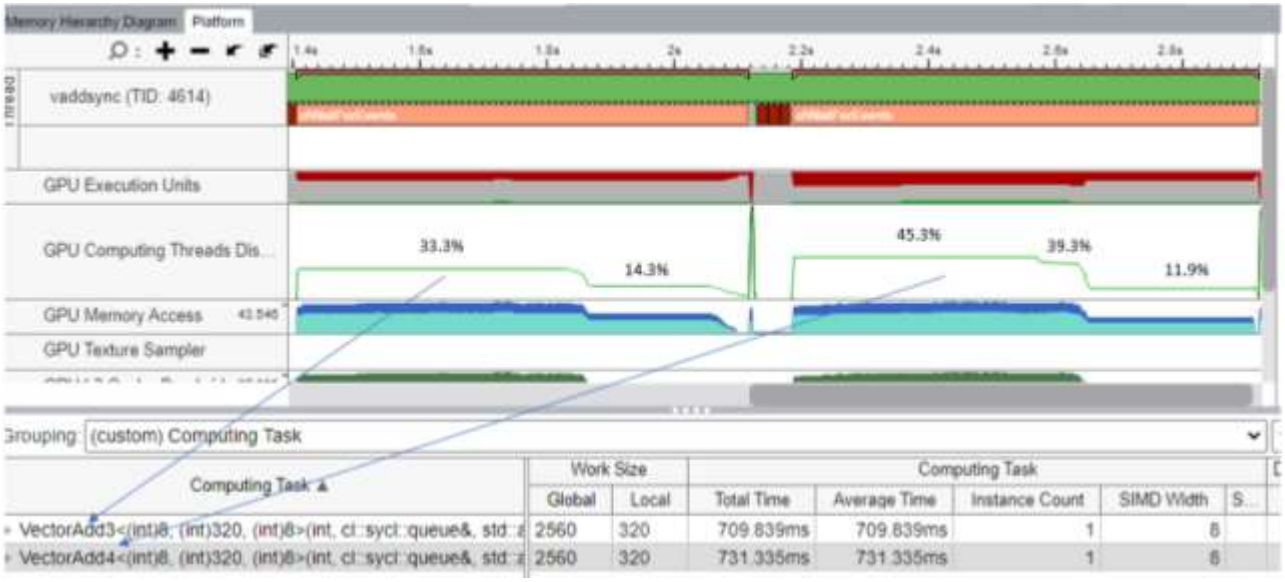
```
1.  template <int groups, int wg_size, int sg_size>
2.    int VectorAdd4(sycl::queue &q, const IntArray &a, const IntArray &b,
3.    IntArray &sum, int iter) {
4.      sycl::range num_items{a.size()};
5.      sycl::buffer a_buf(a);
6.      sycl::buffer b_buf(b);
7.      sycl::buffer sum_buf(sum.data(), num_items);
8.      size_t num_groups = groups;
9.      auto start = std::chrono::steady_clock::now();
10.     q.submit([&](auto &h) {
11.       // 入力アクセサー
12.       sycl::accessor a_acc(a_buf, h, sycl::read_only);
13.       sycl::accessor b_acc(b_buf, h, sycl::read_only);
14.       // 出力アクセサー
15.       sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
16.       h.parallel_for(
17.         sycl::nd_range<1>(num_groups * wg_size, wg_size),
18.         [=](sycl::nd_item<1> index) [[intel::reqd_sub_group_size(sg_size)]] {
19.           index.barrier(sycl::access::fence_space::local_space);
20.           size_t grp_id = index.get_group()[0];
21.           size_t loc_id = index.get_local_id();
22.           size_t start = grp_id * mysize;
23.           size_t end = start + mysize;
24.           for (int j = 0; j < iter; j++) {
25.             for (size_t i = start + loc_id; i < end; i += wg_size) {
26.               sum_acc[i] = a_acc[i] + b_acc[i];
27.             }
28.           }
29.       });
30.     });
31.     q.wait();
32.   auto end = std::chrono::steady_clock::now();
33.   std::cout << "VectorAdd4<" << groups << "> completed on device - took "
34.             << (end - start).count() << " u-secs\n";
35.   return ((end - start).count());
36. } // VectorAdd4 の終わり
```

スレッドのスケジュール方法を説明するため、上記の 2 つのカーネルが、次のように 8 つの work-group、sub-group サイズ 8、および work-group サイズ 320 で呼び出されることを考えてみます。work-group サイズと sub-group サイズから、ハードウェアは work-group ごとに 40 スレッドをスケジュールします。

```
Initialize(sum);
VectorAdd3<8, 320, 8>(q, a, b, sum, 10000); Initialize(sum);
VectorAdd4<8, 320, 8>(q, a, b, sum, 10000);
```

次のインテル® VTune™ プロファイラーによるグラフは、`VectorAdd3` と `VectorAdd4` カーネルで測定された占有率を示します。

### VectorAdd3、VectorAdd4 カーネルの GPU 占有率




`VectorAdd3` カーネルでは、占有率に 2 つのフェーズがあることが分かります。合計 672 スレッドの TGL† マシンで、最初が 33.3% (224 スレッドを占有) で、その次は 14.3% (95 スレッドを占有) です。それぞれ 40 スレッドを持つ合計 8 つの work-group があることが判明しているため、6 つの work-group のスレッドがスケジュールされる 2 つのサブスライス (それぞれ 112 スレッド) があると考えることができます。これは、最初のフェーズでは、4 つの work-group にそれぞれの 40 スレッドがスケジュールされ、2 つの work-group に 32 スレッドがスケジュールされていることを意味します。それに続く 2 番目のフェーズでは、残りの 2 つの work-group に 40 スレッドがスケジュールされています。

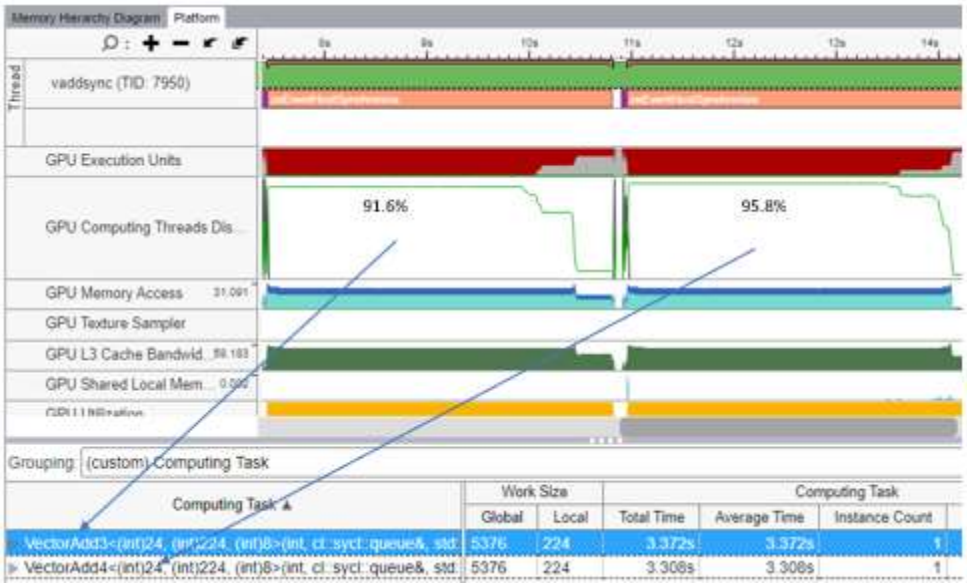
`VectorAdd4` カーネルでは、占有率は 45.3% (304 スレッド)、39.3% (264 スレッド)、および 11.9% (80 スレッド) の 3 つのフェーズがあります。最初のフェーズでは 8 つの work-group すべてが 3 つのサブスライスでスケジュールされています。2 つのサブスライスにはそれぞれ 112 スレッド (2 つの work-group で 80、1 つで 32) が割り当てられ、1 つのサブスライスは 80 スレッドが割り当てられます (2 つの work-group から)。2 番目のフェーズでは、1 つの work-group が実行を完了し、占有率は (304 - 40 = 264) となります。そして最後のフェーズで、2 つの work-group の残りの 8 スレッドがスケジュールされ実行が完了します。

上記のカーネルをサブスライス内のスレッドがスレッド数の倍数になるように構成された、異なる work-group サイズで実行した場合、以下の図のようにより多くの work-group がハードウェアを十分に活用してほぼ 100% の占有率を達成することができます。

```
Initialize(sum);
VectorAdd3<24, 224, 8>(q, a, b, sum, 10000); Initialize(sum);
VectorAdd4<24, 224, 8>(q, a, b, sum, 10000);
```

このカーネルの実行は、スレッド数が多く、work-group サイズが $X^e$ コア内のスレッド数の倍数であるため、スレッドの占有率は異なります。この様子を以下のインテル® VTune™ プロファイラーにおけるスレッド占有率のメトリックに示します。

**インテル® VTune™ プロファイラーが示すスレッド占有率のメトリック:**




スライスごとの占有率を調査する方法がまだないため、上記のスケジュールは、異なる占有率に基づく推測であることに注意してください。

上記のカーネルをさまざまな環境と設定の組み合わせで実行し、GPU ハードウェアがエグゼキューション・ユニットにソフトウェア・スレッドをどのようにスケジュールするか理解することができます。GPU ハードウェアを効率良く利用するには、work-group の数に加えて、work-group と sub-group のサイズに注意する必要があります。

## インテル® GPU の占有率計算

要約すると、SYCL* work-group は通常、$X^e$ コアにディスパッチされ、work-group 内のすべての work-item は、work-group 内のスレッドバリアとメモリーフェンスによる同期のため、$X^e$ コアの同じ SLM を共有します。VE ALU、SLM、およびスレッド・コンテキストが十分にあれば、複数の work-group を同じ $X^e$ コアにディスパッチできます。

利用可能なすべての $X^e$ コアを完全に利用することで、高いパフォーマンスを得ることができます。カーネルの GPU 占有率に影響するパラメーターは、work-group サイズと SIMD sub-group サイズであり、これは work-group 内のスレッド数を決定します。

インテル® GPU の占有率計算 (英語) を使用すると、指定されたカーネルとその work-group パラメーターに対するインテル® GPU の占有率を計算できます。

# カーネル

カーネルは oneAPI オフロードモデルの計算単位です。反復空間のカーネルを送信することで、指定するデータ・オブジェクトに計算を適用することを要求します。

このセクションでは、カーネルのコーディング、送信、および実行に関連するトピックについて説明します。


- sub-group と SIMD ベクトル化
- 条件チェックを排除
- レジスターの使用とスピルの回避
- レジスタープレッシャーの高いコードのインテル® データセンター GPU マックス・シリーズへの移植
- スモール・レジスター・モードとラージ・レジスター・モード
- 共有ローカルメモリー
- ポインターのエイリアシングと restrict ディレクティブ
- カーネル内のスレッド間での同期
- work-group サイズを選択する考慮事項
- プリフェッチ
- リダクション
- カーネルの起動
- デバイス上で同時に複数のカーネルを実行
- 複数のキューへカーネルを送信
- 冗長なキューの構築を避ける
- SYCL* 結合行列拡張を使用したインテル® XMX のプログラミング
- カーネル内での I/O


## sub-group と SIMD ベクトル化

ND-Range カーネルのインデックス空間は、work-group、sub-group、および work-item に分割されます。work-item は基本単位です。work-item の集合は sub-group を形成し、sub-group の集合は work-group を形成します。work-item および work-group のハードウェアベクトルエンジン (VE) へのマッピングは、実装に依存します。すべての work-group は同時に実行されますが、リソースの可用性に応じて異なるタイミングで実行するようにスケジュールされることもあります。work-group の実行は、ベースとなるハードウェアの能力に応じて、プリエンプトされる場合と、されない場合があります。同じ work-group 内の work-item は、同時に実行されることが保証されています。同じ sub-group 内の work-item は追加のスケジュールが保証されており、追加機能を利用できることがあります。

sub-group は、同じ VE スレッドで実行されるグローバル・インデックス空間で連続する work-item の集合です。デバイス・コンパイラーがカーネルをコンパイルする場合、複数の work-item がベクトル化によって sub-group にパックされるため、生成される SIMD 命令ストリームは複数の work-item を同時に実行できます。work-item を sub-group にパーティション化する場合、その粒度により大きなパフォーマンスの差が生じる可能性があります。

sub-group を示す簡単な例を見てみましょう。

```
1.  q.submit([&](auto &h) {
2.    sycl::stream out(65536, 256, h);
3.    h.parallel_for(sycl::nd_range(sycl::range{32}, sycl::range{32}),
4.      [=](sycl::nd_item<1> it) {
5.        int groupId = it.get_group(0);
6.        int globalId = it.get_global_linear_id();
7.        auto sg = it.get_sub_group();
8.        int sgSize = sg.get_local_range()[0];
9.        int sgGroupId = sg.get_group_id()[0];
10.       int sgId = sg.get_local_id()[0];
11.       out << "globalId = " << sycl::setw(2) << globalId
12.           << " groupId = " << groupId
13.           << " sgGroupId = " << sgGroupId << " sgId = " << sgId
14.           << " sgSize = " << sycl::setw(2) << sgSize
15.           << sycl::endl;
16.   );
17. });
```

この例の出力は次のようになります。

```
Device: Intel(R) Gen12HP
globalId = 0 groupId = 0 sgGroupId = 0 sgId = 0 sgSize = 16
globalId = 1 groupId = 0 sgGroupId = 0 sgId = 1 sgSize = 16
globalId = 2 groupId = 0 sgGroupId = 0 sgId = 2 sgSize = 16
globalId = 3 groupId = 0 sgGroupId = 0 sgId = 3 sgSize = 16
globalId = 4 groupId = 0 sgGroupId = 0 sgId = 4 sgSize = 16
globalId = 5 groupId = 0 sgGroupId = 0 sgId = 5 sgSize = 16
globalId = 6 groupId = 0 sgGroupId = 0 sgId = 6 sgSize = 16
globalId = 7 groupId = 0 sgGroupId = 0 sgId = 7 sgSize = 16
globalId = 16 groupId = 0 sgGroupId = 1 sgId = 0 sgSize = 16
globalId = 17 groupId = 0 sgGroupId = 1 sgId = 1 sgSize = 16
globalId = 18 groupId = 0 sgGroupId = 1 sgId = 2 sgSize = 16
globalId = 19 groupId = 0 sgGroupId = 1 sgId = 3 sgSize = 16
globalId = 20 groupId = 0 sgGroupId = 1 sgId = 4 sgSize = 16
globalId = 21 groupId = 0 sgGroupId = 1 sgId = 5 sgSize = 16
globalId = 22 groupId = 0 sgGroupId = 1 sgId = 6 sgSize = 16
globalId = 23 groupId = 0 sgGroupId = 1 sgId = 7 sgSize = 16
globalId = 8 groupId = 0 sgGroupId = 0 sgId = 8 sgSize = 16
globalId = 9 groupId = 0 sgGroupId = 0 sgId = 9 sgSize = 16
globalId = 10 groupId = 0 sgGroupId = 0 sgId = 10 sgSize = 16
globalId = 11 groupId = 0 sgGroupId = 0 sgId = 11 sgSize = 16
globalId = 12 groupId = 0 sgGroupId = 0 sgId = 12 sgSize = 16
globalId = 13 groupId = 0 sgGroupId = 0 sgId = 13 sgSize = 16
globalId = 14 groupId = 0 sgGroupId = 0 sgId = 14 sgSize = 16
globalId = 15 groupId = 0 sgGroupId = 0 sgId = 15 sgSize = 16
globalId = 24 groupId = 0 sgGroupId = 1 sgId = 8 sgSize = 16
globalId = 25 groupId = 0 sgGroupId = 1 sgId = 9 sgSize = 16
globalId = 26 groupId = 0 sgGroupId = 1 sgId = 10 sgSize = 16
globalId = 27 groupId = 0 sgGroupId = 1 sgId = 11 sgSize = 16
globalId = 28 groupId = 0 sgGroupId = 1 sgId = 12 sgSize = 16
globalId = 29 groupId = 0 sgGroupId = 1 sgId = 13 sgSize = 16
globalId = 30 groupId = 0 sgGroupId = 1 sgId = 14 sgSize = 16
globalId = 31 groupId = 0 sgGroupId = 1 sgId = 15 sgSize = 16
```

この例の sub-group は、work-item 数 16 または sub-group サイズ 16 を持っています。これは、各スレッドが同時に 16 個の work-item を実行し、32 個の work-item が 2 つの VE スレッドで実行されることを意味しています。

デフォルトでは、コンパイラーはデバイス固有の情報とヒューリスティックから sub-group サイズを決定します。ユーザーは、カーネル属性 `intel::reqd_sub_group_size` を使用してコンパイラーの選択をオーバーライドすることで、sub-group の最大サイズを変更できます。sub-group サイズを明示的に要求することでパフォーマンスが向上することがあります。

```
1.  q.submit([&](auto &h) {
2.    sycl::stream out(65536, 256, h);
3.    h.parallel_for(sycl::nd_range(sycl::range{32}, sycl::range{32}),
4.      [=](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(32)]] {
5.        int groupId = it.get_group(0);
6.        int globalId = it.get_global_linear_id();
7.        auto sg = it.get_sub_group();
8.        int sgSize = sg.get_local_range()[0];
9.        int sgGroupId = sg.get_group_id()[0];
10.       int sgId = sg.get_local_id()[0];
11.       out << "globalId = " << sycl::setw(2) << globalId
12.           << " groupId = " << groupId
13.           << " sgGroupId = " << sgGroupId << " sgId = " << sgId
14.           << " sgSize = " << sycl::setw(2) << sgSize
15.           << sycl::endl;
16.   });
17. });
```

出力は次のようになります。

```
Device: Intel(R) Gen12HP
globalId = 0 groupId = 0 sgGroupId = 0 sgId = 0 sgSize = 32
globalId = 1 groupId = 0 sgGroupId = 0 sgId = 1 sgSize = 32
globalId = 2 groupId = 0 sgGroupId = 0 sgId = 2 sgSize = 32
globalId = 3 groupId = 0 sgGroupId = 0 sgId = 3 sgSize = 32
globalId = 4 groupId = 0 sgGroupId = 0 sgId = 4 sgSize = 32
globalId = 5 groupId = 0 sgGroupId = 0 sgId = 5 sgSize = 32
globalId = 6 groupId = 0 sgGroupId = 0 sgId = 6 sgSize = 32
globalId = 7 groupId = 0 sgGroupId = 0 sgId = 7 sgSize = 32
globalId = 8 groupId = 0 sgGroupId = 0 sgId = 8 sgSize = 32
globalId = 9 groupId = 0 sgGroupId = 0 sgId = 9 sgSize = 32
globalId = 10 groupId = 0 sgGroupId = 0 sgId = 10 sgSize = 32
globalId = 11 groupId = 0 sgGroupId = 0 sgId = 11 sgSize = 32
globalId = 12 groupId = 0 sgGroupId = 0 sgId = 12 sgSize = 32
globalId = 13 groupId = 0 sgGroupId = 0 sgId = 13 sgSize = 32
globalId = 14 groupId = 0 sgGroupId = 0 sgId = 14 sgSize = 32
globalId = 15 groupId = 0 sgGroupId = 0 sgId = 15 sgSize = 32
globalId = 16 groupId = 0 sgGroupId = 0 sgId = 16 sgSize = 32
globalId = 17 groupId = 0 sgGroupId = 0 sgId = 17 sgSize = 32
globalId = 18 groupId = 0 sgGroupId = 0 sgId = 18 sgSize = 32
globalId = 19 groupId = 0 sgGroupId = 0 sgId = 19 sgSize = 32
globalId = 20 groupId = 0 sgGroupId = 0 sgId = 20 sgSize = 32
globalId = 21 groupId = 0 sgGroupId = 0 sgId = 21 sgSize = 32
globalId = 22 groupId = 0 sgGroupId = 0 sgId = 22 sgSize = 32
globalId = 23 groupId = 0 sgGroupId = 0 sgId = 23 sgSize = 32
globalId = 24 groupId = 0 sgGroupId = 0 sgId = 24 sgSize = 32
globalId = 25 groupId = 0 sgGroupId = 0 sgId = 25 sgSize = 32
globalId = 26 groupId = 0 sgGroupId = 0 sgId = 26 sgSize = 32
globalId = 27 groupId = 0 sgGroupId = 0 sgId = 27 sgSize = 32
globalId = 28 groupId = 0 sgGroupId = 0 sgId = 28 sgSize = 32
globalId = 29 groupId = 0 sgGroupId = 0 sgId = 29 sgSize = 32
globalId = 30 groupId = 0 sgGroupId = 0 sgId = 30 sgSize = 32
globalId = 31 groupId = 0 sgGroupId = 0 sgId = 31 sgSize = 32
```

有効な sub-group のサイズはデバイスによって異なります。デバイスを照会して次の情報を得られます。

```
1.  std::cout << "sub-group Sizes: ";
2.    for (const auto &s :
3.      q.get_device().get_info<sycl::info::device::sub_group_sizes>()) {
4.        std::cout << s << " ";
5.    }
6.  std::cout << std::endl;
```

サポートされる有効な sub-group のサイズは以下です。

```
Device: Intel(R) Gen12HP
Subgroup Sizes: 8 16 32
```

次に、sub-group を使用してパフォーマンスを向上する方法を示します。

## ベクトル化とメモリーアクセス

インテル® グラフィックス・デバイスは複数の VE を搭載しています。各 VE はマルチスレッド SIMD プロセッサーです。コンパイラーは、VE スレッドで同時にされる sub-group に複数の work-item をパックする SIMD 命令を生成します。SIMD 幅 (sub-group サイズ) は、デバイスの特性とヒューリスティックに基づいてコンパイラーによって選択されるか、カーネルによって明示的に要求され、8、16、または 32 にできます。

指定する SIMD 幅で SIMD レーンの使用率を最大化すると、最適な命令パフォーマンスが得られます。1 つ以上のレーン (またはカーネル・インスタンスや work-item) に分かれている場合、スレッドはパスが後続でマージされる前に両方の分岐パスを実行するため、動的な命令数は増加します。SIMD の分散はパフォーマンスに影響します。コンパイラーは分散を最小限化しようとしますが、可能であればソースコードで分散を回避することが有益です。

work-item のメモリーアクセスは、sub-group のメモリーアクセスや SIMD レーンの使い方に影響します。work-item の連続したメモリーアクセスは、多くの場合最適とは言えません。次に例を示します。

```
1.  constexpr int N = 1024 * 1024;
2.  int *data = sycl::malloc_shared<int>(N, q);
3.  auto e = q.submit([&](auto &h) {
4.    h.parallel_for(sycl::nd_range(sycl::range{N / 16}, sycl::range{32}),
5.      [=](sycl::nd_item<1> it) {
6.        int i = it.get_global_linear_id();
7.        i = i * 16;
8.        for (int j = i; j < (i + 16); j++) {
9.          data[j] = -1;
10.       }
11.   });
12. });
13. q.wait();
```

この簡単なカーネルは 1024 x 1024 の整数配列を初期化します。各 work-item は連続した 16 個の整数を初期化します。コンパイラーが sub-group サイズに 16 を選択したと仮定すると、各 sub-group またはスレッドで 256 個の整数が初期化されます。ただし、SIMD レーン 16 のストアはスキャッターされます。

work-item で 16 個の連続する整数を繰り返し初期化する代わりに、単一の SIMD 命令で 16 個の連続する整数を一度に初期化するほうが効果的です。

```
1.  constexpr int N = 1024 * 1024;
2.  int *data = sycl::malloc_shared<int>(N, q);
3.  auto e = q.submit([&](auto &h) {
4.    h.parallel_for(sycl::nd_range(sycl::range{N / 16}, sycl::range{32}),
5.      [=](sycl::nd_item<1> it) {
6.        int i = it.get_global_linear_id();
7.        auto sg = it.get_sub_group();
8.        int sgSize = sg.get_local_range()[0];
9.        i = (i / sgSize) * sgSize * 16 + (i % sgSize);
10.       for (int j = 0; j < sgSize * 16; j += sgSize) {
11.         data[i + j] = -1;
12.       }
13.   });
14. });
```

この例では、メモリー書き込みを行いますが、同じ手法をメモリー読み取りにも適用できます。

```
1.  constexpr int N = 1024 * 1024;
2.  int *data = sycl::malloc_shared<int>(N, q);
3.  int *data2 = sycl::malloc_shared<int>(N, q);
4.  memset(data2, 0xFF, sizeof(int) * N);
5.  auto e = q.submit([&](auto &h) {
6.    h.parallel_for(sycl::nd_range(sycl::range{N / 16}, sycl::range{32}),
7.      [=](sycl::nd_item<1> it) {
8.        int i = it.get_global_linear_id();
9.        i = i * 16;
10.       for (int j = i; j < (i + 16); j++) {
11.         data[j] = data2[j];
12.       }
13.   });
14. });
```

このカーネルは、1024 x 1024 の整数配列を同じサイズの別の整数配列にコピーします。各 work-item は連続する 16 個の整数をコピーします。ただし、`data2` からの読み取りはギャザーされ、`data` へのストアはスキャッターされます。work-item の代わりに各 sub-group で連即する整数を読み取って格納するようにコードを変更すると、より効率が高まります。

```
1.  constexpr int N = 1024 * 1024;
2.  int *data = sycl::malloc_shared<int>(N, q);
3.  int *data2 = sycl::malloc_shared<int>(N, q);
4.  memset(data2, 0xFF, sizeof(int) * N);
5.  auto e = q.submit([&](auto &h) {
6.    h.parallel_for(sycl::nd_range(sycl::range{N / 16}, sycl::range{32}),
7.      [=](sycl::nd_item<1> it) {
8.        int i = it.get_global_linear_id();
9.        auto sg = it.get_sub_group();
10.       int sgSize = sg.get_local_range()[0];
11.       i = (i / sgSize) * sgSize * 16 + (i % sgSize);
12.       for (int j = 0; j < sgSize * 16; j += sgSize) {
13.         data[i + j] = data2[i + j];
14.       }
15.   });
16. });
```

## メモリー帯域幅の利用率を最大化

上記の例の work-item は、ループ反復ごとに 1 つの整数、または 4 バイトをロードおよびストアします (`data[i + j] = data2[i + j];`)。もしくは、ベクトル化されたメモリー操作では、64 バイトをロード/ストアします (グローバルサイズは 16)。メモリー帯域幅は飽和状態ではありません。

各 work-item が一度のメモリー操作でロードまたはストアするデータのペイロードやサイズを増やすと、帯域幅の利用率が高まります。

```
1.  constexpr int N = 1024 * 1024;
2.  int *data = sycl::malloc_shared<int>(N, q);
3.  int *data2 = sycl::malloc_shared<int>(N, q);
4.  memset(data2, 0xFF, sizeof(int) * N);
5.  auto e = q.submit([&](auto &h) {
6.    h.parallel_for(sycl::nd_range(sycl::range{N / 16}, sycl::range{32}),
7.      [=](sycl::nd_item<1> it) {
8.        int i = it.get_global_linear_id();
9.        auto sg = it.get_sub_group();
10.       int sgSize = sg.get_local_range()[0];
11.       i = (i / sgSize) * sgSize * 16 + (i % sgSize) * 4;
12.       for (int j = 0; j < 4; j++) {
13.         sycl::vec<int, 4> x; sycl::vec<int, 4> *q =
14.            (sycl::vec<int, 4> *)(&(data2[i + j * sgSize * 4])); x = *q;
```

```
15.          sycl::vec<int, 4> *r =
16.             (sycl::vec<int, 4> *)(&(data[i + j * sgSize * 4])); *r = x;
17.        }
18.    });
19. });
```

各 work-item は、すべてのループ反復で整数に代わって `sycl::vec<int, 4>` をロード/ストアします。連続した 256 バイトのメモリーリード/ライト (sub-group サイズ 16 と仮定) は、ベクトル化されることですべてのメモリー操作のペイロードは 4 倍になります。

最大帯域幅はハードウェアによって異なります。インテル® VTune™ プロファイラーを使用して帯域幅を測定し、最適なサイズを特定できます。

ただし、ベクトルタイプを使用するとレジスターの負荷が増加する可能性があります。レジスタースピルが発生しない場合に限り、長いベクトルを使用することを推奨します。レジスタースピルを回避する方法については、「レジスターの使用とスピルの回避」の節を参照してください。

## メモリーのブロックロードとブロックストア

インテル® グラフィックスには、メモリーのブロックロード/ストアに最適化された命令が用意されています。sub-group 内の work-item が連続するメモリーブロックにアクセスする場合、sub-group ブロックのアクセス関数を使用して、ブロック単位でロード/ストア命令を利用できます。

```
1.  constexpr int N = 1024 * 1024;
2.  int *data = sycl::malloc_shared<int>(N, q);
3.  int *data2 = sycl::malloc_shared<int>(N, q);
4.  memset(data2, 0xFF, sizeof(int) * N);
5.  auto e = q.submit([&](auto &h) {
6.    h.parallel_for(
7.      sycl::nd_range(sycl::range{N / 16}, sycl::range{32}), [=](sycl::nd_item<1> it)
8.      [[intel::reqd_sub_group_size(16)]] {
9.        auto sg = it.get_sub_group();
10.       sycl::vec<int, 8> x;
11.       using global_ptr =
12.         sycl::multi_ptr<int, sycl::access::address_space::global_space>;
13.       int base = (it.get_group(0) * 32 +
14.         sg.get_group_id()[0] * sg.get_local_range()[0]) * 16;
15.       x = sg.load<8>(global_ptr(&(data2[base + 0])));
16.       sg.store<8>(global_ptr(&(data[base + 0])), x);
17.       x = sg.load<8>(global_ptr(&(data2[base + 128])));
18.       sg.store<8>(global_ptr(&(data[base + 128])), x);
19.   });
20. });
```

この例では、パフォーマンス向上のため `sycl::vec` を使用していますが、整数 `x` はメモリー内で連続していないため、sub-group サイズのストライドを使用しています。

sub-group サイズが 16 であることに気付きましたか? sub-group 関数を利用する場合、コンパイラーの選択をオーバーライドして sub-group サイズを常に期待通りにすることを推奨します。

## データ共有

sub-group 内の work-item は同一スレッドで実行されるため、データが各 work-item でプライベートであっても work-item 間で共有するほうが効率的です。また、sub-group でデータを共有するほうが、共有ローカルメモリーや SLM を使用して work-group 間でデータを共有するよりも効果的です。sub-group 内の work-item 間でデータを共有するもう一つの方法は、シャッフル関数を使用することです。

```
1.  constexpr size_t BLOCK_SIZE = 16;
2.  sycl::buffer<uint, 2> m(matrix.data(), sycl::range<2>(N, N));
3.  auto e = q.submit([&](auto &h) {
4.    sycl::accessor marr(m, h);
5.    sycl::local_accessor<uint, 2> barr1(
6.      sycl::range<2>(BLOCK_SIZE, BLOCK_SIZE), h);
7.      sycl::local_accessor<uint, 2> barr2(
8.      sycl::range<2>(BLOCK_SIZE, BLOCK_SIZE), h);
9.      h.parallel_for(
10.       sycl::nd_range<2>(sycl::range<2>(N / BLOCK_SIZE, N),
11.       sycl::range<2>(1, BLOCK_SIZE)),
12.       [=](sycl::nd_item<2> it) [[intel::reqd_sub_group_size(16)]] {
13.         int gi = it.get_group(0);
14.         int gj = it.get_group(1);
15.         auto sg = it.get_sub_group();
16.         uint sgId = sg.get_local_id()[0];
17.         uint bcol[BLOCK_SIZE];
18.         int ai = BLOCK_SIZE * gi;
19.         int aj = BLOCK_SIZE * gj;
20.         for (uint k = 0; k < BLOCK_SIZE; k++) {
21.           bcol[k] = sg.load(marr.get_pointer() + (ai + k) * N + aj);
22.         }
23.         uint tcol[BLOCK_SIZE];
24.         for (uint n = 0; n < BLOCK_SIZE; n++) {
25.           if (sgId == n) {
26.             for (uint k = 0; k < BLOCK_SIZE; k++) {
27.               tcol[k] = sg.shuffle(bcol[n], k);
28.             }
29.           }
30.         }
31.         for (uint k = 0; k < BLOCK_SIZE; k++) {
32.           sg.store(marr.get_pointer() + (ai + k) * N + aj, tcol[k]);
33.         }
34.     });
35. });
```

このカーネルは 16 x 16 の行列を転置します。前述の例より複雑に見えますが、考え方は単純です。sub-group が 16 x 16 のサブ行列をロードして、シャッフル関数を使用してサブ行列を転置します。サブ行列は 1 つの行列であるため、必要となる sub-group は 1 つのみです。4096 x 4096 のような大きな行列も同様に転置できます。各 sub-group はサブ行列をロードし、サブ行列は sub-group のシャッフル関数を使用して転置されます。これは演習として皆さんに任せますので、試してみてください。

SYCL* で使用できる sub-group シャッフル関数にはいくつかのバリエーションがあります。それぞれのバリアントは、特定デバイスの特定用途向けに最適化されています。独自の関数を作成せずに、これらの最適化された関数を利用することを推奨します (要件に一致する場合)。

## sub-group サイズと最大 sub-group サイズ

ここまでの例では、work-group のサイズは sub-group のサイズと、work-group のサイズおよび sub-group のサイズの両方で割り切れます (2 の累乗)。work-group のサイズが最大 sub-group サイズで割り切れ、両方のサイズが 2 の累乗である場合、sub-group サイズと最大 sub-group サイズは同じです。しかし、work-group サイズが sub-group サイズで割り切れない場合はどうでしょう? 次の例について考えてみます。

```
1.  auto e = q.submit([&](auto &h) {
2.    sycl::stream out(65536, 128, h);
3.    h.parallel_for(sycl::nd_range<1>(7, 7),
4.      [=](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(16)]] {
5.        int i = it.get_global_linear_id();
6.        auto sg = it.get_sub_group();
7.        int sgSize = sg.get_local_range()[0];
8.        int sgMaxSize = sg.get_max_local_range()[0];
9.        int sId = sg.get_local_id()[0];
```

```
10.       int j = data[i];
11.       int k = data[i + sgSize];
12.       out << "globalId = " << i << " sgMaxSize = " << sgMaxSize
13.           << " sgSize = " << sgSize << " sId = " << sId
14.           << " j = " << j << " k = " << k << sycl::endl;
15.   });
16. });
17. q.wait();
```

この例の出力では次のようになります。

```
globalId = 0 sgMaxSize = 16 sgSize = 7 sId = 0 j = 0 k = 7
globalId = 1 sgMaxSize = 16 sgSize = 7 sId = 1 j = 1 k = 8
globalId = 2 sgMaxSize = 16 sgSize = 7 sId = 2 j = 2 k = 9
globalId = 3 sgMaxSize = 16 sgSize = 7 sId = 3 j = 3 k = 10
globalId = 4 sgMaxSize = 16 sgSize = 7 sId = 4 j = 4 k = 11
globalId = 5 sgMaxSize = 16 sgSize = 7 sId = 5 j = 5 k = 12
globalId = 6 sgMaxSize = 16 sgSize = 7 sId = 6 j = 6 k = 13
```

sub-group サイズは 7 ですが、最大 sub-group サイズは 16 のままです。sub-group の最大サイズは SIMD 幅であるため変更されませんが、sub-group 内の work-item は 8 未満であるため、sub-group のサイズは 7 となります。そのため、work-group サイズが sub-group の最大サイズで割り切れない場合は注意が必要です。work-item が少ない最後の sub-group は、別の処理が必要な場合があります。

## 条件チェックを排除

「sub-group と SIMD ベクトル化」の項で、SIMD の分散がパフォーマンスに影響する可能性を説明しました。sub-group 内の work-item がすべて同じ命令を実行すると、SIMD レーンを最大限に活用できます。1 つ以上の work-item が分散パスで処理される場合、マージする前に両方のパスが実行される必要があります。

すべての条件チェックが分散を引き起こすわけではありませんが、通常、分散は条件チェックによって生じます。一部の条件チェックは、SIMD の分散を引き起こさなくてもパフォーマンス・ハザードとなる可能性があります。一般に、条件チェックを排除するとパフォーマンスの向上につながります。

### バッファーをパディングして条件チェックを排除

共有ローカルメモリーの畳み込みの例を見てみましょう。

```
1.  sycl::buffer<int> ibuf(input.data(), N);
2.  sycl::buffer<int> obuf(output.data(), N);
3.  sycl::buffer<int> kbuf(kernel.data(), M);
4.  auto e = q.submit([&](auto &h) {
5.    sycl::accessor iacc(ibuf, h, sycl::read_only);
6.    sycl::accessor oacc(obuf, h);
7.    sycl::accessor kacc(kbuf, h, sycl::read_only);
8.    h.parallel_for(sycl::nd_range<1>(sycl::range{N}, sycl::range{256}),
9.      [=](sycl::nd_item<1> it) {
10.       int i = it.get_global_linear_id();
11.       int group = it.get_group()[0];
12.       int gSize = it.get_local_range()[0];
13.       int t = 0;
14.       int _M = static_cast<int>(M);
15.       int _N = static_cast<int>(N);
16.       if ((group == 0) || (group == _N / gSize - 1)) {
17.         if (i < _M / 2) {
18.           for (int j = _M / 2 - i, k = 0; j < _M; ++j, ++k) {
19.             t += iacc[k] * kacc[j];
20.           }
21.         } else {
```

```
22.           if (i + _M / 2 >= _N) {
23.             for (int j = 0, k = i - _M / 2;j < _M / 2 + _N - i; ++j, ++k) {
24.               t += iacc[k] * kacc[j];
25.             }
26.           } else {
27.             for (int j = 0, k = i - _M / 2; j < _M; ++j, ++k) {
28.               t += iacc[k] * kacc[j];
29.             }
30.           }
31.         }
32.       } else {
33.         for (int j = 0, k = i - _M / 2; j < _M; ++j, ++k) {
34.           t += iacc[k] * kacc[j];
35.         }
36.       }
37.     oacc[i] = t;
38.   });
39. });
```

入れ子になった `if-then-else` の条件チェックは、入力の最初と最後の 128 要素を処理するために必要です。これにより、インデックスが範囲外となることはありません。入力配列の最初と最後に 0 を埋めると、次のように条件チェックを安全に排除できます。

```
1.  std::vector<int> input(N + M / 2 + M / 2);
2.  std::vector<int> output(N);
3.  std::vector<int> kernel(M);
4.  srand(2009);
5.  for (size_t i = M / 2; i < N + M / 2; ++i) {
6.    input[i] = rand();
7.  }
8.  for (size_t i = 0; i < M / 2; ++i) {
9.    input[i] = 0;
10.   input[i + N + M / 2] = 0;
11. }
12. for (size_t i = 0; i < M; ++i) {
13.   kernel[i] = rand();
14. }
15. sycl::buffer<int> ibuf(input.data(), N + M / 2 + M / 2);
16. sycl::buffer<int> obuf(output.data(), N);
17. sycl::buffer<int> kbuf(kernel.data(), M);
18. auto e = q.submit([&](auto &h) {
19.   sycl::accessor iacc(ibuf, h, sycl::read_only);
20.   sycl::accessor oacc(obuf, h);
21.   sycl::accessor kacc(kbuf, h, sycl::read_only);
22.   h.parallel_for(sycl::nd_range(sycl::range{N}, sycl::range{256}),
23.     [=](sycl::nd_item<1> it) {
24.       int i = it.get_global_linear_id();
25.       int t = 0;
26.       for (size_t j = 0; j < M; ++j) {
27.         t += iacc[i + j] * kacc[j];
28.       }
29.       oacc[i] = t;
30.   });
31. });
32. q.wait();
33. size_t kernel_ns = (e.template get_profiling_info<
34.   sycl::info::event_profiling::command_end>() -
35.   e.template get_profiling_info<sycl::info::event_profiling::command_start>());
36. std::cout << "Kernel Execution Time Average: total = "
37.           << kernel_ns * 1e-6
38.           << " msec" << std::endl;
39. }
```

### 条件チェックをリレーショナル関数に置き換える

条件チェックをリレーショナル関数 (特にビルトイン・リレーショナル関数) に置き換えることで排除することもできます。ビルトイン関数が提供される場合、それらの利用を推奨します。SYCL* は、`select()`、`min()`、`max()` など豊富なビルトイン・リレーショナル関数セットを提供します。多くの場合、条件チェックをこれらの関数に置き換えることでパフォーマンスを向上できます。

畳み込みの例をもう一度考えてみましょう。`if-then-else` 条件チェックは、ビルトイン関数 `min()` と `max()` に置き換えることができます。

```
1.  sycl::buffer<int> ibuf(input.data(), N);
2.  sycl::buffer<int> obuf(output.data(), N);
3.  sycl::buffer<int> kbuf(kernel.data(), M);
4.  auto e = q.submit([&](auto &h) {
5.    sycl::accessor iacc(ibuf, h, sycl::read_only);
6.    sycl::accessor oacc(obuf, h);
7.    sycl::accessor kacc(kbuf, h, sycl::read_only);
8.    h.parallel_for(sycl::nd_range(sycl::range{N}, sycl::range{256}),
9.      [=](sycl::nd_item<1> it) {
10.       int i = it.get_global_linear_id();
11.       int t = 0;
12.       int startj = sycl::max<int>(M / 2 - i, 0);
13.       int endj = sycl::min<int>(M / 2 + N - i, M);
14.       int startk = sycl::max<int>(i - M / 2, 0);
15.       for (int j = startj, k = startk; j < endj; j++, k++) {
16.         t += iacc[k] * kacc[j];
17.       }
18.       oacc[i] = t;
19.   });
20. });
```

## レジスターの使用とスピルの回避

### レジスターとパフォーマンス

レジスターはメモリー階層における最速のストレージです。データを可能な限り長くレジスターに留めることは、パフォーマンスにとって重要です。一方、レジスター数は限られており、メモリー空間よりもはるかに少ないデータしか保持できません。例えば、現世代のインテル® GPU には、XVE スレッドごとに 32 バイト幅の汎用レジスターが 128 個備わっています。コンパイラーはできるだけ多くの変数をレジスターに割り当てようとしますが、実行中の任意の時点では限られた数のレジスターに少数の変数セットをのみを割り当てることができます。すべての変数が同時に必要となるわけではなく、さまざまな変数のセットが処理経過とともに変わるため、レジスターは処理中にいろいろな変数を保持することになります。必要とするすべての変数をレジスターに保持できない場合、レジスターがスピル (溢れる) することがあります。また、レジスターにある変数をメモリーに退避して、別の変数向けのスペースを確保する必要が生じる場合もあります。

SYCL* では、コンパイラーはレジスターに work-item のプライベート変数を割り当てます。sub-group 内のすべての work-item は、単一の XVE スレッドにパックされます。デフォルトでは、コンパイラーは SIMD 幅または sub-group サイズを決定するヒューリスティックの 1 つとしてレジスター・プレッシャーを考慮します。レジスター・プレッシャーが高いと、sub-group サイズが小さくなる可能性があります。例えば、sub-group サイズが明示的に指定されていない場合、レジスタースピルが発生するか特定の変数がレジスターに格納されないとサイズは 16 ではなく 8 になります。

sub-group サイズや SIMD 幅が、ハードウェアの持つ最大値でない場合、ハードウェアが十分に活用されていないと考えられます。レジスターのスピルがホットなループ内で発生すると、パフォーマンスが大幅に低下する可能性があります。変数をレジスターに格納していない場合、その変数へのアクセスによりメモリー・トラフィックが大幅に増加することがあります。

コンパイラーはレジスターのスピルを回避もしくは最小化しようとしますが、開発者が最適化することで、コンパイラーの適切な判断に役立ち、多くの場合パフォーマンスに良い結果をもたらします。

## 最適化の手法

レジスター・プレッシャーを減らすにはいくつかの方法があります。

- 値を変数にロードしてからその変数を使用するまでの距離をなるべく短くします。
- コンパイラーは命令をスケジュールして距離を最適化しますが、同じ変数へのロードと使用を近接させたり、ソース内の依存関係を排除することで、コンパイラーの最適化に有効な作用をもたらすことがあります。
- 過度なループアンロールを回避します。

  ループアンロールは、コンパイラーによる命令スケジュールの最適化の可能性を高め、パフォーマンスの向上につながります。しかし、アンロールによって使用される一時変数がレジスター割り当てのプレッシャーを高め、レジスタースピルを引き起こす可能性があります。ループアンロールの有無、およびアンロールの回数を比較して、ループをアンロールするかまたその回数を決定してください。
- USM ポインターを推奨します。

  バッファーアクセサーは、USM ポインターよりも多くのスペースを使用します。USM ポインターとバッファーアクセサーを選択可能な場合は、USM ポインターを選択します。
- 低いコストで求められる値はその都度計算します。そうしないと、レジスターに長期間保持されることになります。
- 大きな配列や構造体の利用は避けます。大きな構造体は小さな複数の構造体に分割します。

  例えば、

  ```
  sycl::float4:
  sycl::float4 v[8];
  ```

  が 4 つの float 配列、

  `float x[8]`、`float y[8]`、`float z[8]`、`float w[8]` に分割可能であれば、`float` の 4 つの配列のすべて、または一部が `sycl::float4` の配列よりもレジスターに割り当てられる可能性が高くなります。
- 大きなループを複数の小さなループに分解して、同時に利用される変数の数を減らすことができます。
- 可能な限り小さなデータタイプを使用します。
- プライベート変数を `volatile` として宣言してはなりません。
- sub-group でレジスターを共有します。
- 可能であれば、sub-group ブロックのロード/ストアを使用します。
- 共有ローカルメモリーを使用します。

上記はすべてを網羅しているわけではありません。

この章の残りでは、上記の手法の特に最後の 5 つを適用する例を示します。

## 小さなデータタイプを選択

```
1.  constexpr int BLOCK_SIZE = 256;
2.  constexpr int NUM_BINS = 32;
```

```
3.  std::vector<unsigned long> hist(NUM_BINS, 0);
4.  sycl::buffer<unsigned long, 1> mbuf(input.data(), N);
5.  sycl::buffer<unsigned long, 1> hbuf(hist.data(), NUM_BINS);
6.  auto e = q.submit([&](auto &h) {
7.    sycl::accessor macc(mbuf, h, sycl::read_only);
8.    auto hacc = hbuf.get_access<sycl::access::mode::atomic>(h);
9.      h.parallel_for(
10.       sycl::nd_range(sycl::range{N / BLOCK_SIZE}, sycl::range{64}),
11.       [=](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(16)]] {
12.         int group = it.get_group()[0];
13.         int gSize = it.get_local_range()[0];
14.         auto sg = it.get_sub_group();
15.         int sgSize = sg.get_local_range()[0];
16.         int sgGroup = sg.get_group_id()[0];
17.         unsigned long histogram[NUM_BINS]; // 分布図の値域は、レジスターに格納するには
18.                                            // ストレージを消費しすぎます
19.         for (int k = 0; k < NUM_BINS; k++) {
20.           histogram[k] = 0;
21.         }
22.         for (int k = 0; k < BLOCK_SIZE; k++) {
23.           unsigned long x =
24.             sg.load(macc.get_pointer() + group * gSize * BLOCK_SIZE +
25.             sgGroup * sgSize * BLOCK_SIZE + sgSize * k);
26. #pragma unroll
27.           for (int i = 0; i < 8; i++) {
28.             unsigned int c = x & 0x1FU;
29.             histogram[c] += 1;
30.             x = x >> 8;
31.           }
32.         }
33.         for (int k = 0; k < NUM_BINS; k++) {
34.           hacc[k].fetch_add(histogram[k]);
35.         }
36.     });
37. });
```

この例では 32 値域の分布図を計算します。各 work-item には、`unsigned long` データタイプの 32 個のプライベート値域があります。大きなストレージが必要であるため、プライベート値域をレジスターに格納できずパフォーマンスが低下します。

`BLOCK_SIZE` が 256 である場合、各プライベート分布図の値域の最大値は、`unsigned int` を超えることはありません。プライベート分布図の値域を `unsigned long` から `unsigned int` にしてレジスター・プレッシャーを減らし、プライベート値域をレジスターに収めることができます。これは簡単な変更ですがパフォーマンスに大きな影響を与えます。

```
1.  constexpr int BLOCK_SIZE = 256;
2.  constexpr int NUM_BINS = 32;
3.  std::vector<unsigned long> hist(NUM_BINS, 0);
4.  sycl::buffer<unsigned long, 1> mbuf(input.data(), N);
5.  sycl::buffer<unsigned long, 1> hbuf(hist.data(), NUM_BINS);
6.  auto e = q.submit([&](auto &h) {
7.    sycl::accessor macc(mbuf, h, sycl::read_only);
8.    auto hacc = hbuf.get_access<sycl::access::mode::atomic>(h);
9.    h.parallel_for(
10.     sycl::nd_range(sycl::range{N / BLOCK_SIZE}, sycl::range{64}),
11.     [=](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(16)]] {
12.       int group = it.get_group()[0];
13.       int gSize = it.get_local_range()[0];
14.       auto sg = it.get_sub_group();
15.       int sgSize = sg.get_local_range()[0];
16.       int sgGroup = sg.get_group_id()[0];
17.       unsigned int histogram[NUM_BINS]; //分布図の値域はデータタイプが小さいほど
```

```
                                                //ストレージの使用量が少なくなります
      for (int k = 0; k < NUM_BINS; k++) {
        histogram[k] = 0;
      }
      for (int k = 0; k < BLOCK_SIZE; k++) {
        unsigned long x =
          sg.load(macc.get_pointer() + group * gSize * BLOCK_SIZE +
          sgGroup * sgSize * BLOCK_SIZE + sgSize * k);
#pragma unroll
        for (int i = 0; i < 8; i++) {
          unsigned int c = x & 0x1FU;
          histogram[c] += 1;
          x = x >> 8;
        }
      }
      for (int k = 0; k < NUM_BINS; k++) {
        hacc[k].fetch_add(histogram[k]);
      }
  });
});
```

## プライベート変数を volatile として宣言しない

前述のコード例を少し変更します。

```
std::vector<unsigned long> hist(NUM_BINS, 0);
sycl::buffer<unsigned long, 1> mbuf(input.data(), N);
sycl::buffer<unsigned long, 1> hbuf(hist.data(), NUM_BINS);
auto e = q.submit([&](auto &h) {
  sycl::accessor macc(mbuf, h, sycl::read_only);
  auto hacc = hbuf.get_access<sycl::access::mode::atomic>(h);
  h.parallel_for(sycl::nd_range(sycl::range{N / BLOCK_SIZE}, sycl::range{64}),
    [=](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(16)]] {
      int group = it.get_group()[0];
      int gSize = it.get_local_range()[0];
      auto sg = it.get_sub_group();
      int sgSize = sg.get_local_range()[0];
      int sgGroup = sg.get_group_id()[0];
      volatile unsigned int histogram[NUM_BINS]; // 不揮発性変数はレジスターに
                                                 // 割り当てられません
      for (int k = 0; k < NUM_BINS; k++) {
        histogram[k] = 0;
      }
      for (int k = 0; k < BLOCK_SIZE; k++) {
        unsigned long x = sg.load(
          macc.get_pointer() + group * gSize * BLOCK_SIZE +
          sgGroup * sgSize * BLOCK_SIZE + sgSize * k);
#pragma unroll
        for (int i = 0; i < 8; i++) {
          unsigned int c = x & 0x1FU;
          histogram[c] += 1;
          x = x >> 8;
        }
      }
      for (int k = 0; k < NUM_BINS; k++) {
        hacc[k].fetch_add(histogram[k]);
      }
  });
});
```

プライベート `histogram` 配列は、`volatile` 配列として修飾されています。`volatile` 変数は揮発性であり、2 つの異なるロード操作間で値が変化する可能性があるため、レジスターには保持されません。

実際には、`histogram` のプライベート配列が `volatile` である理由はありません。プライベート変数が揮発性である必要があるのであれば、それはもはやプライベートではありません。

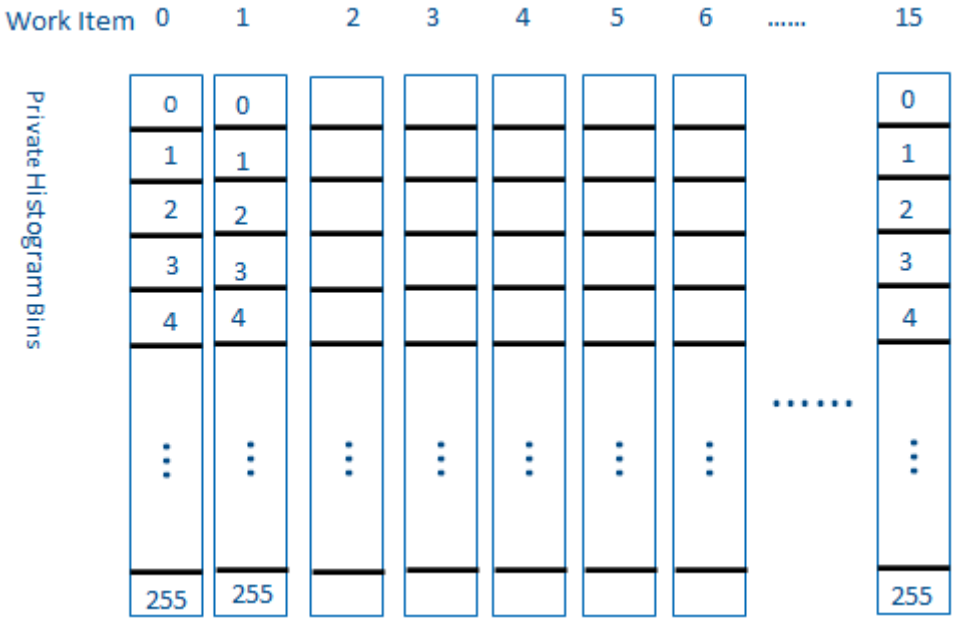
## sub-group でレジスターを共有

次に分布図の値域を 256 に増やします。

```
1.  constexpr int BLOCK_SIZE = 256;
2.  constexpr int NUM_BINS = 256;
3.  std::vector<unsigned long> hist(NUM_BINS, 0);
4.  sycl::buffer<unsigned long, 1> mbuf(input.data(), N);
5.  sycl::buffer<unsigned long, 1> hbuf(hist.data(), NUM_BINS);
6.  auto e = q.submit([&](auto &h) {
7.    sycl::accessor macc(mbuf, h, sycl::read_only);
8.    auto hacc = hbuf.get_access<sycl::access::mode::atomic>(h);
9.    h.parallel_for(
10.     sycl::nd_range(sycl::range{N / BLOCK_SIZE}, sycl::range{64}),
11.     [=](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(16)]] {
12.       int group = it.get_group()[0];
13.       int gSize = it.get_local_range()[0];
14.       auto sg = it.get_sub_group();
15.       int sgSize = sg.get_local_range()[0];
16.       int sgGroup = sg.get_group_id()[0];
17.       unsigned int histogram[NUM_BINS]; // 分布図の値域は、レジスターに格納するには
18.                                         // ストレージを消費しすぎます
19.       for (int k = 0; k < NUM_BINS; k++) {
20.         histogram[k] = 0;
21.       }
22.       for (int k = 0; k < BLOCK_SIZE; k++) {
23.         unsigned long x =
24.           sg.load(macc.get_pointer() + group * gSize * BLOCK_SIZE +
25.           sgGroup * sgSize * BLOCK_SIZE + sgSize * k);
26. #pragma unroll
27.         for (int i = 0; i < 8; i++) {
28.           unsigned int c = x & 0x1FU;
29.           histogram[c] += 1;
30.           x = x >> 8;
31.         }
32.       }
33.       for (int k = 0; k < NUM_BINS; k++) {
34.         hacc[k].fetch_add(histogram[k]);
35.       }
36.   });
37. });
```

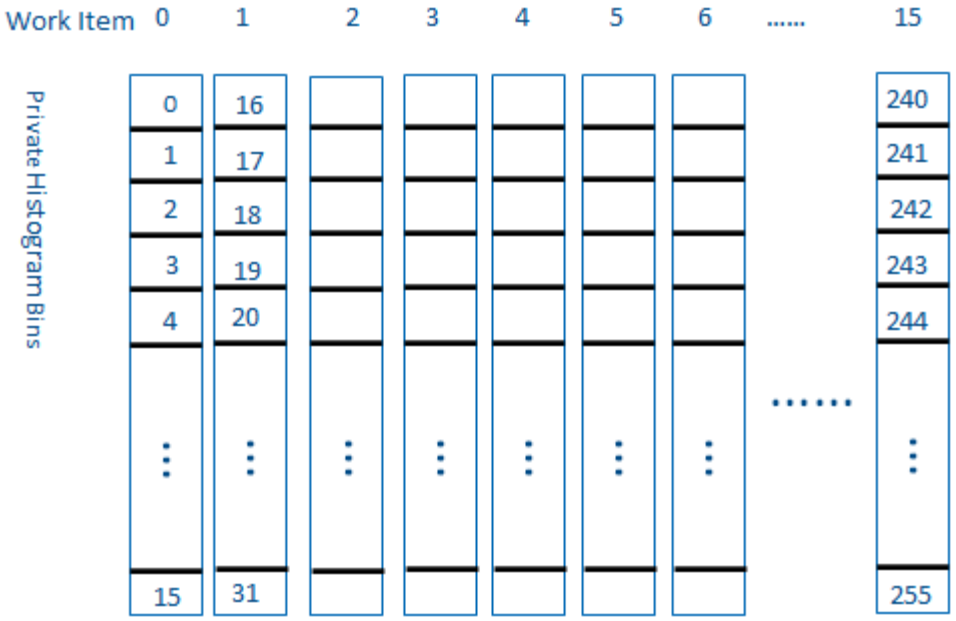
256 個の分布図値域では、データタイプが `unsigned int` でもパフォーマンスは低下します。各 work-item のプライベート値域のストレージは、レジスターに格納するには大きすぎます。

### 各 work-item が 256 個のプライベート分布図値域を持つ




sub-group サイズが 16 である場合、16 個の work-item が 1 つのベクトルエンジン (VE) のスレッドにパックされていることが分かります。また、同じ sub-group の work-item は相互に効率良く通信およびデータ共有できることが判明しています。同じ sub-group 内の work-item がプライベート分布図の値域を供する場合、sub-group 全体で 256 個のプライベート値域が、または work-item には 16 個のプライベート値域が必要になります。

### sub-group が 256 個のプライベート分布図値域を持つ



sub-group の分布図値域を共有するには、各 work-item は入力データを同じ sub-group のすべての work-item にブロードキャストします。対応する分布図値域を所有する work-item が更新を行います。

```
1.  constexpr int BLOCK_SIZE = 256;
2.  constexpr int NUM_BINS = 256;
3.  std::vector<unsigned long> hist(NUM_BINS, 0);
4.  sycl::buffer<unsigned long, 1> mbuf(input.data(), N);
5.  sycl::buffer<unsigned long, 1> hbuf(hist.data(), NUM_BINS);
6.  auto e = q.submit([&](auto &h) {
7.    sycl::accessor macc(mbuf, h, sycl::read_only);
8.    auto hacc = hbuf.get_access<sycl::access::mode::atomic>(h);
9.    h.parallel_for(
10.     sycl::nd_range(sycl::range{N / BLOCK_SIZE}, sycl::range{64}),
11.     [=](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(16)]] {
12.       int group = it.get_group()[0];
13.       int gSize = it.get_local_range()[0];
14.       auto sg = it.get_sub_group();
15.       int sgSize = sg.get_local_range()[0];
16.       int sgGroup = sg.get_group_id()[0];
17.       unsigned int histogram[NUM_BINS / 16]; // 分布図はレジスターに格納するには
18.                                              // ストレージを消費しすぎます
19.       for (int k = 0; k < NUM_BINS / 16; k++) {
20.         histogram[k] = 0;
21.       }
22.       for (int k = 0; k < BLOCK_SIZE; k++) {
23.         unsigned long x =
24.           sg.load(macc.get_pointer() + group * gSize * BLOCK_SIZE +
25.           sgGroup * sgSize * BLOCK_SIZE + sgSize * k);
26. // subgroup サイズは 16
27. #pragma unroll
28.         for (int j = 0; j < 16; j++) {
29.           unsigned long y = sycl::group_broadcast(sg, x, j);
30. #pragma unroll
31.           for (int i = 0; i < 8; i++) {
32.             unsigned int c = y & 0xFF;
33.                                      // (c & 0xF) は値域が存在する work-item です。
34.                                      // (c >> 4) は値域のインデックスです。
35.             if (sg.get_local_id()[0] == (c & 0xF)) {
36.               histogram[c >> 4] += 1;
37.             }
38.             y = y >> 8;
39.           }
40.         }
41.       }
42.       for (int k = 0; k < NUM_BINS / 16; k++) {
43.         hacc[16 * k + sg.get_local_id()[0]].fetch_add(histogram[k]);
44.       }
45.   });
46. });
```

## sub-goup ブロックのロード/ストアを使用する

メモリーブロックのロード/ストアはベクトル化されます。ベクトルロード/ストア命令の各レーンには、固有のアドレスとデータがあります。アドレスとデータの両方がレジスター空間を持ちます。次に例を示します。

```
1.  constexpr int N = 1024 * 1024;
2.  int *data = sycl::malloc_shared<int>(N, q);
3.  int *data2 = sycl::malloc_shared<int>(N, q);
4.  memset(data2, 0xFF, sizeof(int) * N);
5.  auto e = q.submit([&](auto &h) {
6.    h.parallel_for(sycl::nd_range(sycl::range{N}, sycl::range{32}),
7.      [=](sycl::nd_item<1> it) {
8.        int i = it.get_global_linear_id();
9.        data[i] = data2[i];
10.   });
11. });
```

メモリーは、`data[i] = data2[i];` 文でロードおよびストアされ、各ベクトルレーンは独自のアドレスを持ちます。SIMD 幅または sub-group サイズが 16 であると仮定すると、16 レーンのアドレスの合計レジスター空間は、128 バイトになります。各 GRF レジスターが 32 バイト幅である場合、アドレスには 4 つの GRF レジスターが必要です。

アドレスが連続しているため。sub-group ブロックのロード/ストアのビルトインを使用することで、アドレス・レジスター空間を節約できます。

```
1.  constexpr int N = 1024 * 1024;
2.  int *data = sycl::malloc_shared<int>(N, q);
3.  int *data2 = sycl::malloc_shared<int>(N, q);
4.  memset(data2, 0xFF, sizeof(int) * N);
5.  auto e = q.submit([&](auto &h) {
6.    h.parallel_for(
7.      sycl::nd_range(sycl::range{N}, sycl::range{32}),
8.      [=](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(16)]] {
9.        auto sg = it.get_sub_group();
10.       int x;
11.       using global_ptr =
12.         sycl::multi_ptr<int, sycl::access::address_space::global_space>;
13.       int base = (it.get_group(0) * 32 +
14.         sg.get_group_id()[0] * sg.get_local_range()[0]);
15.       x = sg.load(global_ptr(&(data2[base + 0])));
16.       sg.store(global_ptr(&(data[base + 0])), x);
17.   });
18. });
19. x = sg.load(global_ptr(&(data2[base + 0])));
20. sg.store(global_ptr(&(data[base + 0])), x);
```

上記の文は、連続したメモリーブロックをロード/ストアしますが、コンパイラーはこの 2 つの文を特別なメモリーブロックのロード/ストア命令にコンパイルします。連続したメモリーブロックであるため、必要となるのはブロックの開始アドレスのみです。これにより、レジスター空間の 128 バイトではなく 8 バイト、または最大で 1 つのレジスターが各ブロックのロード/ストアのアドレスとして使用されます。

## 共有ローカルメモリーを使用

例えば、分布図値域を 1024 に増やすと、プライベート値域が同じ sub-group で共有されていても、プライベート値域向けのレジスターが足りなくなります。メモリー・トラフィックを軽減するには、ローカル分布図値域を共有ローカルメモリーに割り当てて、同じ work-group 内の work-item で共有します。「共有ローカルメモリー」の項で分布図の例を確認してください。

# レジスタープレッシャーの高いコードのインテル® データセンター GPU マックス・シリーズへの移植

コードを CUDA* から SYCL* に移行すると、NVIDIA* GPU では良好なパフォーマンスを示していても、移行後の SYCL* コードがインテル® データセンター GPU (インテル® データセンター GPU マックス 1550 など) では、期待するパフォーマンスを示さないことがあります。その原因の 1 つとして、汎用レジスターファイル (GRF) のサイズが異なるため、NVIDIA* GPU で発生しないレジスタースピルがインテル® GPU で発生することがあげられます。コードを変更することなくレジスタースピルを回避する方法については以下で説明します。この節では説明しませんが、レジスター全体のプレッシャーを軽減する最適化テクニックについては、「レジスターの使用とスピルの回避」を参照してください。

## レジスター・プレッシャーの高いシンプルなコード

ここで示す例は、多変量双対数を使用して自動微分を実行するコードからヒントを得たものです。この例を理解するため二重数に関する知識は必要ありません。二重数の詳細については Wikipedia のページおよびこちらの書籍 (英語) を参照してください。

```
#include <chrono>
#include <iostream>
#include <sycl/sycl.hpp>

#define NREGISTERS 80

#ifndef SIMD16
#define SIMD_SIZE 32
#else
#define SIMD_SIZE 16
#endif

class my_type {
public:
  my_type(double x0, double x1, double x2, double x3, double x4, double x5,
          double x6, double x7)
      : x0_(x0), x1_(x1), x2_(x2), x3_(x3), x4_(x4), x5_(x5), x6_(x6), x7_(x7) {
  }

  my_type(double const *const vals)
      : x0_(vals[0]), x1_(vals[1]), x2_(vals[2]), x3_(vals[3]), x4_(vals[4]),
        x5_(vals[5]), x6_(vals[6]), x7_(vals[7]) {}

  my_type operator+(const my_type &rhs) {
    return my_type(rhs.x0_ + x0_, rhs.x1_ + x1_, rhs.x2_ + x2_, rhs.x3_ + x3_,
                   rhs.x4_ + x4_, rhs.x5_ + x5_, rhs.x6_ + x6_, rhs.x7_ + x7_);
  }

  void WriteBack(double *const vals) {
    vals[0] += x0_;
    vals[1] += x1_;
    vals[2] += x2_;
    vals[3] += x3_;
    vals[4] += x4_;
    vals[5] += x5_;
    vals[6] += x6_;
    vals[7] += x7_;
  }

private:
  double x0_, x1_, x2_, x3_, x4_, x5_, x6_, x7_;
};

int main(int argc, char **argv) {
```

```
sycl::queue Q(sycl::gpu_selector_v);
size_t nsubgroups;
if (argc > 1)
  nsubgroups = 10;
else
  nsubgroups = 8;

const size_t ARR_SIZE = nsubgroups * SIMD_SIZE * NREGISTERS;
double *val_in = sycl::malloc_shared<double>(ARR_SIZE, Q);
double *val_out = sycl::malloc_shared<double>(ARR_SIZE, Q);

std::cout << "Using simd size " << SIMD_SIZE << std::endl;

Q.parallel_for(ARR_SIZE, [=](auto it) {
   val_in[it.get_id()] = argc * it.get_id();
 }).wait();

auto start_time = std::chrono::high_resolution_clock::now();

for (int rep = 0; rep < 1000; rep++) {
  Q.parallel_for(sycl::nd_range<1>(nsubgroups * SIMD_SIZE, SIMD_SIZE),
                 [=](auto it) [[intel::reqd_sub_group_size(SIMD_SIZE)]] {
                   const size_t id = it.get_global_linear_id();

                   double const *const val_offs = val_in + id * NREGISTERS;

                   my_type v0(val_offs + 0);
                   my_type v1(val_offs + 8);
                   my_type v2(val_offs + 16);
                   my_type v3(val_offs + 24);
                   my_type v4(val_offs + 32);
                   my_type v5(val_offs + 40);
                   my_type v6(val_offs + 48);
                   my_type v7(val_offs + 56);
                   my_type v8(val_offs + 64);
                   my_type v9(val_offs + 72);

                   my_type p0 = v0 + v5;
                   my_type p1 = v1 + v6;
                   my_type p2 = v2 + v7;
                   my_type p3 = v3 + v8;
                   my_type p4 = v4 + v9;

                   p0 = p0 + p1 + v5;
                   p2 = p2 + p3 + v6;
                   p4 = p4 + p1 + v7;
                   p3 = p3 + p2 + v8;
                   p2 = p0 + p4 + v9;
                   p1 = p1 + v0;

                   double *const vals_out_offs = val_out + id * NREGISTERS;
                   p0.WriteBack(vals_out_offs);
                   p1.WriteBack(vals_out_offs + 8);
                   p2.WriteBack(vals_out_offs + 16);
                   p3.WriteBack(vals_out_offs + 24);
                   p4.WriteBack(vals_out_offs + 32);
                   v0.WriteBack(vals_out_offs + 40);
                   v1.WriteBack(vals_out_offs + 48);
                   v2.WriteBack(vals_out_offs + 56);
                   v3.WriteBack(vals_out_offs + 64);
                   v4.WriteBack(vals_out_offs + 72);
                 })
      .wait();
}
auto end_time = std::chrono::high_resolution_clock::now();

std::cout << "Took "
          << std::chrono::duration_cast<std::chrono::microseconds>(end_time -
                                                                   start_time)
```

```
                    .count()
            << " microseconds" << std::endl;

  sycl::free(val_in, Q);
  sycl::free(val_out, Q);

  return 0;
}
```

このコードはクラス `my_type` をリストしたものです。このコードは、7 次元の双対数値と 8 つの倍精度浮動小数点のラップ、および算術オペレーターのオーバーロードを含んでいます。プログラムのメイン関数は、一部のデータを初期化し、特定の GPU カーネルを 1000 回起動します (70 行目の for ループ、72 行目のカーネル起動)。カーネルを繰り返し起動することは、より正確なパフォーマンス数値を取得することを目的としており、レジスタースピルには影響しません。カーネルは、データを使用していくつかの `my_type` の二重数値を初期化し、高いレジスター・プレッシャーをともなうコードで算術オペレーターを実行します。

oneAPI for NVIDIA* GPU プラグインを使用して、次のコマンドで NVIDIA* A100 GPU 向けの SYCL* コードをコンパイルします。

```
$ clang++ -O2 -Xcuda-ptxas -v -fsycl -fsycl-targets=nvidia_gpu_sm_80 --cuda-path=/opt/
hpc_software/compilers/nvidia/cuda-12.0 main.cpp -o exec_cuda.out
```

168 個のレジスターが割り当てられており、次の出力でスピルがないことを確認できます。

```
ptxas info : Used 168 registers, 380 bytes cmem[0]
```

また、次のコマンドで oneAPI 2023.2.0 を使用して事前 (AOT) コンパイルによる PVC コードを生成します。

```
$ icpx -fsycl -fsycl-targets=spir64_gen -Xsycl-target-backend "-device pvc" main.cpp -o
exec_jit.out
```

次のコンパイラーの警告から、レジスタースピルが発生することが分かります。

```
warning: kernel _ZTSZ4mainEUlT_E0_ compiled SIMD32 allocated 128 regs and spilled around
396
```

次の表に、NVIDIA* A100 80GB GPU とインテル® データセンター GPU マックス 1550 (単一スタック) でのコードの実行時間を示します。

**単一スタックのインテル® データセンター GPU マックス 1550 と NVIDIA* A100 80GB での実行時間**

| ハードウェア | 時間 (ms) 低い値が良い | レジスタースピル |
|---|---|---|
| A100 80GB | 21.5 | なし |
| 1T マックス 1550 | 36.1 | あり |

インテル® GPU での実行時間は、NVIDIA* GPU の実行時間を大幅に上回っていることが分かります。レジスタースピルの有無から、この実行時間の違いにはレジスター割り当てが関連していると考えることができます。根本的な原因を理解するため、次のセクションではレジスターファイルの違いを説明します。

## インテル® データセンター GPU マックス 1550 のレジスター

レジスター割り当ての違いは、汎用レジスターファイル (GRF) のサイズの違いによるものです。NVIDIA* A100 GPU は、work-item ごとに最大 256 個の 32 ビット・レジスターを提供しますが、インテル® データセンター GPU マックス 1550 は、デフォルトで 64 個の 32 ビット・レジスターを提供します (sub-group ごとに同数の 32 work-item を想定。インテル® Xe GPU アーキテクチャーを参照)。これは、NVIDIA* GPU の 4 分の 1 です。利用可能なレジスター数を調整するには 2 つの方法があります。それは、ラージ・レジスター・モード (スモール・レジスター・モードとラージ・レジスター・モードを参照) と sub-group サイズの縮小 (sub-group と SIMD ベクトル化を参照) です。上記の例に対するこのオプションの影響については、次のセクションで説明します。

### インテル® データセンター GPU マックス 1550 でのラージ GRF ありなしと SIMD 幅の違い

| ハードウェア | SIMD 幅 | GRF モード | work-item ごとのレジスター |
|---|---|---|---|
| A100 80GB | 32 | なし | 256 x 32 ビット |
| 1T マックス 1550 | 32 | デフォルト | 64 x 32 ビット |
| 1T マックス 1550 | 32 | ラージ GRF | 128 x 32 ビット |
| 1T マックス 1550 | 16 | デフォルト | 128 x 32 ビット |
| 1T マックス 1550 | 16 | ラージ GRF | 256 x 32 ビット |

## work-item ごとのレジスター数を増加

前述の例では 168 個のレジスターが必要です。これまでの説明から、ラージ GRF モードと 16 個の縮小された SIMD 幅を組み合わせることで、レジスタースピルの問題は回避できるはずです。以下では、GRF モードと SIMD 幅の組み合わせを確認します。実際にすべてのレジスタースピルを回避するには、ラージ GRF モードと SIMD 16 が必要であることが分かります。

次のコマンドでコードをラージ GRF モードでコンパイルします。

```
$ icpx -fsycl -fsycl-targets=spir64_gen -Xsycl-target-backend "-device pvc -options -ze-opt-largeregister-file" main.cpp -o exec_largegrf.out
```

以前より減少しましたが、レジスタースピルに関する警告はまだ表示されます。

```
warning: kernel _ZTSZ4mainEUlT_E0_ compiled SIMD32 allocated 256 regs and spilled around 206
```

168 個のレジスターが必要であるのに 128 個の 32 ビット・レジスターしか利用できないことが想定されます。

レジスタースピルを減らすため 2 番目のオプションを検討します。これは、コードに最小限の変更を加えて SIMD 幅を 32 から 16 に減らすことです。これには、次のコマンドを使用します。

```
$ icpx -DSIMD16 -fsycl -fsycl-targets=spir64_gen -Xsycl-target-backend "-device pvc" main.cpp -o exec_simd16.out
```

SIMD 幅 16 でデフォルトの GRF モードを使用すると、レジスタースピルに関する警告が再び表示されます。

```
warning: kernel _ZTSZ4mainEUlT_E0_ compiled SIMD16 allocated 128 regs and spilled around 112
```

SIMD 幅 16 への縮小とラージ GRF モードの両方を使用すると、すべてのレジスタースピルが排除され、コンパイラーは警告を発生しなくなります。これは、最初に期待したものです。

次の表に 4 つの構成の実行時間を示します。

**単一スタックのインテル® データセンター GPU マックス 1550 の異なる設定での実行時間**

| ハードウェア | 時間 (ms) 低い値が良い | レジスタースピル |
|---|---|---|
| A100 80GB | 21.5 | なし |
| 1T マックス 1550 | 36.1 | あり |

明らかに、レジスタースピルが発生しない場合にパフォーマンスが向上することが分かります。興味深いことは、SIMD 幅 16 がラージ GRF モードよりも好ましいということです。ラージ GRF モードでは、ハードウェア・スレッドの半分が無効になることを考えると、それほど驚くべきことではありません。

## SIMD 16 へポーティングする際の考慮事項

上記の方法が適さない、または特別な注意が必要なことがあります。一方、ラージ GRF モードを使用すると、ハードウェア・スレッドの半分が無効になり、特定のコードのパフォーマンスが低下する可能性があります。

SIMD 32 から SIMD 16 への移行により、誤った結果が生じる可能性があるため、注意を要する場合があります。次にコード例を示します。

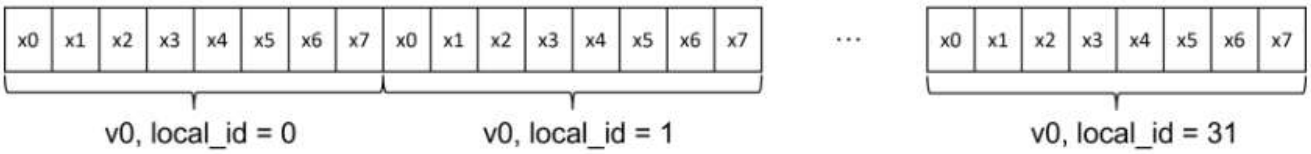
```
#include <chrono>
#include <iostream>
#include <sycl/sycl.hpp>

#define NREGISTERS 80

#ifndef SIMD16
#define SIMD_SIZE 32
#else
#define SIMD_SIZE 16
#endif

class my_type {
public:
  my_type(double x0, double x1, double x2, double x3, double x4, double x5,
          double x6, double x7)
      : x0_(x0), x1_(x1), x2_(x2), x3_(x3), x4_(x4), x5_(x5), x6_(x6), x7_(x7) {
  }

  my_type(double const *const vals)
      : x0_(vals[0]), x1_(vals[1]), x2_(vals[2]), x3_(vals[3]), x4_(vals[4]),
        x5_(vals[5]), x6_(vals[6]), x7_(vals[7]) {}

  my_type operator+(const my_type &rhs) {
    return my_type(rhs.x0_ + x0_, rhs.x1_ + x1_, rhs.x2_ + x2_, rhs.x3_ + x3_,
                   rhs.x4_ + x4_, rhs.x5_ + x5_, rhs.x6_ + x6_, rhs.x7_ + x7_);
  }

  void WriteBack(double *const vals) {
    vals[0] += x0_;
    vals[1] += x1_;
    vals[2] += x2_;
    vals[3] += x3_;
    vals[4] += x4_;
    vals[5] += x5_;
    vals[6] += x6_;
    vals[7] += x7_;
  }

  void Load(double const *const vals) {
    x0_ = vals[0];
    x1_ = vals[1];
```

```
    x2_ = vals[2];
    x3_ = vals[3];
    x4_ = vals[4];
    x5_ = vals[5];
    x6_ = vals[6];
    x7_ = vals[7];
  }

private:
  double x0_, x1_, x2_, x3_, x4_, x5_, x6_, x7_;
};

int main(int argc, char **argv) {
  sycl::queue Q(sycl::gpu_selector_v);
  size_t nsubgroups;
  if (argc > 1)
    nsubgroups = 10;
  else
    nsubgroups = 8;

  const size_t ARR_SIZE = nsubgroups * SIMD_SIZE * NREGISTERS;
  double *val_in = sycl::malloc_shared<double>(ARR_SIZE, Q);
  double *val_out = sycl::malloc_shared<double>(ARR_SIZE, Q);

  std::cout << "Using simd size " << SIMD_SIZE << std::endl;

  Q.parallel_for(ARR_SIZE, [=](auto it) {
     val_in[it.get_id()] = argc * it.get_id();
   }).wait();

  auto start_time = std::chrono::high_resolution_clock::now();

  for (int rep = 0; rep < 1000; rep++) {
    Q.submit([&](auto &h) {
       sycl::accessor<double, 1, sycl::access::mode::read_write,
                      sycl::access::target::local>
           slm(sycl::range(32 * 8), h);

       h.parallel_for(sycl::nd_range<1>(nsubgroups * SIMD_SIZE, 32),
                      [=](auto it) [[intel::reqd_sub_group_size(SIMD_SIZE)]] {
                        const int id = it.get_global_linear_id();
                        const int local_id = it.get_local_id(0);
                        for (int i = 0; i < 8; i++) {
                          slm[i + local_id * 8] = 0.0;
                        }
#ifdef SIMD16
                        it.barrier(sycl::access::fence_space::local_space);
#endif
                        double const *const val_offs = val_in + id * NREGISTERS;
                        my_type v0(val_offs + 0);
                        my_type v1(val_offs + 8);
                        my_type v2(val_offs + 16);
                        my_type v3(val_offs + 24);
                        my_type v4(val_offs + 32);
                        my_type v5(val_offs + 40);
                        my_type v6(val_offs + 48);
                        my_type v7(val_offs + 56);
                        my_type v8(val_offs + 64);
                        my_type v9(val_offs + 72);

                        my_type p0 = v0 + v5;
                        v0.WriteBack(&slm[local_id * 8]);
#ifdef SIMD16
                        it.barrier(sycl::access::fence_space::local_space);
#endif
                        v0.Load(&slm[32 * 8 - 8 - local_id * 8]);
                        my_type p1 = v1 + v6;
                        my_type p2 = v2 + v7;
                        my_type p3 = v3 + v8;
```

```
                        my_type p4 = v4 + v9;

                        p0 = p0 + p1 + v5;
                        p2 = p2 + p3 + v6;
                        p4 = p4 + p1 + v7;
                        p3 = p3 + p2 + v8;
                        p2 = p0 + p4 + v9;

                        p1 = p1 + v0;

                        double *const vals_out_offs = val_out + id * NREGISTERS;
                        p0.WriteBack(vals_out_offs);
                        p1.WriteBack(vals_out_offs + 8);
                        p2.WriteBack(vals_out_offs + 16);
                        p3.WriteBack(vals_out_offs + 24);
                        p4.WriteBack(vals_out_offs + 32);
                        v0.WriteBack(vals_out_offs + 40);
                        v1.WriteBack(vals_out_offs + 48);
                        v2.WriteBack(vals_out_offs + 56);
                        v3.WriteBack(vals_out_offs + 64);
                        v4.WriteBack(vals_out_offs + 72);
                      });
    }).wait();
  }
  auto end_time = std::chrono::high_resolution_clock::now();

  std::cout << "Took "
            << std::chrono::duration_cast<std::chrono::microseconds>(end_time -
                                                                     start_time)
                   .count()
            << " microseconds" << std::endl;

  sycl::free(val_in, Q);
  sycl::free(val_out, Q);

  return 0;
}
```

これは、最初の例に似ていますが、各 work-group (SIMD を使用する場合は sub-group) 内の v0 に格納されている値を転置する際に共有ローカルメモリー配列 (SLM – Shared Local Memory) が導入されています。

**次の図は、2 番目のコード例で SLM がどのように使用されるかを示しています。work-group 内の各 work-item の v0 値は SLM に保存されます。v0 は 8 つの double 値 x0 – x7 で構成されています。**




**v0 値を SLM に保存した後、逆の順番で読み取ります。**

| SLM に保存される work-item のローカル id | SLM からロードされる work-item のローカル id |
|---|---|
| 0 | 31 |
| 1 | 30 |
| 2 | 29 |
| ... | ... |
| 29 | 2 |
| 30 | 1 |
| 31 | 0 |

この場合、SIMD 32 から SIMD 16 への切り替えでは、一般に誤った結果が生じます。それは、SIMD 16 の場合、SLM にアクセスする 32 個の work-item 間の同期が保証されないためです。SIMD 32 では、work-item は同じ sub-group にあり、一緒に処理されるため、これは問題になりません (sub-group と SIMD ベクトル化を参照)。つまり、SIMD 16 では、SLM 値が実際に書き込まれる (116 行目) 前に読み取られて使用される (120 行目) 問題が発生する可能性があります。この問題を解決するには、118 行目に示すように SLM バリアを追加して、値が読み取られる前にすべての値が書き込まれることを保証する必要があります。ただし、これらのバリアは計算時間に影響する可能性があることを忘れないでください。次の表に、4 つの構成における計算時間を示します。

**各種設定による単一スタックのインテル® データセンター GPU マックス 1550 での SLM サンプルの実行時間**

| GRF モード | SIMD 幅 | レジスタースピル | 時間 (ms) 低いほど良い |
|---|---|---|---|
| デフォルト | 32 | あり | 38.0 |
| ラージ GRF | 32 | あり | 33.7 |
| デフォルト | 16 | あり | 31.3 |
| ラージ GRF | 16 | なし | 29.7 |

## スモール・レジスター・モードとラージ・レジスター・モード

インテル® データセンター GPU マックス・シリーズでは、2 つの汎用レジスター (GRF) モード、スモール GRF モードとラージ GRF モードを利用できます。インテル® グラフィックス・コンパイラー (IGC) には、これらのモードを選択する、(1) コマンドラインと (2) カーネルごとに指定する 2 つの方法があります。この節では、ユーザーが SYCL* バックエンドと OpenMP* バックエンドに対し、この制御を行うガイドラインを提供します。

### GRF - 概要

インテル® データセンター GPU マックス・シリーズ製品は、スモール GRF とラージ GRF の 2 つのモードをサポートしています。各エグゼキューション・ユニット (EU) には、レジスターとして利用できる合計 64KB ストレージがあります。スモール GRF モードでは、EU 内の単一ハードウェア・スレッドは、それぞれ 64B の 128 個のレジスターを利用できます。このモードでは、EU ごとに 8 つのハードウェア・スレッドがあります。ラージ GRF モードでは、EU 内の単一ハードウェア・スレッドは、それぞれ 64B の 256 個のレジスターを利用できます。このモードでは、EU ごとに 4 つのハードウェア・スレッドがあります。

### コマンドラインでの GRF モード指定

IGC グラフィックス・コンパイラーで GRF モードの選択に利用できるバックエンド・オプションを以下に示します。

- `-ze-opt-large-register-file`

  IGC はすべてにカーネルに対しラージ GRF モードを選択します。
- `-ze-opt-intel-enable-auto-large-GRF-mode`

  IGC はヒューリスティックに従ってカーネルごとにスモール/ラージ GRF モードを選択できるようになります。
- デフォルト

  IGC はすべてのカーネルに対しスモール GRF モードを選択します。

## OpenMP* - GRF モードの選択 (AOT)

以下に、OpenMP* バックエンドの AOT コンパイルで指定できるバックエンド・オプションの組み合わせを示します。ここではソースファイルに `test.cpp` を使用していますが任意のプログラムに変更できます。

```
$ icpx -fiopenmp -fopenmp-targets=spir64_gen -Xopenmp-target-backend
"-device pvc -options -ze-opt-large-register-file" test.cpp
// IGC はすべてのカーネルに対しラージ GRF モードを強制します。
```

```
$ icpx -fiopenmp -fopenmp-targets=spir64_gen -Xopenmp-target-backend
"-device pvc -options -ze-intel-enable-auto-large-GRF-mode" test.cpp
// IGC は、コンパイラーのヒューリスティックにしたがって、カーネルごとにスモール GRF とラージ GRF モードを選択します。
```

```
$ icpx -fiopenmp -fopenmp-targets=spir64_gen -Xopenmp-target-backend "-device pvc"
test.cpp
// IGC はすべてのカーネルに対しスモール GRF モードを選択します。
```

## OpenMP* - GRF モードの選択 (JIT)

OpenMP* バックエンドの JIT コンパイルでは、環境変数 `LIBOMPTARGET_LEVEL0_COMPILATION_OPTIONS` を使用してバックエンド・オプションを指定します。以下に利用可能なオプションを示します。

```
$ export LIBOMPTARGET_LEVEL0_COMPILATION_OPTIONS="-ze-opt-large-register-file"
// IGC はすべてのカーネルに対しラージ GRF モードを強制します。
$ export LIBOMPTARGET_LEVEL0_COMPILATION_OPTIONS
="-ze-intel-enable-auto-large-GRF-mode"
// IGC は、コンパイラーのヒューリスティックにしたがって、カーネルごとにスモール GRF とラージ GRF モードを選択します。
env-var not set
// IGC はすべてのカーネルに対しスモール GRF モードを選択します。
```

## SYCL* - GRF モードの選択 (AOT)

以下に、SYCL* バックエンドの AOT コンパイルで指定できるバックエンド・オプションの組み合わせを示します。ここではソースファイルに `test.cpp` を使用していますが任意の SYCL* プログラムに変更できます。

```
$ icpx -fsycl -fsycl-targets=spir64_gen -Xsycl-target-backend
"-device pvc -options -ze-opt-large-register-file" test.cpp
// IGC はすべてのカーネルに対しラージ GRF モードを強制します。
```

```
$ icpx -fsycl -fsycl-targets=spir64_gen -Xsycl-target-backend
"-device pvc -options -ze-intel-enable-auto-large-GRF-mode" test.cpp
// IGC は、コンパイラーのヒューリスティックにしたがって、カーネルごとにスモール GRF とラージ GRF モードを選択します。
```

```
$ icpx -fsycl -fsycl-targets=spir64_gen -Xsycl-target-backend "-device pvc" test.cpp
// IGC はすべてのカーネルに対しスモール GRF モードを選択します。
```

## SYCL* - GRF モードの選択 (JIT)

以下に、SYCL* バックエンドの JIT コンパイルで指定できるバックエンド・オプションの組み合わせを示します。ここではソースファイルに `test.cpp` を使用していますが任意の SYCL* プログラムに変更できます。

```
icpx -fsycl -fsycl-targets=spir64 -Xsycl-target-backend
"-options -ze-opt-large-register-file" test.cpp
// IGC はすべてのカーネルに対しラージ GRF モードを強制します。
```

```
icpx -fsycl -fsycl-targets=spir64 -Xsycl-target-backend
"-options -ze-intel-enable-auto-large-GRF-mode" test.cpp
// IGC は、コンパイラーのヒューリスティックにしたがって、カーネルごとにスモール GRF とラージ GRF モードを選択します。
```

```
icpx -fsycl -fsycl-targets=spir64 -Xsycl-target-backend "" test.cpp
// IGC はすべてのカーネルに対しスモール GRF モードを選択します。
```

## GRF モード選択を使用したパフォーマンス・チューニング

この節では、GRF モードの選択がデバイスコードのパフォーマンスに与える影響について説明します。ここでは、OpenMP* オフロード機能を使用した、JIT コンパイルを使用します。GRF モードを制御することで、(1) カーネルコードのレジスター・プレッシャー、と (2) 並列実行スレッド数、を考慮できます。次に OpenMP* オフロード領域を含むコードの例を示します。

```
1.  #pragma omp target teams distribute thread_limit(ZDIM *NX1 *NX1)
2.  for (int e = 0; e < nelt; e++) {
3.    double s_u[NX1 * NX1 * NX1];
4.    double s_D[NX1 * NX1];
5.    // SLM は 3 つの配列で使用されます
6.    double s_ur[NX1 * NX1 * NX1];
7.    double s_us[NX1 * NX1 * NX1];
8.    double s_ut[NX1 * NX1 * NX1];
9.  #pragma omp parallel for
10.   for (int inner = 0; inner < innerub; inner++) {
11.     int k = inner / (NX1 * NX1);
12.     int j = (inner - k * NX1 * NX1) / NX1;
13.     int i = inner - k * NX1 * NX1 - j * NX1;
14.     if (k == 0)
15.       s_D[I2(i, j)] = D[I2(i, j)];
16.     for (; k < NX1; k += ZDIM) {
17.       s_u[I3(i, j, k)] = u[I4(i, j, k, e)];
18.     }
19.   }
20. #pragma omp parallel for
21.   for (int inner = 0; inner < innerub; inner++) {
22.     int k = inner / (NX1 * NX1);
23.     int j = (inner - k * NX1 * NX1) / NX1;
24.     int i = inner - k * NX1 * NX1 - j * NX1;
25.     double r_G00, r_G01, r_G02, r_G11, r_G12, r_G22;
26.     for (; k < NX1; k += ZDIM) {
27.       double r_ur, r_us, r_ut;
28.       r_ur = r_us = r_ut = 0;
29. #ifdef FORCE_UNROLL
30. #pragma unroll NX1
31. #endif
32.       for (int m = 0; m < NX1; m++) {
33.         r_ur += s_D[I2(i, m)] * s_u[I3(m, j, k)];
34.         r_us += s_D[I2(j, m)] * s_u[I3(i, m, k)];
35.         r_ut += s_D[I2(k, m)] * s_u[I3(i, j, m)];
```

```
36.       }
37.       const unsigned gbase = 6 * I4(i, j, k, e);
38.       r_G00 = g[gbase + 0];
39.       r_G01 = g[gbase + 1];
40.       r_G02 = g[gbase + 2];
41.       s_ur[I3(i, j, k)] = r_G00 * r_ur + r_G01 * r_us + r_G02 * r_ut;
42.       r_G11 = g[gbase + 3];
43.       r_G12 = g[gbase + 4];
44.       s_us[I3(i, j, k)] = r_G01 * r_ur + r_G11 * r_us + r_G12 * r_ut;
45.       r_G22 = g[gbase + 5];
46.       s_ut[I3(i, j, k)] = r_G02 * r_ur + r_G12 * r_us + r_G22 * r_ut;
47.     }
48.   }
49. #pragma omp parallel for
50.   for (int inner = 0; inner < innerub; inner++) {
51.     int k = inner / (NX1 * NX1);
52.     int j = (inner - k * NX1 * NX1) / NX1;
53.     int i = inner - k * NX1 * NX1 - j * NX1;
54.     for (; k < NX1; k += ZDIM) {
55.       double wr = 0.0;
56.       for (int m = 0; m < NX1; m++) {
57.         double s_D_i = s_D[I2(m, i)];
58.         double s_D_j = s_D[I2(m, j)];
59.         double s_D_k = s_D[I2(m, k)];
60.         wr += s_D_i * s_ur[I3(m, j, k)] + s_D_j * s_us[I3(i, m, k)] +
61.           s_D_k * s_ut[I3(i, j, m)];
62.       }
63.       w[I4(i, j, k, e)] = wr;
64.     }
65.   }
66. }
```

ここで変更可能なパラメーターは 2 つあります。(1) 行番号 30 にある内部ループのアンロール係数 (2) 行番号 1 にある OpenMP* チーム数アンロール係数を変更することでレジスター・プレッシャーを制御できます。アンロール係数を大きくすると、レジスターへのプレッシャーは高くなります。OpenMP* チーム数を変更することで、並列スレッド数を制御できます。ここでは、デバイス上でのカーネル実行時間をパフォーマンスの指標とします。ユーザーの環境やデバイスの設定よって異なるため、ここでは実際の数値は示しません。以下に、観察された結果を示します。

- アンロールを無効にすると、スモール GRF モードでパフォーマンスが向上することが分かりました。これは、ラージ GRF モードのメリットを得るほど、レジスター・プレッシャーが高くないことを意味します。
- アンロールを有効にすると、ラージ GRF モードでパフォーマンスが向上することが分かりました。これは、レジスター・プレッシャーが高く、ラージ GRF モードがプレッシャーの軽減に有効であることを意味します。
- スモール GRF モードを使用すると、チーム数が増加し、ワークロードが大きくなる (プレッシャーが高まる) 場合、パフォーマンスが向上する傾向があります。

## 共有ローカルメモリー

ほとんどのケースでは、work-item はデータを共有して相互に通信を行う必要があります。一方、すべての work-group 内のすべての work-item はグローバルメモリーにアクセスできることから、データ共有と通信はグローバルメモリー経由で行うことができます。ただし、グローバルメモリーの帯域幅は低く、レイテンシーが長いため共有と通信の効率は低下します。ベクトルエンジン (VE) スレッドが同時に実行する sub-group 内の work-item は、データを共有して互いに効率良く通信できますが、sub-group 内の work-item 数は少ないため共有と通信の範囲は限定されます。

広範囲の work-item にアクセスできる高い帯域幅の低レイテンシーのメモリーは、work-item 間のデータ共有と通信には理想的です。インテル® GPU に搭載される共有ローカルメモリー (SLM) は、そのような用途向けに設計されています。

インテル® GPU の各 Xe コアには個別の SLM があります。SLM へのアクセスは、Xe コアの VE、または同じ Xe コアの VE で実行されるようにスケジュールされた同じ work-group 内の work-item に限定されます。これは Xe コア (または work-group) に対してローカルであり、同じ Xe コア (または同じ work-group 内の work-item) 内の VE によって共有されるため SLM と呼ばれます。SLM は各 Xe コアでオンチップであるため、グローバルメモリーよりもはるかに高い帯域幅と低いレイテンシーを持っています。また、SLM は work-group 内のすべての work-item にアクセスできることから、work-group サイズに応じて数百の work-item 間のデータ共有と通信に対応できます。

SLM は work-group が管理するキャッシュであると考えることができます。work-group が開始されると、work-group 内の work-item はグローバルメモリーから SLM にデータを明示的にロードでき、データを高速にアクセスするため work-group のライフタイム中 SLM に保持できます。work-group が終了する前に、明示的に work-item によって SLM 内のデータをグローバルメモリーに書き戻すことができます。work-group の実行が完了すると、SLM のデータも失われ無効化されます。SLM とグローバルメモリー間の整合性を保つのはプログラマーの責任です。適切に SLM を使用することでパフォーマンスを大幅に向上できます。

## 共有ローカルメモリーのサイズと work-group サイズ

オンチップの SLM のサイズは制限されます。work-group で利用可能なメモリー量はデバイスによって異なりますが、デバイスに照会することで確認できます。以下に例を示します。

```
std::cout << "Local Memory Size: "
          << q.get_device().get_info<sycl::info::device::local_mem_size>()
          << std::endl;
```

出力は以下のようになります。

```
Local Memory Size: 65536
```

サイズはバイト単位です。この GPU デバイスには、work-group ごとに 64,536 バイト (64KB) の SLM があることが分かります。

work-group が利用できる最大 SLM サイズを知ることが重要です。多くの場合、work-group で利用可能な SLM の合計サイズは、work-group 内の work-item 数で一定ではない関数です。SLM の最大数は、グループ内の work-item の総数、つまり work-group サイズを制限することがあります。例えば、最大 SLM サイズが 64KB で、それぞれの work-item が 512 バイトの SLM を必要とする場合、最大 work-group サイズが 128 を超えることはありません。

## バンク競合

SLM は、高帯域幅で同時アクセス可能な同じサイズのメモリーバンクに分割されます。バンクの合計数はデバイスによって異なります。この例では、連続する 64 バイトが 4 バイト (32 ビット) の粒度で 16 個のバンクに格納されます。異なるバンクへのアクセス要求は並行して処理できますが、同一バンクの異なるアドレス要求はバンク競合を発生させシリアル化されます。バンク競合はパフォーマンスに悪影響を及ぼします。次の例について考えてみます。

```
1.  constexpr int N = 32;
2.  int *data = sycl::malloc_shared<int>(N, q);
3.  auto e = q.submit([&](auto &h) {
4.    sycl::local_accessor<int, 1> slm(sycl::range(32 * 64), h);
5.    h.parallel_for(sycl::nd_range(sycl::range{N}, sycl::range{32}),
6.      [=](sycl::nd_item<1> it) {
7.        int i = it.get_global_linear_id();
8.        int j = it.get_local_linear_id();
9.        slm[j * 16] = 0;
10.       it.barrier(sycl::access::fence_space::local_space);
11.       for (int m = 0; m < 1024 * 1024; m++) {
12.         slm[j * 16] += i * m;
```

```
13.         it.barrier(sycl::access::fence_space::local_space);
14.       }
15.       data[i] = slm[j * 16];
16.   });
17. });
```

バンク数が 16 である場合、上記のすべての work-item は、同じバンクの異なるアドレスの読み取りと書き込みを行います。それぞれのバンクのメモリー帯域幅は全体の 1/16 になります。

次の例では、SLM バンクの競合がなく、すべての work-item が異なるバンクの異なるアドレスを読み書きするためメモリー帯域幅を完全に活用します。

```
1.  constexpr int N = 32;
2.  int *data = sycl::malloc_shared<int>(N, q);
3.  auto e = q.submit([&](auto &h) {
4.    sycl::local_accessor<int, 1> slm(sycl::range(32 * 64), h);
5.    h.parallel_for(sycl::nd_range(sycl::range{N}, sycl::range{32}),
6.      [=](sycl::nd_item<1> it) {
7.        int i = it.get_global_linear_id();
8.        int j = it.get_local_linear_id();
9.        slm[j] = 0;
10.       it.barrier(sycl::access::fence_space::local_space);
11.       for (int m = 0; m < 1024 * 1024; m++) {
12.         slm[j] += i * m;
13.         it.barrier(sycl::access::fence_space::local_space);
14.       }
15.       data[i] = slm[j];
16.   });
17. });
```

## データ共有と work-group バリア

「レジスターの使用とスピルの回避」の項の 256 値域のヒストグラムの例を見直してみましょう。

```
1.  constexpr int BLOCK_SIZE = 256;
2.  constexpr int NUM_BINS = 256;
3.  std::vector<unsigned long> hist(NUM_BINS, 0);
4.  sycl::buffer<unsigned long, 1> mbuf(input.data(), N);
5.  sycl::buffer<unsigned long, 1> hbuf(hist.data(), NUM_BINS);
6.  auto e = q.submit([&](auto &h) {
7.    sycl::accessor macc(mbuf, h, sycl::read_only);
8.    auto hacc = hbuf.get_access<sycl::access::mode::atomic>(h);
9.    h.parallel_for(
10.     sycl::nd_range(sycl::range{N / BLOCK_SIZE}, sycl::range{64}),
11.     [=](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(16)]] {
12.       int group = it.get_group()[0];
13.       int gSize = it.get_local_range()[0];
14.       auto sg = it.get_sub_group();
15.       int sgSize = sg.get_local_range()[0];
16.       int sgGroup = sg.get_group_id()[0];
17.       unsigned int histogram[NUM_BINS / 16]; // 分布図はレジスターに格納するには
18.                                              // ストレージを消費しすぎます
19.       for (int k = 0; k < NUM_BINS / 16; k++) {
20.         histogram[k] = 0;
21.       }
22.       for (int k = 0; k < BLOCK_SIZE; k++) {
23.         unsigned long x =
24.           sg.load(macc.get_pointer() + group * gSize * BLOCK_SIZE +
25.             sgGroup * sgSize * BLOCK_SIZE + sgSize * k);
26. // subgroup サイズは 16
27. #pragma unroll
28.         for (int j = 0; j < 16; j++) {
29.           unsigned long y = sycl::group_broadcast(sg, x, j);
```

```
#pragma unroll
          for (int i = 0; i < 8; i++) {
            unsigned int c = y & 0xFF;
                              // (c & 0xF) は値域が存在する work-item です。
                              // (c >> 4) は値域のインデックスです。
            if (sg.get_local_id()[0] == (c & 0xF)) {
              histogram[c >> 4] += 1;
            }
            y = y >> 8;
          }
        }
      }
      for (int k = 0; k < NUM_BINS / 16; k++) {
        hacc[16 * k + sg.get_local_id()[0]].fetch_add(histogram[k]);
      }
  });
});
```

この例を最適化して、long の代わりに int データタイプを使用し、最適なパフォーマンスを得るためプライベート・ヒストグラムの値域がレジスターに収まるようにして sub-group 内のレジスターを共有します。より多くの値域サイズ (例えば 1024) が必要な場合、プライベート・ヒストグラムの値域がグローバルメモリーにあふれることは避けられません。

ヒストグラムの値域は、それぞれの値域がアトミックに更新される限り、work-group 内の work-item 間で共有できます。

```
constexpr int NUM_BINS = 1024;
constexpr int BLOCK_SIZE = 256;
std::vector<unsigned long> hist(NUM_BINS, 0);
sycl::buffer<unsigned long, 1> mbuf(input.data(), N);
sycl::buffer<unsigned long, 1> hbuf(hist.data(), NUM_BINS);
auto e = q.submit([&](auto &h) {
  sycl::accessor macc(mbuf, h, sycl::read_only);
  sycl::accessor hacc(hbuf, h, sycl::read_write);
   sycl::local_accessor<unsigned int, 1> local_histogram(sycl::range(NUM_BINS), h);
  h.parallel_for(
    sycl::nd_range(sycl::range{N / BLOCK_SIZE}, sycl::range{64}),
    [=](sycl::nd_item<1> it) {
      int group = it.get_group()[0];
      int gSize = it.get_local_range()[0];
      auto sg = it.get_sub_group();
      int sgSize = sg.get_local_range()[0];
      int sgGroup = sg.get_group_id()[0];
      int factor = NUM_BINS / gSize;
      int local_id = it.get_local_id()[0];
      if ((factor <= 1) && (local_id < NUM_BINS)) {
        sycl::atomic_ref<unsigned int, sycl::memory_order::relaxed,
          sycl::memory_scope::device,
          sycl::access::address_space::local_space>
          local_bin(local_histogram[local_id]);
        local_bin.store(0);
      } else {
        for (int k = 0; k < factor; k++) {
          sycl::atomic_ref<unsigned int, sycl::memory_order::relaxed,
            sycl::memory_scope::device,
            sycl::access::address_space::local_space>
            local_bin(local_histogram[gSize * k + local_id]);
          local_bin.store(0);
        }
      }
      it.barrier(sycl::access::fence_space::local_space);
      for (int k = 0; k < BLOCK_SIZE; k++) {
        unsigned long x =
          sg.load(macc.get_pointer() + group * gSize * BLOCK_SIZE +
          sgGroup * sgSize * BLOCK_SIZE + sgSize * k);
```

```
40. #pragma unroll
41.       for (std::uint8_t shift : {0, 16, 32, 48}) {
42.         constexpr unsigned long mask = 0x3FFU;
43.         sycl::atomic_ref<unsigned int, sycl::memory_order::relaxed,
44.           sycl::memory_scope::device,
45.           sycl::access::address_space::local_space>
46.           local_bin(local_histogram[(x >> shift) & mask]);
47.         local_bin += 1;
48.       }
49.     }
50.     it.barrier(sycl::access::fence_space::local_space);
51.     if ((factor <= 1) && (local_id < NUM_BINS)) {
52.       sycl::atomic_ref<unsigned int, sycl::memory_order::relaxed,
53.         sycl::memory_scope::device,
54.         sycl::access::address_space::local_space>
55.         local_bin(local_histogram[local_id]);
56.       sycl::atomic_ref<unsigned long, sycl::memory_order::relaxed,
57.         sycl::memory_scope::device,
58.         sycl::access::address_space::global_space>
59.         global_bin(hacc[local_id]);
60.       global_bin += local_bin.load();
61.     } else {
62.       for (int k = 0; k < factor; k++) {
63.         sycl::atomic_ref<unsigned int, sycl::memory_order::relaxed,
64.           sycl::memory_scope::device,
65.           sycl::access::address_space::local_space>
66.           local_bin(local_histogram[gSize * k + local_id]);
67.         sycl::atomic_ref<unsigned long, sycl::memory_order::relaxed,
68.           sycl::memory_scope::device,
69.           sycl::access::address_space::global_space>
70.           global_bin(hacc[gSize * k + local_id]);
71.         global_bin += local_bin.load();
72.       }
73.     }
74.   });
75. });
```

work-group が開始されると、work-group 内の各 work-item は SLM の `histogram` の値域を 0 に初期化します (上記コードの 20 - 34 行目)。単一の work-item ですべての `histogram` の値域を初期化することもできますが、通常は、work-group 内のすべての work-item でジョブを分散するほうが効果的です。

35 行目にある初期化後の work-group バリアは、work-item が値域を更新する前にすべての `histogram` 値域が 0 に初期化されることを保証します。

SLM にある `histogram` 値域はすべての work-item で共有されるため、任意の work-item による値域の更新はアトミックでなければなりません。

work-group 内のローカル分布図が完成すると、グローバル `histogram` が更新されます。ただし、ローカル SLM の値域を読み取ってグローバル値域を更新する前に、50 行目で work-group バリアを再度呼び出してすべての work-item のワークが完了するのを待機します。

SLM のデータを共有する場合、work-item の同期に work-group バリアが必要になることがあります。バリアにはコストが生じるため、work-group のサイズが増えるとコストも増加します。アプリケーションに最適な work-group サイズを特定するため、いろいろなサイズを試してください。

また、`examples` フォルダーには 256 個の値域を持つ SLM バージョンのヒストグラムも用意されています。レジスターを使用したバージョンとパフォーマンスを比較することもできます。予想を上回る結果を得られても、さらにパフォーマンスを向上する最適化が可能であるか考えることもあるでしょう。これは演習として皆さんにお任せします。

## キャッシュとして SLM を使用

ハードウェアにキャッシュの管理を任せるよりも、アプリケーションでホットなデータの管理を行うほうが望ましい場合があります。アプリケーションでデータのキャッシングを直接管理すると、データが必要なときにそのデータがどこにあるのか、そしてデータアクセスのコストを正確に推測できます。SLM はそのような用途に利用できます。

次の 1-D 畳み込みの例について考えてみます。

```
1.  sycl::buffer<int> ibuf(input.data(), N);
2.  sycl::buffer<int> obuf(output.data(), N);
3.  sycl::buffer<int> kbuf(kernel.data(), M);
4.  auto e = q.submit([&](auto &h) {
5.  sycl::accessor iacc(ibuf, h, sycl::read_only);
6.  sycl::accessor oacc(obuf, h);
7.  sycl::accessor kacc(kbuf, h, sycl::read_only);
8.  h.parallel_for(sycl::nd_range<1>(sycl::range{N}, sycl::range{256}),
9.    [=](sycl::nd_item<1> it) {
10.     int i = it.get_global_linear_id();
11.     int group = it.get_group()[0];
12.     int gSize = it.get_local_range()[0];
13.     int t = 0;
14.     int _M = static_cast<int>(M);
15.     int _N = static_cast<int>(N);
16.     if ((group == 0) || (group == _N / gSize - 1)) {
17.       if (i < _M / 2) {
18.         for (int j = _M / 2 - i, k = 0; j < _M; ++j, ++k) {
19.           t += iacc[k] * kacc[j];
20.         }
21.       } else {
22.         if (i + _M / 2 >= _N) {
23.           for (int j = 0, k = i - _M / 2;j < _M / 2 + _N - i; ++j, ++k) {
24.             t += iacc[k] * kacc[j];
25.           }
26.         } else {
27.           for (int j = 0, k = i - _M / 2; j < _M; ++j, ++k) {
28.             t += iacc[k] * kacc[j];
29.           }
30.         }
31.       }
32.     } else {
33.       for (int j = 0, k = i - _M / 2; j < _M; ++j, ++k) {
34.         t += iacc[k] * kacc[j];
35.       }
36.     }
37.     oacc[i] = t;
38.   });
39. });
```

この例では、257 要素のカーネル配列を使用して、8192 x 8192 要素の整数配列を畳み込んで結果を出力配列に書き込みます。各 work-item は 1 つの要素を畳み込みます。ただし、1 つの要素を畳み込むには、最大 256 個の隣接する要素が必要になります。

各入力要素が複数の work-item で使用されていることを知っていれば、work-group 全体で必要な要素を SLM に事前ロードすることができ、以降で要素が必要になった際に、グローバルメモリーに代わって SLM からロードできます。

```
1.  sycl::buffer<int> ibuf(input.data(), N);
2.  sycl::buffer<int> obuf(output.data(), N);
3.  sycl::buffer<int> kbuf(kernel.data(), M);
4.  auto e = q.submit([&](auto &h) {
5.    sycl::accessor iacc(ibuf, h, sycl::read_only);
6.    sycl::accessor oacc(obuf, h);
```

```
7.    sycl::accessor kacc(kbuf, h, sycl::read_only);
8.    sycl::local_accessor<int, 1> ciacc(sycl::range(256 + (M / 2) * 2), h);
9.    h.parallel_for(
10.     sycl::nd_range(sycl::range{N}, sycl::range{256}),
11.     [=](sycl::nd_item<1> it) {
12.       int i = it.get_global_linear_id();
13.       int group = it.get_group()[0];
14.       int gSize = it.get_local_range()[0];
15.       int local_id = it.get_local_id()[0];
16.       int _M = static_cast<int>(M);
17.       ciacc[local_id + M / 2] = iacc[i];
18.       if (local_id == 0) {
19.         if (group == 0) {
20.           for (int j = 0; j < _M / 2; ++j) {
21.             ciacc[j] = 0;
22.           }
23.         } else {
24.           for (int j = 0, k = i - _M / 2; j < _M / 2; ++j, ++k) {
25.             ciacc[j] = iacc[k];
26.           }
27.         }
28.       }
29.       if (local_id == gSize - 1) {
30.         if (group == static_cast<int>(it.get_group_range()[0]) - 1) {
31.           for (int j = gSize + _M / 2; j < gSize + _M / 2 + _M / 2; ++j) {
32.             ciacc[j] = 0;
33.           }
34.         } else {
35.           for (int j = gSize + _M / 2, k = i + 1;j < gSize+_M/2+_M /2; ++j, ++k) {
36.             ciacc[j] = iacc[k];
37.           }
38.         }
39.       }
40.       it.barrier(sycl::access::fence_space::local_space);
41.       int t = 0;
42.       for (int j = 0, k = local_id; j < _M; ++j, ++k) {
43.         t += ciacc[k] * kacc[j];
44.       }
45.       oacc[i] = t;
46.   });
47. });
```

work-group が開始されると、各 work-item で必要とされるすべての入力要素は SLM にロードされます。最初と最後の work-item を除いて、各 work-item は 1 つの要素を SLM にロードします。最初の work-item は最初の要素の左に隣接する要素をロードし、最後の work-item は SLM の最後の要素の右に隣接する要素をロードします。隣接する要素がない場合、SLM の要素は 0 で埋められます。

各 work-item で畳み込みを開始する前に、ローカルバリアを呼び出し、すべての入力要素が SLM にロードされるのを待機します。

各 work-item の畳み込みはシンプルです。隣接するすべての要素は、グローバルメモリーではなく高速な SLM からロードされます。

## SLM エラーのトラブルシューティング

`CL_OUT_OF_RESOURCES` エラーは、カーネルがエミュレーターのデフォルトでサポートされる `__private` メモリーもしくは `__local` メモリーよりも多くのメモリーを使用すると発生する可能性があります。このエラーが発生すると、次のようなメッセージが表示されます。

```
$ ./myapp
:
Problem size: c(150,600) = a(150,300) * b(300,600)
```

```
terminate called after throwing an instance of
    'cl::sycl::runtime_error'
    what(): Native API failed. Native API returns:
    -5 (CL_OUT_OF_RESOURCES) -5 (CL_OUT_OF_RESOURCES)
Aborted (core dumped)
$
```

また、`onetrace` を使用する場合は次のようなメッセージが表示されます。

```
$ onetrace -c ./myapp
:
>>>> [6254070891] zeKernelSuggestGroupSize: hKernel = 0x263b7a0 globalSizeX = 163850
globalSizeY = 1 globalSizeZ = 1 groupSizeX = 0x7fff94e239f0 groupSizeY =
0x7fff94e239f4 groupSizeZ = 0x7fff94e239f8
<<<< [6254082074] zeKernelSuggestGroupSize [922 ns] ->
ZE_RESULT_ERROR_OUT_OF_DEVICE_MEMORY(0x1879048195)
terminate called after throwing an instance of 'cl::sycl::runtime_error'
    what(): Native API failed. Native API returns: -5 (CL_OUT_OF_RESOURCES) -5
(CL_OUT_OF_RESOURCES)
Aborted (core dumped)
$
```

共有ローカルメモリーにコピーされたメモリーとハードウェアの上限を確認するには、デバッグキーを設定します。

```
export PrintDebugMessages=1 export NEOReadDebugKeys=1
```

これにより、出力は次のようになります。

```
$ ./myapp
:
Size of SLM (656384) larger than available (131072)
terminate called after throwing an instance of 'cl::sycl::runtime_error' what():
Native API failed.Native API returns: -5 (CL_OUT_OF_RESOURCES) -5
(CL_OUT_OF_RESOURCES)
Aborted (core dumped)
$
```

また、`onetrace` を使用する場合は次のようになります。

```
$ onetrace -c ./myapp :
>>>> [317651739] zeKernelSuggestGroupSize: hKernel = 0x2175ae0 globalSizeX = 163850
globalSizeY = 1 globalSizeZ = 1 groupSizeX = 0x7ffd9caf0950 groupSizeY =
0x7ffd9caf0954 groupSizeZ = 0x7ffd9caf0958
Size of SLM (656384) larger than available (131072) <<<< [317672417]
zeKernelSuggestGroupSize [10325 ns] ->
ZE_RESULT_ERROR_OUT_OF_DEVICE_MEMORY(0x1879048195)
terminate called after throwing an instance of 'cl::sycl::runtime_error'
    what(): Native API failed.Native API returns: -5 (CL_OUT_OF_RESOURCES) -5
(CL_OUT_OF_RESOURCES)
Aborted (core dumped)
$
```

必要なメモリー使用量が決まったら、インテル® oneAPI プログラミング・ガイドの「エミュレーターの環境設定」(英語) で説明されているように、環境変数 `CL_CONFIG_CPU_FORCE_PRIVATE_MEM_SIZE` または `CL_CONFIG_CPU_FORCE_LOCAL_MEM_SIZE` により大きな値を設定してください。

## ポインターのエイリアシングと restrict ディレクティブ

カーネルは通常、ポインター引数として渡される要素の配列を操作します。コンパイラーがポインターの相互エイリアスを判断できない場合、安全性の観点から保守的にエイリアスがあると想定し、コンパイラーはポインターに対する操作のリオーダーを行いません。ループ反復内に 2 つのロードと 1 つのストアがあるベクトル加算の例を考えてみます。

```
1.  size_t VectorAdd(sycl::queue &q, const IntArray &a, const IntArray &b,
2.                   IntArray &sum, int iter) {
3.    sycl::range num_items{a.size()};
4.    sycl::buffer a_buf(a);
5.    sycl::buffer b_buf(b);
6.    sycl::buffer sum_buf(sum.data(), num_items);
7.    auto start = std::chrono::steady_clock::now();
8.    for (int i = 0; i < iter; i++) {
9.      auto e = q.submit([&](auto &h) {
10.       // 入力アクセサー
11.       sycl::accessor a_acc(a_buf, h, sycl::read_only);
12.       sycl::accessor b_acc(b_buf, h, sycl::read_only);
13.       // 出力アクセサー
14.       sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
15.       h.parallel_for(num_items,
16.         [=](auto i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
17.       });
18.   }
19.   q.wait();
20.   auto end = std::chrono::steady_clock::now();
21.   std::cout << "Vector add completed on device - took " << (end - start).count()
22.             << " u-secs\n";
23.   return ((end - start).count());
24. }// VectorAdd の終わり
```

ここでは、プログラマーはベクトル長と work-group 数の選択をコンパイラーに任せています。ほとんどのケースでは、コンパイラーは最高のパフォーマンスを得るため適切なパラメーターを選択することができます。しかし、パフォーマンス向上のため work-group 数と work-group サイズを明示的に指定して、コンパイラーにヒントを与えることが適切である場合があります。

次のカーネルは、work-item ごとに配列の複数の要素を処理するように記述されており、work-group 数と work-group サイズを明示的に指定しています。23 行目の `intel::kernel_args_restrict` は、このカーネルのバッファーアクセサーが互いにエイリアスを持たないことをコンパイラーに通知します。これにより、コンパイラーはロードとストアのホイストが可能となり、命令が完了するまでの時間が長くなることから命令のスケジュールが改善されます。26 行目のプラグマは、2 回のアンロールを行うことをコンパイラーに指示します。

```
size_t VectorAdd2(sycl::queue &q, const IntArray &a, const IntArray &b,
                  IntArray &sum, int iter) {
  sycl::range num_items{a.size()};
  sycl::buffer a_buf(a);
  sycl::buffer b_buf(b);
  sycl::buffer sum_buf(sum.data(), num_items);
// size_t num_groups =
// q.get_device().get_info<sycl::info::device::max_compute_units>(); size_t
// wg_size =
// q.get_device().get_info<sycl::info::device::max_work_group_size>();
  size_t num_groups = 1;
  size_t wg_size = 16;
  auto start = std::chrono::steady_clock::now();
  for (int i = 0; i < iter; i++) {
    q.submit([&](auto &h) {
      // 入力アクセサー
      sycl::accessor a_acc(a_buf, h, sycl::read_only);
      sycl::accessor b_acc(b_buf, h, sycl::read_only);
      // 出力アクセサー
      sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
      h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
        [=](sycl::nd_item<1> index) [[intel::reqd_sub_group_size(16)]]
        [[intel::kernel_args_restrict]] {
          size_t loc_id = index.get_local_id();
          // unroll with a directive
#pragma unroll(2)
          for (size_t i = loc_id; i < mysize; i += wg_size) {
            sum_acc[i] = a_acc[i] + b_acc[i];
          }
      });
    });
  }
  q.wait();
  auto end = std::chrono::steady_clock::now();
  std::cout << "Vector add2 completed on device - took "
            << (end - start).count() << " u-secs\n";
  return ((end - start).count());
} // VectorAdd2 の終わり
```

次のカーネルは、コンパイラー・ディレクティブの代わりに、ループを手動でアンロールする方法を示します (コンパイラーは、内部のヒューリスティックによるコストモデルによって、ディレクティブを受け入れる場合とそうでない場合があります)。アンロールの利点は、ループを繰り返す回数が減るため、実行される命令数が少なくなることです (比較命令と分岐命令の数が減ります)。

```
size_t VectorAdd3(sycl::queue &q, const IntArray &a, const IntArray &b,
                  IntArray &sum, int iter) {
  sycl::range num_items{a.size()};
  sycl::buffer a_buf(a);
  sycl::buffer b_buf(b);
  sycl::buffer sum_buf(sum.data(), num_items);
  size_t num_groups = 1;
  size_t wg_size = 16;
  auto start = std::chrono::steady_clock::now();
  for (int i = 0; i < iter; i++) {
    q.submit([&](auto &h) {
      // 入力アクセサー
      sycl::accessor a_acc(a_buf, h, sycl::read_only);
      sycl::accessor b_acc(b_buf, h, sycl::read_only);
      // 出力アクセサー
      sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
      h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
        [=](sycl::nd_item<1> index) [[intel::reqd_sub_group_size(16)]] {
        // 手動でアンロール
        size_t loc_id = index.get_local_id();
```

```
21.          for (size_t i = loc_id; i < mysize; i += 32) {
22.            sum_acc[i] = a_acc[i] + b_acc[i];
23.            sum_acc[i + 16] = a_acc[i + 16] + b_acc[i + 16];
24.          }
25.        });
26.      });
27.    }
28.    q.wait();
29.    auto end = std::chrono::steady_clock::now();
30.    std::cout << "Vector add3 completed on device - took "
31.              << (end - start).count() << " u-secs\n";
32.    return ((end - start).count());
33. } // VectorAdd3 の終わり
```

次のカーネルは、ロードとストアを再配置することで、データを必要とする操作が実行される前にロードが発行されるようにする方法を示しています。通常、GPU 上のすべてのスレッドには多くの未処理のロードが存在する可能性があります。ロードするデータを処理する前にロードを発行することを推奨します。これにより、データが計算で実際に使用される前にロードを完了することが可能になります。

```
1.  size_t VectorAdd4(sycl::queue &q, const IntArray &a, const IntArray &b,
2.                    IntArray &sum, int iter) {
3.    sycl::range num_items{a.size()};
4.    sycl::buffer a_buf(a);
5.    sycl::buffer b_buf(b);
6.    sycl::buffer sum_buf(sum.data(), num_items);
7.    size_t num_groups = 1;
8.    size_t wg_size = 16;
9.    auto start = std::chrono::steady_clock::now();
10.   for (int i = 0; i < iter; i++) {
11.     q.submit([&](auto &h) {
12.       // 入力アクセサー
13.       sycl::accessor a_acc(a_buf, h, sycl::read_only);
14.       sycl::accessor b_acc(b_buf, h, sycl::read_only);
15.       // 出力アクセサー
16.       sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
17.       h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
18.         [=](sycl::nd_item<1> index)[[intel::reqd_sub_group_size(16)]] {
19.           // 手動アンロール
20.           size_t loc_id = index.get_local_id();
21.           for (size_t i = loc_id; i < mysize; i += 32) {
22.             int t1 = a_acc[i];
23.             int t2 = b_acc[i];
24.             int t3 = a_acc[i + 16];
25.             int t4 = b_acc[i + 16];
26.             sum_acc[i] = t1 + t2;
27.             sum_acc[i + 16] = t3 + t4;
28.           }
29.       });
30.     });
31.   }
32.   q.wait();
33.   auto end = std::chrono::steady_clock::now();
34.   std::cout << "Vector add4 completed on device - took "
35.             << (end - start).count() << " u-secs\n";
36.   return ((end - start).count());
37. } // VectorAdd4 の終わり
```

次のカーネルでは、restrict ディレクティブが追加されています。このディレクティブは、ループ内で書き込まれるベクトルにエイリアスがなく、前述の例では手動で行ったようなストアを超えてロードをホイスト可能であるヒントをコンパイラーに与えます。

```
1.  size_t VectorAdd5(sycl::queue &q, const IntArray &a, const IntArray &b,
2.                    IntArray &sum, int iter) {
3.    sycl::range num_items{a.size()};
4.    sycl::buffer a_buf(a);
5.    sycl::buffer b_buf(b);
6.    sycl::buffer sum_buf(sum.data(), num_items);
7.    size_t num_groups = 1;
8.    size_t wg_size = 16;
9.    auto start = std::chrono::steady_clock::now();
10.   for (int i = 0; i < iter; i++) {
11.     q.submit([&](auto &h) {
12.       // 入力アクセサー
13.       sycl::accessor a_acc(a_buf, h, sycl::read_only);
14.       sycl::accessor b_acc(b_buf, h, sycl::read_only);
15.       // 出力アクセサー
16.       sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
17.       h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
18.         [=](sycl::nd_item<1> index) [[intel::reqd_sub_group_size(
19.         16)]] [[intel::kernel_args_restrict]] {
20.           // コンパイラーはロードをホイストする必要があります
21.           size_t loc_id = index.get_local_id();
22.           for (size_t i = loc_id; i < mysize; i += 32) {
23.             sum_acc[i] = a_acc[i] + b_acc[i];
24.             sum_acc[i + 16] = a_acc[i + 16] + b_acc[i + 16];
25.           }
26.       });
27.     });
28.   }
29.   q.wait();
30.   auto end = std::chrono::steady_clock::now();
31.   std::cout << "Vector add5 completed on device - took "
32.             << (end - start).count() << " u-secs\n";
33.   return ((end - start).count());
34. } // VectorAdd5 の終わり
```

## カーネル内のスレッド間での同期

カーネル内の work-item を同期してデータ交換や更新を行ったり、相互に連携して特定の順番でタスクを実行するにはさまざまな方法があります。次にその方法を示します。

- **アクセサークラス**
  アクセサークラスは、バッファーおよびイメージデータ構造の取得と解放を指定します。それらが生成および破棄される場所に応じて、ランタイムは適切なデータ転送を行い、同期プリミティブを生成します。

- **アトミック操作**
  SYCL* デバイスは、C++ アトミックの制限があるサブセットをサポートします。

- **フェンス**
  フェンス・プリミティブは、ロードとストアを順序付けするために使用されます。フェンスはセマンティクスの取得と解放、またはその両方を保持できます。

- **バリア**
  バリアは、それぞれのグループ内の work-item を同期するために使用されます。

- **階層的な並列ディスパッチ**
  計算を表現する階層並列処理モデルでは、work-group 内の同期は明示的な work-group バリア操作を使用せず、`parallel_for_work_item` 関数呼び出しの複数インスタンスを介して明示的に行われます。

- **デバイスイベント**
  イベントは、カーネル関数内での非同期操作の完了待機に使用されます。

多くの場合、上記の同期イベントのいずれかを使用して同じ機能を表現できますが、効率とパフォーマンスが異なります。


- アトミック操作
- ローカルバリアとグローバルバリア


## アトミック操作

アトミックはメモリーを介した work-item 間の通信で複数の work-item を同期できます。SYCL* アトミックは C++ アトミックに類似しており、アトミックで保護されるリソースへのアクセスが単一ユニットとして実行することを保証しています。アトミック操作のパフォーマンスと正当性には、次の要因が影響します。

- データタイプ
- バッファーとローカルアドレス空間
- ホスト、共有、デバイスに割り当てられた USM

## アトミック操作のデータタイプ

次のカーネルは、すべての work-item がグローバル・アキュムレーターをアトミックに更新する SYCL* リダクション操作の実装を示しています。加算の入力データタイプと、リダクション操作が適用されるベクトルは整数です。このカーネルのパフォーマンスは、リダクションに使用されるブロック化など他の手法と比較しても妥当と言えます。

```
1.  q.submit([&](auto &h) {
2.    sycl::accessor buf_acc(buf, h, sycl::read_only);
3.    sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
4.    h.parallel_for(data_size, [=](auto index) {
5.      size_t glob_id = index[0];
6.        auto v = sycl::atomic_ref<int,
7.          sycl::memory_order::relaxed,
8.          sycl::memory_scope::device, sycl::access::address_space::global_space>(
9.        sum_acc[0]);
10.       v.fetch_add(buf_acc[glob_id]);
11.   });
12. });
```

次のカーネルのようにベクトルのデータタイプが `float` または `double` である場合、`float` と `double` にはハードウェアによるアトミック操作のサポートがないことからアクセラレーターのパフォーマンスが低くなることがあります。次の 2 つのカーネルは、アトミック加算を実行する時間が、ネイティブでアトミック操作がサポートされるかどうかにより大幅に異なる可能性があることを示しています。

```
1.  //
2.  int VectorInt(sycl::queue &q, int iter) {
3.    VectorAllocator<int> alloc;
4.    AlignedVector<int> a(array_size, alloc);
5.    AlignedVector<int> b(array_size, alloc);
6.    InitializeArray<int>(a);
7.    InitializeArray<int>(b);
8.    sycl::range num_items{a.size()};
```

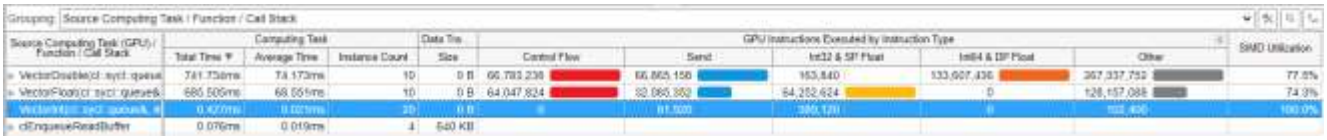
```
9.    sycl::buffer a_buf(a);
10.   sycl::buffer b_buf(b);
11.   auto start = std::chrono::steady_clock::now();
12.   for (int i = 0; i < iter; i++) {
13.     q.submit([&](sycl::handler &h) {
14.       // 入力アクセサー
15.       sycl::accessor a_acc(a_buf, h, sycl::read_write);
16.       sycl::accessor b_acc(a_buf, h, sycl::read_only);
17.       h.parallel_for(num_items, [=](auto i) {
18.         auto v = sycl::atomic_ref<int, sycl::memory_order::relaxed,
19.         sycl::memory_scope::device,
20.           sycl::access::address_space::global_space>(
21.         a_acc[0]);
22.         v += b_acc[i];
23.       });
24.     });
25.   }
26.   q.wait();
27.   auto end = std::chrono::steady_clock::now();
28.   std::cout << "Vector int completed on device - took "
29.             << (end - start).count()
30.             << " u-secs\n";
31.   return ((end - start).count());
32. }
```

アトミックを使用する場合、ハードウェアによるサポートがあり効率良く実行されるか確認する必要があります。Gen9 およびインテル® Iris® Xᵉ 統合グラフィックスでは、`float` と `double` データタイプのアトミックはサポートされていないため、`VectorDouble` のパフォーマンスは非常に低くなります。

```
1.  //
2.  int VectorDouble(sycl::queue &q, int iter) {
3.    VectorAllocator<double> alloc;
4.    AlignedVector<double> a(array_size, alloc);
5.    AlignedVector<double> b(array_size, alloc);
6.    InitializeArray<double>(a);
7.    InitializeArray<double>(b);
8.    sycl::range num_items{a.size()};
9.    sycl::buffer a_buf(a);
10.   sycl::buffer b_buf(b);
11.   auto start = std::chrono::steady_clock::now();
12.   for (int i = 0; i < iter; i++) {
13.     q.submit([&](sycl::handler &h) {
14.       // 入力アクセサー
15.       sycl::accessor a_acc(a_buf, h, sycl::read_write);
16.       sycl::accessor b_acc(a_buf, h, sycl::read_only);
17.       h.parallel_for(num_items, [=](auto i) {
18.         auto v = sycl::atomic_ref<double, sycl::memory_order::relaxed,
19.           sycl::memory_scope::device,
20.           sycl::access::address_space::global_space>(
21.             a_acc[0]);
22.             v += b_acc[i];
23.       });
24.     });
25.   }
26.   q.wait();
27.   auto end = std::chrono::steady_clock::now();
28.   std::cout << "Vector Double completed on device - took "
29.             << (end - start).count() << " u-secs\n";
30.   return ((end - start).count());
31. }
```

インテル® VTune™ プロファイラーを使用してこれらのカーネルを解析すると、ネイティブアトミックの影響を測定することができます。`VectorInt` カーネルは、`VectorDouble` および `VectorFloat` カーネルよりもはるかに高速であることが分かります。

### インテル® VTune™ プロファイラーにおける動的命令



インテル® VTune™ プロファイラーの動的命令解析を使用すると、ネイティブアトミックがサポートされていない場合、命令数が劇的に変化することが分かります。

以下に `VectorInt` カーネルのアセンブリー・コードを示します。

### インテル® VTune™ プロファイラーにおけるアトミック int

| Addr... ▲ | So... | Assembly | GPU Instructions Executed |
|---|---|---|---|
| **0** | | **Block 1:** | |
| 0 | 53 | (W) mov (8\|M0) r3.0<1>:ud r0.0<1;1,0>:ud | 20,480 |
| 0x10 | 53 | (W) or (1\|M0) cr0.0<1>:ud cr0.0<0;1,0>:ud 0x4C0:uw {Swi | 20,480 |
| 0x20 | | (W) mul (1\|M0) r10.0<1>:d r11.0<0;1,0>:d r3.1<0;1,0>:d | 20,480 |
| 0x28 | | (W) mov (16\|M0) r11.0<1>:d 0:w | 20,480 |
| 0x38 | | (W) shl (1\|M0) r10.2<1>:d r9.6<0;1,0>:d 2:w | 20,480 |
| 0x48 | | (W) mov (16\|M16) r4.0<1>:d 0:w | 20,480 |
| 0x58 | | (W) shl (1\|M0) r10.1<1>:d r9.0<0;1,0>:d 2:w | 20,480 |
| 0x68 | | add (16\|M0) r8.0<1>:d r10.0<0;1,0>:d r1.0<16;16,1>:uw | 20,480 |
| 0x78 | | add (16\|M16) r4.0<1>:d r10.0<0;1,0>:d r2.0<16;16,1>:uw | 20,480 |
| 0x88 | | (W) add (1\|M0) r10.0<1>:d r10.2<0;1,0>:d r10.5<0;1,0>:d | 20,480 |
| 0x98 | | (W) mov (8\|M0) r112.0<1>:ud r3.0<8;8,1>:ud {Compacted} | 20,480 |
| 0xa0 | | (W) add (1\|M0) r14.0<1>:d r10.1<0;1,0>:d r10.4<0;1,0>:d | 20,480 |
| 0xb0 | | add (16\|M0) r8.0<1>:d r8.0<8;8,1>:d r7.0<0;1,0>:d {Comp | 20,480 |
| 0xb8 | | add (16\|M16) r4.0<1>:d r4.0<8;8,1>:d r7.0<0;1,0>:d | 20,480 |
| 0xc8 | | shl (16\|M0) r8.0<1>:d r8.0<8;8,1>:d 2:w | 20,480 |
| 0xd8 | | shl (16\|M16) r4.0<1>:d r4.0<8;8,1>:d 2:w | 20,480 |
| 0xe8 | | add (16\|M0) r8.0<1>:d r10.0<0;1,0>:d r8.0<8;8,1>:d {Com | 20,480 |
| 0xf0 | | add (16\|M16) r4.0<1>:d r10.0<0;1,0>:d r4.0<8;8,1>:d | 20,480 |
| 0x100 | | send (16\|M0) r6:w r8 0xC 0x04205E01 [Data Cache Data Po | 20,480 |
| 0x110 | | send (16\|M16) r4:w r4 0xC 0x04205E01 [Data Cache Data P | 20,480 |
| 0x120 | | mov (16\|M0) r11.0<1>:d r6.0<8;8,1>:d {Compacted} | 20,480 |
| 0x128 | | (W) add (16\|M0) r4.0<1>:d r11.0<8;8,1>:d r4.0<8;8,1>:d | 20,480 |
| 0x130 | | (W) add (8\|M0) r2.0<1>:d r4.0<8;8,1>:d r5.0<8;8,1>:d {C | 20,480 |
| 0x138 | | (W) add (4\|M0) r2.0<1>:d r2.0<4;4,1>:d r2.4<4;4,1>:d {C | 20,480 |
| 0x140 | | (W) add (2\|M0) r2.0<1>:d r2.0<2;2,1>:d r2.2<2;2,1>:d | 20,480 |
| 0x150 | | (W) add (1\|M0) r2.0<1>:d r2.0<0;1,0>:d r2.1<0;1,0>:d {C | 20,480 |
| 0x158 | | (W) sends (1\|M0) null:ud r14 r2 0x4C 0x02009700 [Data C | 20,480 |
| 0x168 | | (W) send (8\|M0) null r112 0x27 0x02000010 {EOT} [Thread | 20,480 |

`VectorDouble` のアセンブリー・コードと比較すると、`VectorDouble` カーネルの実行には、さらに 3,300 万の GPU 命令が必要であることが分かります。

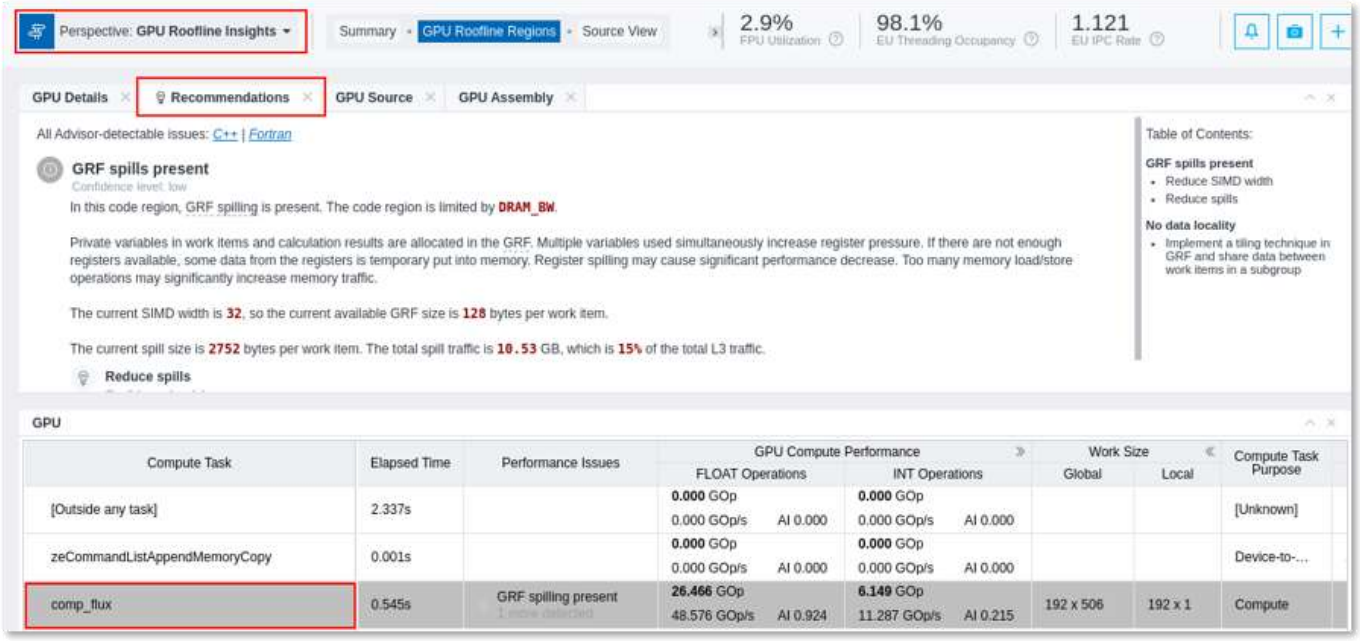
### インテル® VTune™ プロファイラーにおけるアトミック double

| Addr... ▲ | Sour... | Assembly | GPU Instructions Executed |
|---|---|---|---|
| 0x50 | | (W) shl (1\|M0) r6.0<1>:q r6.0<0;1,0>:q 3:w | 20,480 |
| 0x60 | | mov (8\|M8) r16.0<1>:q 0:w | 20,480 |
| 0x70 | | add (16\|M0) r8.0<1>:d r8.0<0;1,0>:d r1.0<16;16,1>:uw | 20,480 |
| 0x80 | | (W) add (1\|M0) r6.2<1>:d r6.2<0;1,0>:d r7.4<0;1,0>:d | 20,480 |
| 0x90 | | (W) add (1\|M0) r26.0<1>:q r6.0<0;1,0>:q r5.0<0;1,0>:q | 20,480 |
| 0xa0 | | add (16\|M0) r8.0<1>:d r8.0<8;8,1>:d r4.0<0;1,0>:d {Comp | 20,480 |
| 0xa8 | | mov (8\|M0) r10.0<1>:uq r26.0<0;1,0>:uq | 20,480 |
| 0xb8 | | mov (8\|M8) r12.0<1>:uq r26.0<0;1,0>:uq | 20,480 |
| 0xc8 | | shl (16\|M0) r8.0<1>:d r8.0<8;8,1>:d 3:w | 20,480 |
| 0xd8 | | add (16\|M0) r8.0<1>:d r6.2<0;1,0>:d r8.0<8;8,1>:d {Comp | 20,480 |
| 0xe0 | | send (16\|M0) r4:w r8 0xC 0x04405C01 [Data Cache Data Pc | 20,480 |
| 0xf0 | | sends (8\|M0) r8:uq r10 r14 0x8C 0x0424B2FF [Data Cache | 20,480 |
| 0x100 | | sends (8\|M8) r10:uq r12 r16 0x8C 0x0424B2FF [Data Cache | 20,480 |
| 0x110 | | mov (8\|M0) r23.0<2>:d r4.0<8;8,1>:d | 20,480 |
| 0x120 | | mov (8\|M8) r21.0<2>:d r5.0<8;8,1>:d | 20,480 |
| 0x130 | | mov (8\|M0) r23.1<2>:d r6.0<8;8,1>:d | 20,480 |
| 0x140 | | mov (8\|M8) r21.1<2>:d r7.0<8;8,1>:d | 20,480 |
| 0x150 | | mov (8\|M0) r19.0<1>:uq r8.0<4;4,1>:uq | 20,480 |
| 0x160 | | mov (8\|M8) r17.0<1>:uq r10.0<4;4,1>:uq | 20,480 |
| **0x170** | | **Block 2:** | |
| 0x170 | | add (8\|M0) r11.0<1>:df r23.0<4;4,1>:df r19.0<4;4,1> | 33,391,619 |
| 0x180 | | mov (8\|M0) r5.0<1>:uq r26.0<0;1,0>:uq | 33,391,619 |
| 0x190 | | mov (8\|M0) r9.0<1>:q r19.0<4;4,1>:q | 33,391,619 |
| 0x1a0 | | add (8\|M8) r15.0<1>:df r21.0<4;4,1>:df r17.0<4;4,1> | 33,391,619 |
| 0x1b0 | | mov (8\|M8) r7.0<1>:uq r26.0<0;1,0>:uq | 33,391,619 |
| 0x1c0 | | mov (8\|M8) r13.0<1>:q r17.0<4;4,1>:q | 33,391,619 |
| 0x1d0 | | sends (8\|M0) r3:uq r5 r9 0x10C 0x0424BEFF [Data Cac | 33,391,619 |
| 0x1e0 | | sends (8\|M8) r5:uq r7 r13 0x10C 0x0424BEFF [Data Ca | 33,391,619 |
| 0x1f0 | | mov (8\|M0) r7.0<1>:uq r3.0<4;4,1>:uq | 33,391,619 |
| 0x200 | | mov (8\|M8) r3.0<1>:uq r5.0<4;4,1>:uq | 33,391,619 |
| 0x210 | | cmp (8\|M0) (eq)f0.0 null<1>:q r7.0<4;4,1>:q r19.0<4 | 33,391,619 |
| 0x220 | | cmp (8\|M8) (eq)f0.0 null<1>:q r3.0<4;4,1>:q r17.0<4 | 33,391,619 |
| 0x230 | | (f0.0) break (16\|M0) bb_4 bb_4 | 33,391,619 |
| **0x240** | | **Block 3:** | |
| 0x240 | | mov (8\|M0) r19.0<1>:q r7.0<4;4,1>:q | 33,371,139 |
| 0x250 | | mov (8\|M8) r17.0<1>:q r3.0<4;4,1>:q | 33,371,139 |
| **0x260** | | **Block 4:** | |
| 0x260 | | while (16\|M0) bb_2 | 33,391,619 |
| **0x270** | | **Block 5:** | |
| 0x270 | | (W) mov (8\|M0) r112.0<1>:ud r2.0<8;8,1>:ud {Compacted} | 20,480 |
| 0x278 | | (W) s | |

VTune Profiler session timed out. All open results were closed and a new session was created. ×

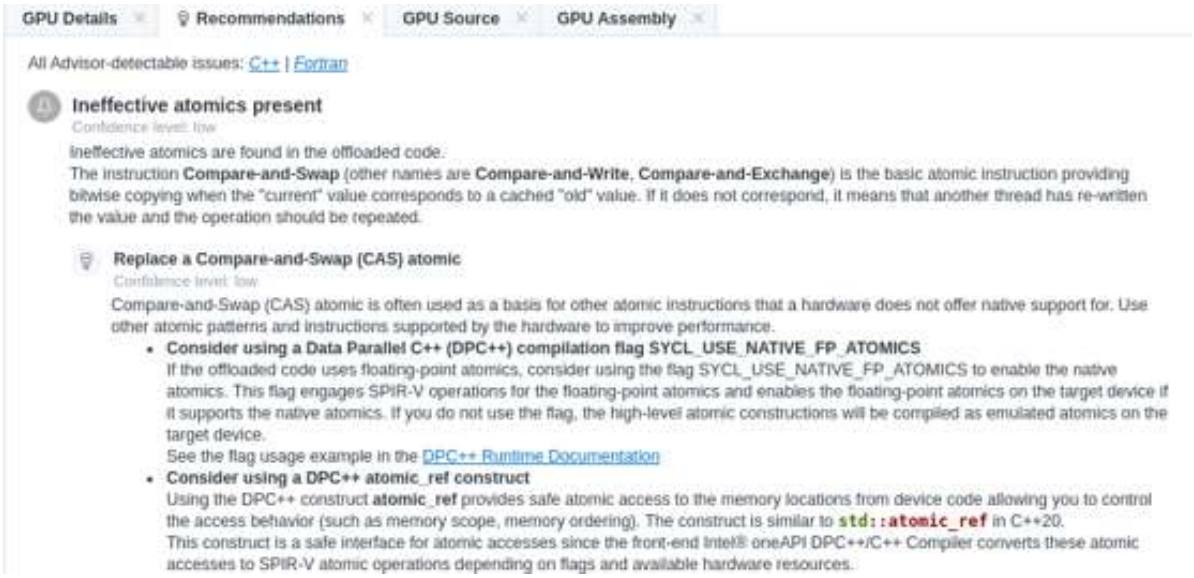
インテル® Advisor には、GPU カーネルのパフォーマンス向上の可能性を示す情報を提供する推奨ペインがあります。

### インテル® Advisor の推奨ペイン




インテル® Advisor の推奨事項の 1 つに「非効率なアトミックが存在します」という項目があります。アトミック操作がハードウェアでネイティブにサポートされていない場合、エミュレートされますが、インテル® Advisor はこれを検出可能であり、可能な解決策に関するアドバイスを提供します。

### インテル® Advisor が示す非効率なアトミック




## グローバルとローカルアドレス空間のアトミック操作

標準の C++ メモリーモデルは、アプリケーションが単一アドレス空間を持つ、単一デバイスで実行されることを前提としています。これらの仮定はどちらも SYCL* アプリケーションには当てはまりません。SYCL* では、アプリケーションの異なる領域を各種デバイス (ホストデバイスと 1 つ以上のアクセラレーター・デバイス) で実行できます。それぞれのデバイスには複数のアドレス空間 (プライベート、ローカル、グローバル) があります。また、デバイスのグローバルアドレス空間は互いに分かれている場合があります (USM サポートに依存します)。

グローバルアドレス空間でアトミックを使用する場合、グローバルの更新はローカルよりも時間がかかるため注意が必要です。

```
#include <CL/sycl.hpp>
#include <iostream>
int main() {
  constexpr int N = 256 * 256;
  constexpr int M = 512;
  int total = 0;
  int *a = static_cast<int *>(malloc(sizeof(int) * N));
  for (int i = 0; i < N; i++)
    a[i] = 1;
  sycl::queue q({sycl::property::queue::enable_profiling()});
  sycl::buffer<int> buf(&total, 1);
  sycl::buffer<int> bufa(a, N);
  auto e = q.submit([&](sycl::handler &h) {
    sycl::accessor acc(buf, h);
    sycl::accessor acc_a(bufa, h, sycl::read_only);
    h.parallel_for(sycl::nd_range<1>(N, M), [=](auto it) {
      auto i = it.get_global_id();
      sycl::atomic_ref<int, sycl::memory_order_relaxed,
      sycl::memory_scope_device,
      sycl::access::address_space::global_space>
        atomic_op(acc[0]);
      atomic_op += acc_a[i];
    });
  });
  sycl::host_accessor h_a(buf);
  std::cout << "Reduction Sum : " << h_a[0] << "\n";
  std::cout
    << "Kernel Execution Time of Global Atomics Ref: "
    << e.get_profiling_info<sycl::info::event_profiling::command_end>() -
      e.get_profiling_info<sycl::info::event_profiling::command_start>()
    << "\n";
  return 0;
}
```

次の例のように、ローカルメモリー空間を使用するようにコードを再構成することができます。

```
#include <CL/sycl.hpp>
#include <iostream>
int main() {
  constexpr int N = 256 * 256;
  constexpr int M = 512;
  constexpr int NUM_WG = N / M;
  int total = 0;
  int *a = static_cast<int *>(malloc(sizeof(int) * N));
  for (int i = 0; i < N; i++)
    a[i] = 1;
  sycl::queue q({sycl::property::queue::enable_profiling()});
  sycl::buffer<int> global(&total, 1);
  sycl::buffer<int> bufa(a, N);
  auto e1 = q.submit([&](sycl::handler &h) {
    sycl::accessor b(global, h);
    sycl::accessor acc_a(bufa, h, sycl::read_only);
    auto acc = sycl::local_accessor<int, 1>(NUM_WG, h);
    h.parallel_for(sycl::nd_range<1>(N, M), [=](auto it) {
      auto i = it.get_global_id(0);
      auto group_id = it.get_group(0);
      sycl::atomic_ref<int, sycl::memory_order_relaxed,
        sycl::memory_scope_device,
        sycl::access::address_space::local_space>
        atomic_op(acc[group_id]);
      sycl::atomic_ref<int, sycl::memory_order_relaxed,
        sycl::memory_scope_device,
        sycl::access::address_space::global_space>
        atomic_op_global(b[0]);
```

```
29.       atomic_op += acc_a[i];
30.       it.barrier(sycl::access::fence_space::local_space);
31.       if (it.get_local_id() == 0)
32.         atomic_op_global += acc[group_id];
33.     });
34.   });
35.   sycl::host_accessor h_global(global);
36.   std::cout << "Reduction Sum : " << h_global[0] << "\n";
37.   int total_time =
38.     (e1.get_profiling_info<sycl::info::event_profiling::command_end>() -
39.     e1.get_profiling_info<sycl::info::event_profiling::command_start>());
40.   std::cout << "Kernel Execution Time of Local Atomics : " << total_time
41.             << "\n";
42.   return 0;
43. }
```

## USM データ上のアトミック操作

ディスクリート GPU には以下が当てはまります。

- ホストに割り当てられた USM (`sycl::malloc_host`) のアトミック操作はサポートされていません。
- ホストとデバイスから共有 USM ロケーションへの同時アクセス (`sycl::malloc_shared`) はサポートされていません。

アトミック操作とデバイスのアルゴリズムには、デバイスに割り当てられた USM (`sycl::malloc_device`) メモリーを使用することを推奨します。

## ローカルバリアとグローバルアトミック

アトミックを使用すると、カーネル内で複数の work-item が共有リソースを操作できます。バリアは、work-group 内で work-item 間の同期を可能にします。カーネルの起動とローカルバリアを適切に使用することで、グローバルアトミック機能を実現できます。アーキテクチャーと関連するデータ量によって、どちらか一方のパフォーマンスが向上する可能性があります。

次の例では、ベクトル内の少量の要素を総計しようとしています。このタスクは個別に実行できます。以下のリストに示すカーネルは、ベクトルのすべての要素をウォークスルーして、単一の work-item のみがそれらの合計を行います。

```
1.  q.submit([&](auto &h) {
2.    sycl::accessor buf_acc(buf, h, sycl::read_only);
3.    sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
4.    h.parallel_for(data_size, [=](auto index) {
5.      int glob_id = index[0];
6.      if (glob_id == 0) {
7.        int sum = 0;
8.        for (int i = 0; i < N; i++)
9.          sum += buf_acc[i];
10.       sum_acc[0] = sum;
11.     }
12.   });
13. });
```

以下のカーネルは、すべての work-item が合計する値をグローバル変数で更新するグローバルアトミックを使用して同じ問題を解決します。これは並列に実行されますが、グローバル変数の競合が非常に多く、ほとんどの場合パフォーマンスはそれほど期待できません。

```
1.  q.submit([&](auto &h) {
2.    sycl::accessor buf_acc(buf, h, sycl::read_only);
```

```
3.    sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
4.    h.parallel_for(data_size, [=](auto index) {
5.      size_t glob_id = index[0];
6.      auto v = sycl::atomic_ref<int, sycl::memory_order::relaxed,
7.        sycl::memory_scope::device,
8.        sycl::access::address_space::global_space>(
9.        sum_acc[0]);
10.     v.fetch_add(buf_acc[glob_id]);
11.   });
12. });
```

次のカーネルは、すべての work-item がベクトルの複数の要素を合計します。この合計は並列に行われ、work-group のすべての work-item 間で共有される配列が更新されます。work-group のすべての work-item がバリアを使用してツリー・リダクションを行い、それらを同期して共有メモリーの中間結果を最終的にリダクションします。このカーネルは、明示的に 1 つの work-group を作成し、ベクトル内のすべての要素を更新する役割を work-group 内の work-item に分散します。スレッド数の観点からはマシンのリソースを使い切っていませんが、問題サイズが小さい場合はこれくらいの並列処理で十分なこともあります。

```
1.  Timer timer;
2.  q.submit([&](auto &h) {
3.    sycl::accessor buf_acc(buf, h, sycl::read_only);
4.    sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
5.    sycl::local_accessor<int, 1> scratch(work_group_size, h);
6.    h.parallel_for(sycl::nd_range<1>{work_group_size, work_group_size},
7.      [=](sycl::nd_item<1> item) {
8.        size_t loc_id = item.get_local_id(0);
9.        int sum = 0;
10.       for (int i = loc_id; i < data_size; i += num_work_items)
11.         sum += buf_acc[i];
12.         scratch[loc_id] = sum;
13.       for (int i = work_group_size / 2; i > 0; i >>= 1) {
14.         item.barrier(sycl::access::fence_space::local_space);
15.         if (loc_id < i)
16.           scratch[loc_id] += scratch[loc_id + i];
17.       }
18.       if (loc_id == 0)
19.         sum_acc[0] = scratch[0];
20.   });
21. });
```

ここに示した 3 つのカーネルのパフォーマンスはプラットフォーム間でかなり異なり、開発者はアプリケーションと実行されるハードウェアに適した手法を選択する必要があります。

## work-group サイズを選択する考慮事項

SYCL* では `nd_range` カーネルの work-group サイズを選択できます。work-group のサイズは、計算リソース、ベクトルレーン、および work-item 間の通信の利用率に影響を及ぼします。同じ work-group 内の work-item は、共有メモリーやハードウェアによる同期機能などのハードウェア・リソースを利用することがあります。これにより、work-group 間で work-item よりも効率良く通信できるようになります。これには一般的に、アクセラレーターでサポートされる最大 work-group サイズを選択する必要があります。最大 work-group サイズは、`device::get_info<cl::sycl::info::device::max_work_group_size>()` を呼び出すことで照会できます。

work-group サイズの選択の影響を明確にするため、大きなベクトル全体のすべての要素を加算するリダクション・カーネルを考えてみます。カーネルを実行する関数は、`work-group-size` と `sub-group-size` を引数として受け取り、異なる値の組み合わせで検証を行うことができるようにします。パフォーマンスの差は、さまざまな値の work-group サイズを使用してカーネルが呼び出されたときにレポートされるタイミングから確認できます。

```
1.  void reduction(sycl::queue &q, std::vector<int> &data, std::vector<int> &flush,
2.                 int iter, int work_group_size) {
3.    const size_t data_size = data.size();
4.    const size_t flush_size = flush.size();
5.    int sum = 0;
6.    const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
7.  // int vec_size =
8.  // q.get_device().get_info<sycl::info::device::native_vector_width_int>();
9.    int num_work_items = data_size / work_group_size;
10.   sycl::buffer<int> buf(data.data(), data_size, props);
11.   sycl::buffer<int> flush_buf(flush.data(), flush_size, props);
12.   sycl::buffer<int> sum_buf(&sum, 1, props);
13.   init_data(q, buf, data_size);
14.   double elapsed = 0;
15.   for (int i = 0; i < iter; i++) {
16.     q.submit([&](auto &h) {
17.       sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
18.       h.parallel_for(1, [=](auto index) { sum_acc[index] = 0; });
19.     });
20.     // キャッシュをフラッシュ
21.     q.submit([&](auto &h) {
22.       sycl::accessor flush_acc(flush_buf, h, sycl::write_only, sycl::no_init);
23.       h.parallel_for(flush_size, [=](auto index) { flush_acc[index] = 1; });
24.     });
25.     Timer timer;
26.     // reductionMapToHWVector main begin
27.     q.submit([&](auto &h) {
28.       sycl::accessor buf_acc(buf, h, sycl::read_only);
29.       sycl::local_accessor<int, 1> scratch(work_group_size, h);
30.       sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
31.       h.parallel_for(
32.         sycl::nd_range<1>(num_work_items, work_group_size),
33.         [=](sycl::nd_item<1> item) [[intel::reqd_sub_group_size(16)]] {
34.           auto v =
35.             sycl::atomic_ref<int, sycl::memory_order::relaxed,
36.             sycl::memory_scope::device,
37.             sycl::access::address_space::global_space>(
38.               sum_acc[0]);
39.               int sum = 0;
40.               int glob_id = item.get_global_id();
41.               int loc_id = item.get_local_id();
42.               for (unsigned int i = glob_id; i < data_size; i += num_work_items)
43.                 sum += buf_acc[i];
44.               scratch[loc_id] = sum;
45.               for (int i = work_group_size / 2; i > 0; i >>= 1) {
46.                 item.barrier(sycl::access::fence_space::local_space);
47.                 if (loc_id < i)
48.                   scratch[loc_id] += scratch[loc_id + i];
49.               }
50.               if (loc_id == 0)
51.                 v.fetch_add(scratch[0]);
52.       });
53.     });
54.     q.wait();
55.     elapsed += timer.Elapsed();
56.     sycl::host_accessor h_acc(sum_buf);
57.     sum = h_acc[0];
58.   }
59.   elapsed = elapsed / iter;
60.   std::string msg = "with work-groups=" + std::to_string(work_group_size);
61.   check_result(elapsed, msg, sum);
62. } // reduction の終わり
```

次のコードでは、上記のカーネルは 2 つの異なる値 (`2 * vec_size` とアクセラレーターでサポートされる最大 work-group サイズ) で呼び出されます。work-group サイズが `2* vec_size` と等しい場合、カーネルのパフォーマンスは work-group サイズが最大値である場合よりも低くなります。

```
int vec_size = 16;
int work_group_size = vec_size;
reduction(q, data, extra, 16, work_group_size);
work_group_size =
    q.get_device().get_info<sycl::info::device::max_work_group_size>();
reduction(q, data, extra, 16, work_group_size);
```

バリアやアトミックが利用されない場合、work-group サイズはパフォーマンスに影響しません。これを理解するため、アトミックやバリアがない `vec_copy` カーネルについて考えてみます。

```
1.  void vec_copy(sycl::queue &q, std::vector<int> &src, std::vector<int> &dst,
2.                std::vector<int> &flush, int iter, int work_group_size) {
3.    const size_t data_size = src.size();
4.    const size_t flush_size = flush.size();
5.    const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
6.    int num_work_items = data_size;
7.    double elapsed = 0;
8.    {
9.      sycl::buffer<int> src_buf(src.data(), data_size, props);
10.     sycl::buffer<int> dst_buf(dst.data(), data_size, props);
11.     sycl::buffer<int> flush_buf(flush.data(), flush_size, props);
12.     for (int i = 0; i < iter; i++) {
13.       // キャッシュをフラッシュ
14.       q.submit([&](auto &h) {
15.         sycl::accessor flush_acc(flush_buf, h, sycl::write_only, sycl::no_init);
16.         h.parallel_for(flush_size, [=](auto index) { flush_acc[index] = 1; });
17.       });
18.       Timer timer;
19.       q.submit([&](auto &h) {
20.         sycl::accessor src_acc(src_buf, h, sycl::read_only);
21.         sycl::accessor dst_acc(dst_buf, h, sycl::write_only, sycl::no_init);
22.         h.parallel_for(sycl::nd_range<1>(num_work_items, work_group_size),
23.           [=](sycl::nd_item<1> item) [[intel::reqd_sub_group_size(16)]] {
24.             int glob_id = item.get_global_id();
25.             dst_acc[glob_id] = src_acc[glob_id];
26.         });
27.       });
28.       q.wait();
29.       elapsed += timer.Elapsed();
30.     }
31.   }
32.   elapsed = elapsed / iter;
33.   std::string msg = "with work-group-size=" + std::to_string(work_group_size);
34.   check_result(elapsed, msg, dst);
35. } // vec_copy の終わり
```

以下のコードでは、上記のカーネルがさまざまな work-group サイズで呼び出されます。上記のカーネル呼び出しはすべて同じランタイムによって行われます。これは、work-group サイズがパフォーマンスに影響しないことを示します。これは、work-group 内で生成されたスレッドとは異なる work-group のスレッドが、work-group にバリアや共有メモリーがない場合、スケジュールとリソースの観点からは同じように動作することによります。

```
int vec_size = 16;
int work_group_size = vec_size;
vec_copy(q, src, dst, extra, 16, work_group_size);
work_group_size = 2 * vec_size;
vec_copy(q, src, dst, extra, 16, work_group_size);
work_group_size = 4 * vec_size;
vec_copy(q, src, dst, extra, 16, work_group_size);
work_group_size = 8 * vec_size;
vec_copy(q, src, dst, extra, 16, work_group_size);
work_group_size = 16 * vec_size;
vec_copy(q, src, dst, extra, 16, work_group_size);
```

一部のアクセラレーターでは、処理要素間でスレッドがスケジュールされるより、最小 sub-group サイズを適用することで良好なパフォーマンスが得られます。そのような状況では、sub-group 数が最小値より小さい場合、大きなパフォーマンスの差が見られます。上記の 3 行目のカーネル呼び出しでは 1 つの sub-group しかありませんが、5 行目の呼び出しでは 2 つの sub-group があります。同時に 2 つの sub-group のスケジュールを行うアクセラレーターでは、それら 2 つのカーネル呼び出しのタイミングによって大幅なパフォーマンスの差が生じます。

## OpenMP* オフロードモードにおけるローカルおよびグローバル work-group サイズを使用したカーネルのチューニング

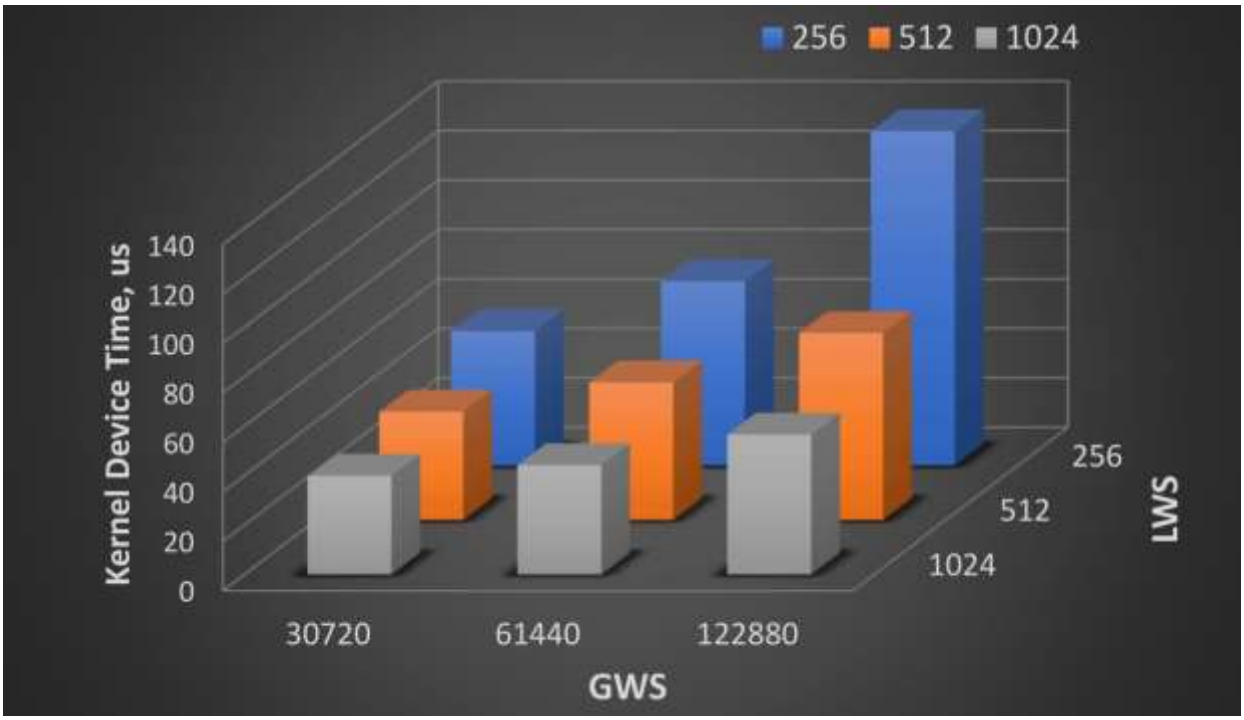
前述した SYCL* におけるアクセラレーター・デバイス向けのカーネルのパフォーマンス・チューニングに関する手法は、OpenMP* を使用した実装にも適用できます。OpenMP* ディレクティブを使用してアプリケーション・カーネルをカスタマイズすることで、適切な work-group サイズを利用できます。ただし、コードの大幅な修正が必要になる可能性があります。OpenMP* による実装では、環境変数を使用してカーネルをカスタマイズするオプションも用意されています。次に示すように、環境変数 `OMP_THREAD_LIMIT` と `OMP_NUM_TEAMS` は、ローカル・ワークグループ・サイズ (`LWS`) とグローバル・ワークグループ・サイズ (`GWS`) を設定するのに役立ちます。

```
LWS = OMP_THREAD_LIMIT
GWS = OMP_THREAD_LIMIT * OMP_NUM_TEAMS
```

以下に、`LWS` と `GWS` を使用してアクセラレーター・デバイスでリダクション・カーネルをチューニングする方法を紹介します。

```
1.  int N = 2048;
2.  double* A = make_array(N, 0.8);
3.  double* B = make_array(N, 0.65);
4.  double* C = make_array(N*N, 2.5);
5.  if ((A == NULL) || (B == NULL) || (C == NULL))
6.    exit(1);
7.  int i, j;
8.  double val = 0.0;
9.  #pragma omp target map(to:A[0:N],B[0:N],C[0:N*N]) map(tofrom:val)
10. {
11. #pragma omp teams distribute parallel for collapse(2) reduction(+ : val)
12.   for (i = 0; i < N; i++) {
13.     for (j = 0; j < N; j++) {
14.       val += C[i * N + j] * A[i] * B[j];
15.     }
16.   }
17. }
18. printf("val = %f10.3\n", val);
19. free(A);
20. free(B);
21. free(C);
```

例: `OMP_THREAD_LIMIT = 1024`、`OMP_NUM_TEAMS = 120` に設定すると、`LWS` と `GWS` パラメーターがそれぞれ `1024` と `122880` に設定されます。




上図は、`LWS = 1024`、`GWS = 30720` で、このカーネルの最高のパフォーマンスが得られることを示しています。これは、`OMP_THREAD_LIMIT = 1024`、`OMP_NUM_TEAMS = 30` に相当します。これらの環境変数は、OpenMP* でオフロードされるすべてのカーネルに対して、`LWS` と `GWS` の値を固定値に設定します。ただし、これらの環境変数は、oneMKL のような高度に調整されたライブラリー・カーネルで使用される `LWS` および `GWS` には影響しません。

## プリフェッチ

グローバルメモリーへのアクセスは、実行ストールの最も一般的な原因の 1 つです。アプリケーションがストールして、データがローカルレジスターに到達するのを待機する遅延をレイテンシーと呼びます。これは、アクセスがキャッシュをミスして、HBM メモリーにアクセスする必要がある場合は特にコストがかかります。GPU は実行する命令の順番を変更し (アウトオブオーダー実行)、ベクトルエンジンでスレッドを切り替えることでメモリーのレイテンシーを隠匿しようとします。これが十分に機能せず、アプリケーションがストールし続け、メモリー帯域幅が十分に活用されない場合、プリフェッチ機能を使用して使用される前にデータをローカルキャッシュにロードするタイミングのヒントを与える場合があります。これを適切に使用すると、以降のメモリーアクセスがキャッシュにヒットし、データがレジスターに到達する距離が短縮され、メモリー待機の時間が短縮されます。プリフェッチは、インテル® データセンター GPU マックス・シリーズ以降の製品で機能します。

プリフェッチは非同期操作であり、プリフェッチを送信するとアプリケーションはデータの到達を待機せず実行を続けます。これは、予期されるアクセスよりもかなり前にプリフェッチを送信する必要があることを意味します。HBM メモリーからローカルキャッシュの間には、データ転送を隠匿するのに十分な命令が存在します。

プリフェッチの一般的な使用例として、ループ内のメモリーアクセスがあります。通常、コンパイラーはループ展開の最適化を試みますが、これによりスケジューラーに実行する命令が与えられ、ループ反復が複合されプロセス内のメモリー・レイテンシーが隠匿されます。この方法を適用できない場合 (ループ境界が不明、大規模なカーネルの高いレジスター・プレッシャー)、プリフェッチを使用することで、以降の反復向けにデータをキャッシュすることでメモリー待機時間を短縮できます。

インテル® VTune™ プロファイラーを使用して、プリフェッチを適用する可能性を調査できます。メモリー帯域幅が十分に活用されておらず、スコアボード・ストール数が多い場合 (データがロードされるまで実行がストールしている)、プリフェッチを適切に挿入することで実行速度が向上する可能性があります。

このセクションの例では、GPU がレイテンシーを隠匿できない場合をシミュレートするシナリオを考えます。最初に work-group を 1 つだけ実行し、ハードウェア・スレッドを 1 つのベクトルエンジン (VE) に送信します。次に、pragma を使用してループ展開を無効にします。これにより、メモリーの待機時間を隠匿するため、再スケジュールできる命令数が制限されます。

最初の例では、カーネルは配列の `values` をレデュースして結果を `results` に保存します。`values` は間接インデックスを使用し、インデックスは配列 `indices` からロードされます。インデックスはコンパイル時に不明であるため、コンパイラーは配列 `values` へのアクセスを最適化できません。ループのアンロールが無効化されているため、GPU は `indices` と `values` にアクセスする間にほかのワークをスケジュールすることができず、`indices` へのアクセスが終了するまで実行はストールします。

```
  auto e = q.submit([&](auto &h) {
    h.parallel_for(
        sycl::nd_range(sycl::range{GLOBAL_WORK_SIZE},
                       sycl::range{GLOBAL_WORK_SIZE}),
        [=](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(SIMD_SIZE)]] {
          const int i = it.get_global_linear_id();
          const int lane = it.get_sub_group().get_local_id()[0];
          const int subgroup = it.get_sub_group().get_group_id()[0];

          // インデックスの開始位置
          int *indexCurrent = indices + lane + subgroup * ITEMS_PER_SUBGROUP;

          float dx = 0.0f;
#pragma unroll(0)
          for (int j = 0; j < ITEMS_PER_LANE; ++j) {
            // 間接アドレス指定のためにインデックスをロード
            int index = *indexCurrent;
            // 完了するまでロードを待機、高レイテンシー
            float v = values[index];
            for (int k = 0; k < 64; ++k)
              dx += sqrtf(v + k);

            indexCurrent += SIMD_SIZE;
          }

          result[i] = dx;
        });
  });
```

2 番目の例では、`indices` から現在の値にアクセスする前に、カーネルは `sycl::global_ptr::prefetch` を呼び出して次のインデックスのプリフェッチを要求しているため、次の反復でのアクセスが高速になる可能性があります。レイテンシーを隠匿するのに十分な数の命令がないため、最初の反復ではデータはプリフェッチされません。

```
  auto e = q.submit([&](auto &h) {
    h.parallel_for(
        sycl::nd_range(sycl::range{GLOBAL_WORK_SIZE},
                       sycl::range{GLOBAL_WORK_SIZE}),
        [=](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(SIMD_SIZE)]] {
          const int i = it.get_global_linear_id();
          const int lane = it.get_sub_group().get_local_id()[0];
          const int subgroup = it.get_sub_group().get_group_id()[0];

          // インデックスの開始位置
          int *indexCurrent = indices + lane + subgroup * ITEMS_PER_SUBGROUP;
```

```
          int *indexNext = indexCurrent + SIMD_SIZE;

          float dx = 0.0f;
#pragma unroll(0)
          for (int j = 0; j < ITEMS_PER_LANE; ++j) {
            // キャッシュする次のインデックスをプリフェッチ
            sycl::global_ptr<int>(indexNext).prefetch(1);
            // Load index, might be cached
            int index = *indexCurrent;
            // インデックスがキャッシュされた場合レイテンシーは軽減される
            float v = values[index];
            for (int k = 0; k < 64; ++k)
              dx += sqrtf(v + k);

            indexCurrent = indexNext;
            indexNext += SIMD_SIZE;
          }

          result[i] = dx;
        });
  });
```

`sycl::global_ptr::prefetch` は、プリフェッチする連続する要素数を示す引数を 1 つ受け取ります。プリフェッチはグローバルアドレス空間でのみ使用でき、この関数には考慮すべきいくつかの制限事項があります。

1. コンパイラーは、各プリフェッチ呼び出しに対し、(大きなデータタイプに対しても) work-item あたり最大 32 バイトをキャッシュします。
2. コンパイラーには、コンパイル時に引数 (プリフェッチする連続する要素数) が定数であることが最適です。
3. どのキャッシュのデータをプリフェッチするかは制御できません。現在の実装では、データは L3 キャッシュにのみロードされます。

キャッシュの制御が十分でないと、`sycl::global_ptr::prefetch` によってパフォーマンスが向上する可能性が制限されます。2 番目の例を実行すると、パフォーマンスが低下する場合もあります。データが L1 キャッシュにロードされる場合、プリフェッチは最良の結果をもたらします。現在、インテル® データセンター GPU マックス・シリーズ専用の機能でのみ可能です。まず、これらの宣言をコードに追加します。

```
#include <CL/sycl.hpp>

enum LSC_LDCC {
  LSC_LDCC_DEFAULT,
  LSC_LDCC_L1UC_L3UC, // 1 // Override to L1 uncached and L3 uncached
  LSC_LDCC_L1UC_L3C,  // 2 // Override to L1 uncached and L3 cached
  LSC_LDCC_L1C_L3UC,  // 3 // Override to L1 cached and L3 uncached
  LSC_LDCC_L1C_L3C,   // 4 // Override to L1 cached and L3 cached
  LSC_LDCC_L1S_L3UC,  // 5 // Override to L1 streaming load and L3 uncached
  LSC_LDCC_L1S_L3C,   // 6 // Override to L1 streaming load and L3 cached
  LSC_LDCC_L1IAR_L3C, // 7 // Override to L1 invalidate-after-read, and L3
                      // cached
};

extern "C" {
SYCL_EXTERNAL void
__builtin_IB_lsc_prefetch_global_uchar(const __attribute__((opencl_global))
                                       uint8_t *base,
                                       int immElemOff, enum LSC_LDCC cacheOpt);
SYCL_EXTERNAL void
__builtin_IB_lsc_prefetch_global_ushort(const __attribute__((opencl_global))
                                        uint16_t *base,
                                        int immElemOff, enum LSC_LDCC cacheOpt);
SYCL_EXTERNAL void
__builtin_IB_lsc_prefetch_global_uint(const __attribute__((opencl_global))
                                      uint32_t *base,
```

```
                                          int immElemOff, enum LSC_LDCC cacheOpt);
SYCL_EXTERNAL void
__builtin_IB_lsc_prefetch_global_uint2(const __attribute__((opencl_global))
                                       uint32_t *base,
                                       int immElemOff, enum LSC_LDCC cacheOpt);
SYCL_EXTERNAL void
__builtin_IB_lsc_prefetch_global_uint3(const __attribute__((opencl_global))
                                       uint32_t *base,
                                       int immElemOff, enum LSC_LDCC cacheOpt);
SYCL_EXTERNAL void
__builtin_IB_lsc_prefetch_global_uint4(const __attribute__((opencl_global))
                                       uint32_t *base,
                                       int immElemOff, enum LSC_LDCC cacheOpt);
SYCL_EXTERNAL void
__builtin_IB_lsc_prefetch_global_uint8(const __attribute__((opencl_global))
                                       uint32_t *base,
                                       int immElemOff, enum LSC_LDCC cacheOpt);
SYCL_EXTERNAL void
__builtin_IB_lsc_prefetch_global_ulong(const __attribute__((opencl_global))
                                       uint64_t *base,
                                       int immElemOff, enum LSC_LDCC cacheOpt);
SYCL_EXTERNAL void
__builtin_IB_lsc_prefetch_global_ulong2(const __attribute__((opencl_global))
                                        uint64_t *base,
                                        int immElemOff, enum LSC_LDCC cacheOpt);
SYCL_EXTERNAL void
__builtin_IB_lsc_prefetch_global_ulong3(const __attribute__((opencl_global))
                                        uint64_t *base,
                                        int immElemOff, enum LSC_LDCC cacheOpt);
SYCL_EXTERNAL void
__builtin_IB_lsc_prefetch_global_ulong4(const __attribute__((opencl_global))
                                        uint64_t *base,
                                        int immElemOff, enum LSC_LDCC cacheOpt);
SYCL_EXTERNAL void
__builtin_IB_lsc_prefetch_global_ulong8(const __attribute__((opencl_global))
                                        uint64_t *base,
                                        int immElemOff, enum LSC_LDCC cacheOpt);
}
```

引数 `cacheOpt` で対象のキャッシュを選択します。`sycl::global_ptr::prefetch` は `LSC_LDCC_L1UC_L3C` にマップされます。ほとんどの場合、`LSC_LDCC_L1UC_L3C` は最良の結果をもたらします。引数 `immElemOff` でベースポインターをオフセットできますが、通常ベースポインターはフェッチする適切なアドレスに設定され `immElemOff` は 0 になります。

L1 へのプリフェッチは、安全な境界外へのアクセスをサポートしません。このタイプのプリフェッチを行う場合、アプリケーションは境界チェックを行う安全な所有権を取得する必要があります。そうしないと、境界外へのアクセスは未定義の動作を引き起こします。

3 番目の例は、境界チェックを含む L1 プリフェッチの動作を示します。

```
  auto e = q.submit([&](auto &h) {
    h.parallel_for(
        sycl::nd_range(sycl::range{GLOBAL_WORK_SIZE},
                       sycl::range{GLOBAL_WORK_SIZE}),
        [=](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(SIMD_SIZE)]] {
          const int i = it.get_global_linear_id();
          const int lane = it.get_sub_group().get_local_id()[0];
          const int subgroup = it.get_sub_group().get_group_id()[0];

          // インデックスの開始位置
          int *indexCurrent = indices + lane + subgroup * ITEMS_PER_SUBGROUP;
          int *indexNext = indexCurrent + SIMD_SIZE;

          float dx = 0.0f;
```

```
#pragma unroll(0)
          for (int j = 0; j < ITEMS_PER_LANE; ++j) {
        // キャッシュする次のインデックスをプリフェッチ
#if __SYCL_DEVICE_ONLY__
            if (j < ITEMS_PER_LANE - 1)
              __builtin_IB_lsc_prefetch_global_uint(
                  (const __attribute__((opencl_global)) uint32_t *)indexNext, 0,
                  LSC_LDCC_L1C_L3C);
#endif
            // キャッシュされるインデックスをロード
            int index = *indexCurrent;
            // インデックスがキャッシュされた場合レイテンシーは軽減される
            float v = values[index];
            for (int k = 0; k < 64; ++k)
              dx += sqrtf(v + k);

            indexCurrent = indexNext;
            indexNext += SIMD_SIZE;
          }

          result[i] = dx;
        });
  });
```

データを手動でキャッシュにプリフェッチする場合、メモリーフェンスによってキャッシュがフラッシュされる可能性があるため、キャッシュに再度データを配置するためプリフェッチを繰り返す必要があることに注意してください。

ここで示した例は、プリフェッチの使用法に関するアイディアを提供するだけであり、実際の使い方を表すものではありません。このようなカーネルは、ループのアンロールのような通常の最適化手法で、メモリー・レイテンシーを隠匿するには十分です。プリフェッチのヒントは、コンパイラーによって適用される通常の最適化では不十分な大規模なカーネルでパフォーマンスを向上することが期待されます。

## リダクション

リダクションは、配列のすべての要素に演算子を適用し、単一の結果を生成する並列プログラミングにおける一般的な操作です。リダクション演算子は結合可能であり、場合によっては可換です。リダクション演算子には、合計、最大、最小などがあります。以下にシリアル・リダクション合計の例を示します。

```
for (int it = 0; it < iter; it++) {
  sum = 0;
  for (size_t i = 0; i < data_size; ++i) {
    sum += data[i];
  }
}
```

リダクションの時間あたりの計算量は、要素数に比例します。これを並列化するにはいくつかの方法がありますが、異なる処理要素間の通信/同期を最小限にするように注意する必要があります。リダクションを並列化する最も単純な方法は、グローバル変数を使用して、スレッドがアトミック操作でこの変数を更新することです。

```
q.submit([&](auto &h) {
  sycl::accessor buf_acc(buf, h, sycl::read_only);
  sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

  h.parallel_for(data_size, [=](auto index) {
    size_t glob_id = index[0];
    auto v = sycl::atomic_ref<int, sycl::memory_order::relaxed,
                              sycl::memory_scope::device,
                              sycl::access::address_space::global_space>(
      sum_acc[0]);
```

```
    v.fetch_add(buf_acc[glob_id]);
});
```

スレッドが同一のメモリー位置をアトミックに更新して大量の競合が発生するため、このカーネルのパフォーマンスは低下します。より良いアプローチは、配列を小さなチャンクに分割し、それぞれのスレッドが各チャンクのローカル合計を計算してから、ローカル合計をシーケンシャルなツリー・リダクションを行うことです。チャンク数はプラットフォームの処理要素数によって異なります。処理要素数は、デバイス・オブジェクトの `get_info<info::device::max_compute_units>()` 関数で照会できます。

```
q.submit([&](auto &h) {
  sycl::accessor buf_acc(buf, h, sycl::read_only);
  sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
  h.parallel_for(num_processing_elements, [=](auto index) {
    size_t glob_id = index[0];
    size_t start = glob_id * BATCH;
    size_t end = (glob_id + 1) * BATCH;
    if (end > N)
      end = N;
    int sum = 0;
    for (size_t i = start; i < end; ++i)
      sum += buf_acc[i];
    accum_acc[glob_id] = sum;
  });
});
```

このカーネルは、共有メモリーをアトミックに更新するカーネルよりも高いパフォーマンスを示します。ただし、コンパイラーがループをベクトル化できないため、まだ十分とは言えません。コンパイラーがベクトルコードを生成できるようにする 1 つの方法は、次のようにループを変更することです。

```
q.submit([&](auto &h) {
  sycl::accessor buf_acc(buf, h, sycl::read_only);
  sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
  h.parallel_for(num_work_items, [=](auto index) {
    size_t glob_id = index[0];
    int sum = 0;
    for (size_t i = glob_id; i < data_size; i += num_work_items)
      sum += buf_acc[i];
    accum_acc[glob_id] = sum;
  });
});
```

コンパイラーはこのコードをベクトル化し、パフォーマンスを改善できます。

GPU では、マシン上の物理プロセッサー (ベクトルエンジン (VE) またはエグゼキューション・ユニット (EU) と呼ばれる) ごとに多くのスレッド・コンテキストを利用できます。そのため、スレッド数が VE 数と等しい上記のコードは、すべてのスレッド・コンテキストを利用できるわけではなく、コアごとに 2 つの論理スレッドを持つ CPU でも、コードがすべてのスレッド・コンテキストを使用できるわけではありません。一般的に、すべてのスレッド・コンテキストを完全に利用するには、ワークを十分な数の work-group に分割します。これにより、コードは長いレイテンシーの命令を許容できるようになります。次の表は、各種デバイスの処理要素ごとに利用可能なスレッド・コンテキストの数を示しています。

**デバイスで利用可能なスレッド・コンテキスト数**

| | VE 数 | VE ごとのスレッド数 \| 合計スレッド数 |
|---|---|---|
| KBL† | 24 | 7 \| 24 x 7 = 168 |
| TGL† | 96 | 7 \| 96 x 7 = 672 |

次のコードは、使用可能なリソースを完全に利用するため、十分なスレッドを持つカーネルを記述する方法を示しています。デバイスで利用可能なスレッド・コンテキストの数を照会する方法がないことに注意してください。そのため、デバイスに応じてワークを分割するため生成する work-item の数をスケーリングするようにします。

```
q.submit([&](auto &h) {
  sycl::accessor buf_acc(buf, h, sycl::read_only);
  sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
  h.parallel_for(num_work_items, [=](auto index) {
    size_t glob_id = index[0];
    int sum = 0;
    for (size_t i = glob_id; i < data_size; i += num_work_items)
      sum += buf_acc[i];
      accum_acc[glob_id] = sum;
  });
});
```

GPU でリダクションを行う一般的な手段として、多くの work-group を生成して、それぞれの work-group でツリー・リダクションを実行する方法があります。以下のカーネルでは、work-group 内の各 work-item がリダクションのネットワークに参加して、その work-group 内のすべての要素を合計しています。次に、work-group からのすべての中間結果のシリアル・リダクションを実行して合計します (中間結果が十分に大きい場合、ツリー・リダクションをさらに数回実行できます)。このツリー・リダクションのアルゴリズムは、work-group 内の work-item 間で高速な同期操作を利用します。このカーネルのパフォーマンスは、大量のカーネルが起動されるため、カーネルの起動効率に大きく依存します。また、実際にワークを実行するスレッド数はループを通過するごとに指数関数的に減少するため、以下のカーネルはそれほど効率的ではありません。

```
q.submit([&](auto &h) {
  sycl::accessor buf_acc(buf, h, sycl::read_only);
  sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
  sycl::local_accessor<int, 1> scratch(work_group_size, h);

  h.parallel_for(sycl::nd_range<1>(num_work_items, work_group_size),
    [=](sycl::nd_item<1> item) {
      size_t global_id = item.get_global_id(0);
      int local_id = item.get_local_id(0);
      int group_id = item.get_group(0);
      if (global_id < data_size)
        scratch[local_id] = buf_acc[global_id];
      else
        scratch[local_id] = 0;
      // work-group 内の項目でツリー・リダクションを行います
      for (int i = work_group_size / 2; i > 0; i >>= 1) {
        item.barrier(sycl::access::fence_space::local_space);
        if (local_id < i)
          scratch[local_id] += scratch[local_id + i];
      }

      if (local_id == 0)
        accum_acc[group_id] = scratch[0];
    });
});
```

単一ステージのリダクションは、ホストに多くのワークが残るため、あまり効率良いとは言えません。ステージを追加すると、ホストのワークが減り、パフォーマンスが大幅に向上します。次のカーネルでは、ステージ 1 で計算された中間結果がステージ 2 の入力であることが分かります。これを一般化して、結果がホストで実行できるよう小さくなるまで、複数ステージのリダクションを構成します。

```
q.submit([&](auto &h) {
  sycl::accessor buf_acc(buf, h, sycl::read_only);
  sycl::accessor accum_acc(accum1_buf, h, sycl::write_only, sycl::no_init);
  sycl::local_accessor<int, 1> scratch(work_group_size, h);
```

```
  h.parallel_for(sycl::nd_range<1>(num_work_items1, work_group_size),
    [=](sycl::nd_item<1> item) {
      size_t global_id = item.get_global_id(0);
      int local_id = item.get_local_id(0);
      int group_id = item.get_group(0);

      if (global_id < data_size)
        scratch[local_id] = buf_acc[global_id];
      else
        scratch[local_id] = 0;

      // work-group 内の項目でツリー・リダクションを行います
      for (int i = work_group_size / 2; i > 0; i >>= 1) {
        item.barrier(sycl::access::fence_space::local_space);
        if (local_id < i)
          scratch[local_id] += scratch[local_id + i];
        }
        if (local_id == 0)
          accum_acc[group_id] = scratch[0];
      });
});
q.submit([&](auto &h) {
  sycl::accessor buf_acc(accum1_buf, h, sycl::read_only);
  sycl::accessor accum_acc(accum2_buf, h, sycl::write_only, sycl::no_init);
  sycl::local_accessor<int, 1> scratch(work_group_size, h);

  h.parallel_for(sycl::nd_range<1>(num_work_items2, work_group_size),
    [=](sycl::nd_item<1> item) {
      size_t global_id = item.get_global_id(0);
      int local_id = item.get_local_id(0);
      int group_id = item.get_group(0);

      if (global_id < static_cast<size_t>(num_work_items2))
        scratch[local_id] = buf_acc[global_id];
      else
        scratch[local_id] = 0;

      // work-group 内の項目でツリー・リダクションを行います
      for (int i = work_group_size / 2; i > 0; i >>= 1) {
        item.barrier(sycl::access::fence_space::local_space);
        if (local_id < i)
          scratch[local_id] += scratch[local_id + i];
      }

      if (local_id == 0)
        accum_acc[group_id] = scratch[0];
      });
});
```

SYCL* はビルトインのリダクション操作をサポートしており、その実装はベースとなるアーキテクチャーに合わせて調整されるため、を使用する必要応じて SYCL* のリダクションを使用してください。次のカーネルは、コンパイラーがビルトイン・リダクション操作を使用する方法を示します。

```
q.submit([&](auto &h) {
  sycl::accessor buf_acc(buf, h, sycl::read_only);
  auto sumr = sycl::reduction(sum_buf, h, sycl::plus<>());
  h.parallel_for(sycl::nd_range<1>{data_size, 256}, sumr,
    [=](sycl::nd_item<1> item, auto &sumr_arg) {
      int glob_id = item.get_global_id(0);
      sumr_arg += buf_acc[glob_id];
    });
});
```

次の最適化として、入力ベクトルのアクセスをブロック化し、共有ローカルメモリーに中間結果を格納することが考えられます。このカーネルを以下に示します。ここでは、すべての work-item が特定数のベクトル要素を操作し、work-group 内の 1 つのスレッドが中間結果を含む共有メモリーをリニアに処理することで、すべての要素を 1 つの結果に集約できます。

```
q.submit([&](auto &h) {
  sycl::accessor buf_acc(buf, h, sycl::read_only);
  sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
   sycl::local_accessor<int, 1> scratch(work_group_size, h);
   h.parallel_for(sycl::nd_range<1>{num_work_items, work_group_size},
     [=](sycl::nd_item<1> item) {
      size_t glob_id = item.get_global_id(0);
      size_t group_id = item.get_group(0);
      size_t loc_id = item.get_local_id(0);
      int offset = ((glob_id >> log2workitems_per_block)
        << log2elements_per_block) +
        (glob_id & mask); int sum = 0;
      for (int i = 0; i < elements_per_work_item; ++i)
        sum +=
          buf_acc[(i << log2workitems_per_block) + offset];
      scratch[loc_id] = sum;
      // シリアル・リダクション
      item.barrier(sycl::access::fence_space::local_space);
      if (loc_id == 0) {
        int sum = 0;
        for (int i = 0; i < work_group_size; ++i)
          sum += scratch[i];
        accum_acc[group_id] = sum;
      }
    });
});
```

以下のカーネルは、ツリー・リダクションにより work-group 内のすべての work-item から中間結果をリダクションすることを除き、上記のカーネルと同じです。ほとんどの場合、これによるパフォーマンスの影響はありません。

```
q.submit([&](auto &h) {
  sycl::accessor buf_acc(buf, h, sycl::read_only);
  sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
  sycl::local_accessor<int, 1> scratch(work_group_size, h);
  h.parallel_for(sycl::nd_range<1>{num_work_items, work_group_size},
    [=](sycl::nd_item<1> item) {
      size_t glob_id = item.get_global_id(0);
      size_t group_id = item.get_group(0);
      size_t loc_id = item.get_local_id(0);
      int offset = ((glob_id >> log2workitems_per_block)
        << log2elements_per_block) +
        (glob_id & mask);
      int sum = 0;
      for (int i = 0; i < elements_per_work_item; ++i)
        sum +=
          buf_acc[(i << log2workitems_per_block) + offset];
      scratch[loc_id] = sum;
      // ツリー・リダクション
      item.barrier(sycl::access::fence_space::local_space);
      for (int i = work_group_size / 2; i > 0; i >>= 1) {
        item.barrier(sycl::access::fence_space::local_space);
        if (loc_id < static_cast<size_t>(i))
          scratch[loc_id] += scratch[loc_id + i];
      }
      if (loc_id == 0)
        accum_acc[group_id] = scratch[0];
    });
});
```

次のカーネルは、ブロック化を適用してからコンパイラーのリダクション演算子で最終的なリダクションを行います。これは、テストを行ったほとんどのプラットフォームで良好なパフォーマンスを得られました。

```
q.submit([&](auto &h) {
  sycl::accessor buf_acc(buf, h, sycl::read_only);
  auto sumr = sycl::reduction(sum_buf, h, sycl::plus<>());
  h.parallel_for(sycl::nd_range<1>{num_work_items, work_group_size}, sumr,
    [=](sycl::nd_item<1> item, auto &sumr_arg) {
      size_t glob_id = item.get_global_id(0);
      int offset = ((glob_id >> log2workitems_per_block)
        << log2elements_per_block) +
        (glob_id & mask);
      int sum = 0;
      for (int i = 0; i < elements_per_work_item; ++i)
        sum +=
          buf_acc[(i << log2workitems_per_block) + offset];
      sumr_arg += sum;
    });
});
```

次のカーネルは、完全に異なるメモリーアクセス手法を使用します。中間結果をベクトル形式で生成するため、sub-group のロードを使用します。次に、この中間結果をホストに戻し、ホスト上で最終的なリダクションが行われます。別のカーネルを生成してこの結果を 1 つの work-group にまとめ、効率良いバリアを介してツリー・リダクションを行うことが適しているケースもあります。

```
q.submit([&](auto &h) {
  const sycl::accessor buf_acc(buf, h);
  sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
  sycl::local_accessor<sycl::vec<int, 8>, 1l> scratch(work_group_size, h);
  h.parallel_for(
    sycl::nd_range<1>{num_work_items, work_group_size},
    [=](sycl::nd_item<1> item) [[intel::reqd_sub_group_size(16)]] {
      size_t group_id = item.get_group(0);
      size_t loc_id = item.get_local_id(0);
      sycl::sub_group sg = item.get_sub_group();
      sycl::vec<int, 8> sum{0, 0, 0, 0, 0, 0, 0, 0};
      using global_ptr =
        sycl::multi_ptr<int, sycl::access::address_space::global_space>;
      int base = (group_id * work_group_size +
            sg.get_group_id()[0] * sg.get_local_range()[0]) *
            elements_per_work_item;
      for (int i = 0; i < elements_per_work_item / 8; ++i)
        sum += sg.load<8>(global_ptr(&buf_acc[base + i * 128]));
      scratch[loc_id] = sum;
      for (int i = work_group_size / 2; i > 0; i >>= 1) {
        item.barrier(sycl::access::fence_space::local_space);
        if (loc_id < static_cast<size_t>(i))
          scratch[loc_id] += scratch[loc_id + i];
      }
      if (loc_id == 0)
        accum_acc[group_id] = scratch[0];
    });
});
```

ここでは、アクセラレーターのアーキテクチャーに応じて異なるパフォーマンス特性を持つ、リダクション操作の各種実装方法を説明します。ここで注意すべきもう 1 つの重要な点は、PCIe* インターフェイス (ディスクリート GPU の場合) を経由するホストへのリダクションにかかる時間は、デバイスで実際にリダクション全体を実行するのとほぼ同じであるということです。これは、ホストとデバイス間のデータ転送を可能な限り避けるか、カーネルの実行をデータ転送とオーバーラップさせる必要があることを示しています。

## カーネルの起動

SYCL* では、特定デバイスのキューにカーネルを送信することでワークが実行されます。カーネルはホストからデバイスに送信され、デバイスで実行された結果が返されます。ホストからのカーネル送信と実行開始にはタイムラグがあります。これらは同期的であるため、カーネルに関連付けられる次のタイミングを追跡する必要があります。

- **カーネル送信開始時間**
  これは、ホストがカーネル送信処理を開始する時間です。

- **カーネル送信終了時間**
  これは、ホストがカーネル送信を終了する時間です。ホストは、引数のキューイングやカーネルがデバイスで実行を開始する際のランタイムリソース割り当てなど、複数のタスクを実行します。

- **カーネル起動時間**
  これは、ホストが送信したカーネルがデバイスで実行を開始する時間です。カーネル送信終了時間と同一でないことに注意してください。カーネル送信終了時間と起動時間には、デバイスの可用性に応じて遅延が生じます。カーネルがデバイスで起動される前に、ホストは多数のカーネルをキューに投入する可能性があります。さらに、カーネルが実行を開始する前にデータ転送を完了しなければならないことがあります。

- **カーネル完了時間**
  これは、カーネルがデバイス上で実行を終了する時間です。現世代のデバイスはプリエンプティブではありません。つまり、カーネルを起動したらその実行を完了する必要があります。

インテル® VTune™ プロファイラー (`vtune`)、`clIntercept` および `onetrace` などのツールは、アプリケーション内のすべてのカーネルに関する上記の時間の視覚的なタイムラインを表示できます。

次の簡単な例では、カーネルの実行時間を測定しています。これには、ホストでのカーネル送信時間、デバイスでのカーネル実行時間、およびデータ転送時間 (ここではバッファーやメモリーがないためゼロです) が含まれます。

```
void emptyKernel1(sycl::queue &q) {
  Timer timer;
  for (int i = 0; i < iters; ++i)
    q.parallel_for(1, [=](auto) {
      /* NOP */
    }).wait();
    std::cout << "  emptyKernel1: Elapsed time: " << timer.Elapsed() / iters
          << " sec\n";
} // emptyKernel1 の終わり
```

`parallel_for` の終わりで待機しない次のコードでは、ホストがカーネルをランタイムに送信するのにかかる時間を測定します。

```
void emptyKernel2(sycl::queue &q) {
  Timer timer;
  for (int i = 0; i < iters; ++i)
    q.parallel_for(1, [=](auto) {
      /* NOP */
    });
    std::cout << "  emptyKernel2: Elapsed time: " << timer.Elapsed() / iters
          << " sec\n";
}
```

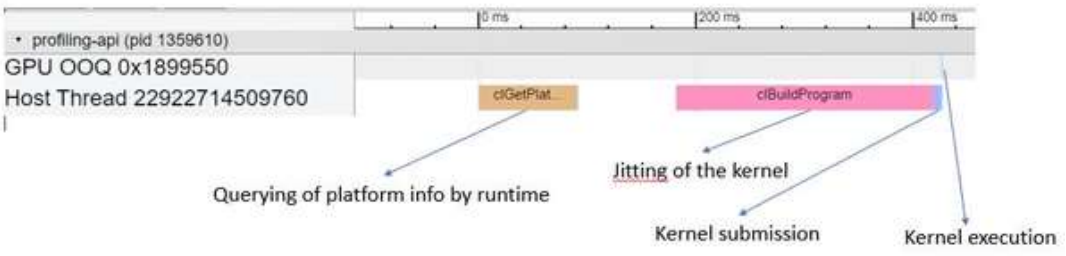
オーバーヘッドは、使用するバックエンド・ランタイムとホストの処理能力に大きく依存します。

デバイスでの実際のカーネル実行時間を測定するには、SYCL* ビルトイン・プロファイル API を使用します。次のコードは、カーネルの実行時間をプロファイルする SYCL* プロファイル API の使い方を示します。カーネルの送信時間も表示できます。カーネルの起動時間はランタイムとデバイスドライバーに依存するため、プログラムで測定する方法がありません。これを理解するにはプロファイル・ツールを使用します。

```
#include <CL/sycl.hpp>

class Timer {
public:
  Timer() : start_(std::chrono::steady_clock::now()) {}

  double Elapsed() {
    auto now = std::chrono::steady_clock::now();
    return std::chrono::duration_cast<Duration>(now - start_).count();
  }

private:
  using Duration = std::chrono::duration<double>;
  std::chrono::steady_clock::time_point start_;
};

int main() {
  Timer timer;
  sycl::queue q{sycl::property::queue::enable_profiling()};
  auto evt = q.parallel_for(1000, [=](auto) {
    /* カーネルをここに記述 */
  });
  double t1 = timer.Elapsed();
  evt.wait();
  double t2 = timer.Elapsed();
  auto startK =
    evt.get_profiling_info<sycl::info::event_profiling::command_start>();
  auto endK =
    evt.get_profiling_info<sycl::info::event_profiling::command_end>();
  std::cout << "Kernel submission time: " << t1 << "secs\n";
  std::cout << "Kernel submission + execution time: " << t2 << "secs\n";
  std::cout << "Kernel execution time: "
        << ((double)(endK - startK)) / 1000000.0 << "secs\n";

   return 0;
}
```

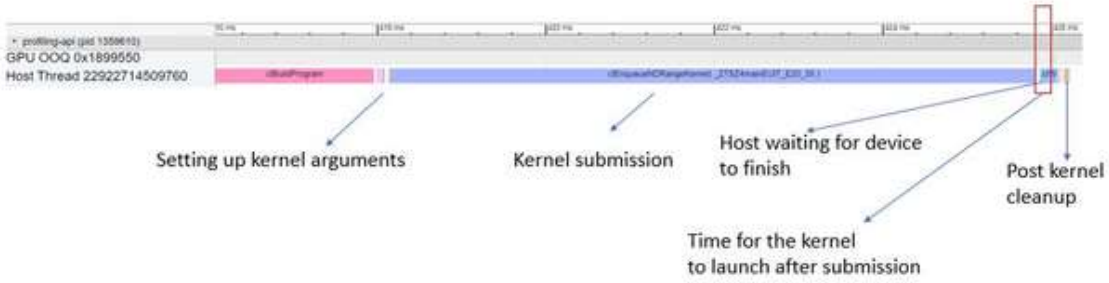
次の図は、上記の例の実行タイムラインを示します。この図は、`clIntercept` を実行してトレースファイルを生成し、Chrome* トレースを使用してタイムラインを視覚化しています。このタイムラインには、2 つのレーンが示されています。1 つはホスト、そしてもう 1 つはデバイスです。デバイス側の唯一のアクティビティーは、送信されたカーネルの実行であることに注目してください。カーネルを実行する準備は、ホスト側でかなりの量のワークが行われています。この例のカーネルは小規模であるため、合計実行時間は、`clBuildProgram` というラベルが付いたブロックでカーネルの JIT コンパイルが大部分を占めていることが分かります。

### カーネル実行のタイムライン



次の拡大図は、カーネルを送信するためホストで呼び出される関数の詳細を示しています。ここでは、時間の大部分が `clEnqueueNDRangeKernel` で占められています。また、ホストでのカーネル送信完了とデバイスでのカーネル起動の間に遅延があることに注意してください。

**カーネルを送信するホスト上の関数呼び出し**




## デバイス上で同時に複数のカーネルを実行

SYCL* には、カーネルを作成して実行のため送信に利用できる 2 種類のキューが用意されています。

- **インオーダー・キュー**
  カーネルはキューに送信された順に実行されます。

- **アウトオブオーダー・キュー**
  カーネルは任意の順に実行されます (カーネル間の依存関係の制約に従います)。

インオーダーまたはアウトオブオーダーでキューを作成するかどうかは、キューの作成時に `sycl::property::queue::in_order()` プロパティーを使用して選択します。デフォルトで、プロパティーが省略された場合のキューはアウトオブオーダーです。

次の例では、反復ごとに 3 つのカーネルが送信されています。それぞれのカーネルは、256 個の work-item を持つ 1 つの work-group のみを使用します。このカーネルは、マシン全体を占有しないようにするため 1 つのグループで生成されています。これは並列カーネル実行の利点を示すために行われます。

```
1.  int multi_queue(sycl::queue &q, const IntArray &a, const IntArray &b) {
2.    IntArray s1, s2, s3;
3.
4.    sycl::buffer a_buf(a);
5.    sycl::buffer b_buf(b);
6.    sycl::buffer sum_buf1(s1);
7.    sycl::buffer sum_buf2(s2);
8.    sycl::buffer sum_buf3(s3);
9.
10.   size_t num_groups = 1;
11.   size_t wg_size = 256;
12.   auto start = std::chrono::steady_clock::now();
13.   for (int i = 0; i < iter; i++) {
14.     q.submit([&](sycl::handler &h) {
15.       sycl::accessor a_acc(a_buf, h, sycl::read_only);
16.       sycl::accessor b_acc(b_buf, h, sycl::read_only);
17.       sycl::accessor sum_acc(sum_buf1, h, sycl::write_only, sycl::no_init);
18.
19.       h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
20.         [=](sycl::nd_item<1> index) {
21.           size_t loc_id = index.get_local_id();
22.           sum_acc[loc_id] = 0;
23.           for (int j = 0; j < 1000; j++)
```

```
24.             for (size_t i = loc_id; i < array_size; i += wg_size) {
25.               sum_acc[loc_id] += a_acc[i] + b_acc[i];
26.             }
27.         });
28.   });
29.   q.submit([&](sycl::handler &h) {
30.     sycl::accessor a_acc(a_buf, h, sycl::read_only);
31.     sycl::accessor b_acc(b_buf, h, sycl::read_only);
32.     sycl::accessor sum_acc(sum_buf2, h, sycl::write_only, sycl::no_init);
33.
34.     h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
35.       [=](sycl::nd_item<1> index) {
36.         size_t loc_id = index.get_local_id();
37.         sum_acc[loc_id] = 0;
38.         for (int j = 0; j < 1000; j++)
39.           for (size_t i = loc_id; i < array_size; i += wg_size) {
40.             sum_acc[loc_id] += a_acc[i] + b_acc[i];
41.           }
42.       });
43.   });
44.   q.submit([&](sycl::handler &h) {
45.     sycl::accessor a_acc(a_buf, h, sycl::read_only);
46.     sycl::accessor b_acc(b_buf, h, sycl::read_only);
47.     sycl::accessor sum_acc(sum_buf3, h, sycl::write_only, sycl::no_init);
48.
49.     h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
50.       [=](sycl::nd_item<1> index) {
51.         size_t loc_id = index.get_local_id();
52.         sum_acc[loc_id] = 0;
53.         for (int j = 0; j < 1000; j++)
54.           for (size_t i = loc_id; i < array_size; i += wg_size) {
55.             sum_acc[loc_id] += a_acc[i] + b_acc[i];
56.           }
57.         });
58.       });
59.   }
60.   q.wait();
61.   auto end = std::chrono::steady_clock::now();
62.   std::cout << "multi_queue completed on device - took "
63.         << (end - start).count() << " u-secs\n";
64.   // 結果をチェック
65.   return ((end - start).count());
66. } // multi_queue の終わり
```

キューがインオーダーである場合、マシンに十分なリソースがあってカーネル間に依存関係がなくても、カーネルを並列に実行することはできず、シーケンシャルに実行されます。これは、すべてのカーネルの合計実行時間が長くなることからも分かります。キューの作成とカーネルの送信方法を以下に示します。

```
sycl::property_list q_prop{sycl::property::queue::in_order()};
std::cout << "In order queue: Jitting+Execution time\n";
sycl::queue q1(sycl::default_selector_v, q_prop);
multi_queue(q1, a, b);
usleep(500 * 1000);
std::cout << "In order queue: Execution time\n";
multi_queue(q1, a, b);
```

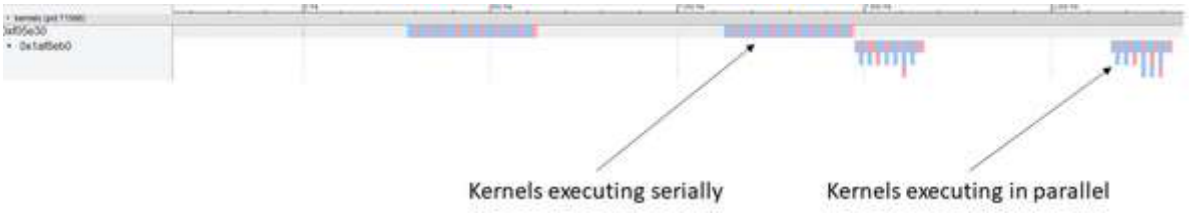
キューがアウトオブオーダーである場合、全体の実行時間はかなり短縮されますが、これは、マシンがキューからカーネルを同時に実行できることを意味します。キューの作成とカーネルの起動方法を以下に示します。

```
sycl::queue q2(sycl::default_selector_v);
std::cout << "Out of order queue: Jitting+Execution time\n";
multi_queue(q2, a, b);
usleep(500 * 1000);
std::cout << "Out of order queue: Execution time\n";
multi_queue(q2, a, b);
```

カーネルがあまりスケーリングせず、マシンの計算リソースを効率良く利用できない状況では、work-group/work-item の値を適切に選択して必要とする計算ユニットだけを割り当てて、同時に複数のカーネルを実行することを推奨します。

次のタイムライン・ニューは、インオーダーおよびアウトオブオーダーのキューで実行されるカーネルを示しています。このタイムラインは、https://github.com/intel/pti-gpu/tree/master/tools/onetrace（英語）で入手できる `onetrace` ツールを使用して収集されました。ここでは、アウトオブオーダー・キューに送信されたカーネルが、並列に実行されていることが明確に分かります。3 つのカーネルすべてが並列に実行されるわけではないことに注意してください。並行に実行されるカーネル数は、ハードウェア・リソースの可用性、カーネル送信間の時間のギャップなど、複数の要因の影響を受けます。

**インオーダーとアウトオブオーダー・キューのカーネル実行タイムライン**




`device` クラスの `create_sub_devices` 関数を使用して、単一デバイスを sub-device に分割することもできます。プログラマーは、適切な sub-device にカーネルを送信するよう制御できます。ただし、デバイスの sub-device へのパーティション化は静的であり、ランタイムでカーネルをある sub-device から別の sub-device へ移行する柔軟性は備わっていないため、アプリケーションの動的な負荷には対応できません。

## 複数のキューへカーネルを送信

キューはアクセラレーターで実行するカーネルを送信するチャネルを提供します。また、デバイスのステートを示すコンテキストも保持します。このステートには、バッファーの内容とカーネルの実行に必要なメモリーが含まれます。ランタイムは、デバイス・コンテキストを追跡して、ホストとデバイス間の不要なメモリー転送を回避します。そのため、異なるコンテキストでカーネルの送信をインターリーブするのではなく、1 つのコンテキストからカーネルを送信して起動することを推奨します。

次の例では、入力と同じバッファーを使用して計算結果を異なる出力バッファーに生成する 30 個の独立したカーネルを送信します。これらのカーネルは完全に独立しており、同時に実行されるため、順不同となる可能性があります。カーネルは 3 つのキューに送信されますが、各カーネルの実行にはキューの作成方法に応じて異なるコストが生じます。

```
1.  int VectorAdd(sycl::queue &q1, sycl::queue &q2, sycl::queue &q3,
2.            const IntArray &a, const IntArray &b) {
3.
4.    sycl::buffer a_buf(a);
5.    sycl::buffer b_buf(b);
6.    sycl::buffer<int> *sum_buf[3 * iter];
7.    for (size_t i = 0; i < (3 * iter); i++)
8.      sum_buf[i] = new sycl::buffer<int>(256);
9.
10.   size_t num_groups = 1;
11.   size_t wg_size = 256;
12.   auto start = std::chrono::steady_clock::now();
13.   for (int i = 0; i < iter; i++) {
14.     q1.submit([&](auto &h) {
15.       sycl::accessor a_acc(a_buf, h, sycl::read_only);
16.       sycl::accessor b_acc(b_buf, h, sycl::read_only);
17.       auto sum_acc = sum_buf[3 * i]->get_access<sycl::access::mode::write>(h);
```

```
18.
19.       h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
20.         [=](sycl::nd_item<1> index) {
21.           size_t loc_id = index.get_local_id();
22.           sum_acc[loc_id] = 0;
23.           for (size_t i = loc_id; i < array_size; i += wg_size) {
24.             sum_acc[loc_id] += a_acc[i] + b_acc[i];
25.           }
26.         });
27.     });
28.     q2.submit([&](auto &h) {
29.       sycl::accessor a_acc(a_buf, h, sycl::read_only);
30.       sycl::accessor b_acc(b_buf, h, sycl::read_only);
31.       auto sum_acc =
32.         sum_buf[3 * i + 1]->get_access<sycl::access::mode::write>(h);
33.
34.       h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
35.         [=](sycl::nd_item<1> index) {
36.           size_t loc_id = index.get_local_id();
37.           sum_acc[loc_id] = 0;
38.           for (size_t i = loc_id; i < array_size; i += wg_size) {
39.             sum_acc[loc_id] += a_acc[i] + b_acc[i];
40.           }
41.         });
42.     });
43.     q3.submit([&](auto &h) {
44.       sycl::accessor a_acc(a_buf, h, sycl::read_only);
45.       sycl::accessor b_acc(b_buf, h, sycl::read_only);
46.       auto sum_acc =
47.         sum_buf[3 * i + 2]->get_access<sycl::access::mode::write>(h);
48.
49.       h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
50.         [=](sycl::nd_item<1> index) {
51.           size_t loc_id = index.get_local_id();
52.           sum_acc[loc_id] = 0;
53.           for (size_t i = loc_id; i < array_size; i += wg_size) {
54.             sum_acc[loc_id] += a_acc[i] + b_acc[i];
55.           }
56.         });
57.     });
58.   }
59.   q1.wait();
60.   q2.wait();
61.   q3.wait();
62.   auto end = std::chrono::steady_clock::now();
63.   std::cout << "Vector add completed on device - took " << (end - start).count()
64.         << " u-secs\n";
65.   // 結果をチェック
66.   for (size_t i = 0; i < (3 * iter); i++)
67.     delete sum_buf[i];
68.   return ((end - start).count());
69. }// VectorAdd の終わり
```

カーネルを同一キューに送信すると、カーネルの転送は最初の一度だけ行われるため、転送のオーバーヘッドが軽減して最高のパフォーマンスが得られます。

```
VectorAdd(q, q, q, a, b);
```

カーネルが同じコンテキストを共有する異なるキューで送信される場合、パフォーマンスは単一のキューで送信された場合とほぼ同等です。ここで注意すべきことは、カーネルが異なるコンテキストで新しいキューに送信されると、JIT プロセスはカーネルをコンテキストに関連する新しいデバイス向けにコンパイルすることです。JIT コンパイルの時間を除けば、カーネルの実行時間はほぼ同じです。

```
sycl::queue q1(sycl::default_selector_v);
sycl::queue q2(q1.get_context(), sycl::default_selector_v);
sycl::queue q3(q1.get_context(), sycl::default_selector_v);
VectorAdd(q1, q2, q3, a, b);
```

カーネルが異なるコンテキストを持つ 3 つの異なるキューに送信される場合、カーネル呼び出し時にランタイムはすべての入力バッファーをアクセラレーターに毎回転送する必要があるため、パフォーマンスが低下します。さらに、カーネルはコンテキストごとに JIT されます。

```
sycl::queue q4(sycl::default_selector_v);
sycl::queue q5(sycl::default_selector_v);
sycl::queue q6(sycl::default_selector_v);
VectorAdd(q4, q5, q6, a, b);
```

何らかの理由で異なるキューを使用する必要がある場合、共有コンテキストでキューを作成することでこの問題を回避できます。これにより、入力バッファーを転送する必要はなくなりますが、すべての出力バッファーがコンテキスト内に存在するため、カーネルが必要とするメモリー容量が増加します (以前は、デバイス上の同じメモリーを出力バッファーとして使用していました)。忘れてならないことは、カーネルのメモリー転送と計算比率の問題です。上記の例では、カーネルの計算量が少ないため、全体の実行はメモリー転送に依存します。計算量が多い場合、転送は実行全体にそれほど影響しません。

これは、カーネルの計算量が 1000 倍に増加し、実行時間が異なる次の例からも分かります。

```
1.  int VectorAdd(sycl::queue &q1, sycl::queue &q2, sycl::queue &q3,
2.          const IntArray &a, const IntArray &b) {
3.
4.    sycl::buffer a_buf(a);
5.    sycl::buffer b_buf(b);
6.    sycl::buffer<int> *sum_buf[3 * iter];
7.    for (size_t i = 0; i < (3 * iter); i++)
8.      sum_buf[i] = new sycl::buffer<int>(256);
9.
10.   size_t num_groups = 1;
11.   size_t wg_size = 256;
12.   auto start = std::chrono::steady_clock::now();
13.   for (int i = 0; i < iter; i++) {
14.     q1.submit([&](auto &h) {
15.       sycl::accessor a_acc(a_buf, h, sycl::read_only);
16.       sycl::accessor b_acc(b_buf, h, sycl::read_only);
17.       auto sum_acc = sum_buf[3 * i]->get_access<sycl::access::mode::write>(h);
18.
19.       h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
20.         [=](sycl::nd_item<1> index) {
21.           size_t loc_id = index.get_local_id();
22.           sum_acc[loc_id] = 0;
23.           for (int j = 0; j < 1000; j++)
24.             for (size_t i = loc_id; i < array_size; i += wg_size) {
25.               sum_acc[loc_id] += a_acc[i] + b_acc[i];
26.             }
27.         });
28.     });
29.     q2.submit([&](auto &h) {
30.       sycl::accessor a_acc(a_buf, h, sycl::read_only);
31.       sycl::accessor b_acc(b_buf, h, sycl::read_only);
32.       auto sum_acc =
33.         sum_buf[3 * i + 1]->get_access<sycl::access::mode::write>(h);
```

```
34.
35.       h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
36.         [=](sycl::nd_item<1> index) {
37.           size_t loc_id = index.get_local_id();
38.           sum_acc[loc_id] = 0;
39.           for (int j = 0; j < 1000; j++)
40.             for (size_t i = loc_id; i < array_size; i += wg_size) {
41.               sum_acc[loc_id] += a_acc[i] + b_acc[i];
42.             }
43.         });
44.     });
45.     q3.submit([&](auto &h) {
46.       sycl::accessor a_acc(a_buf, h, sycl::read_only);
47.       sycl::accessor b_acc(b_buf, h, sycl::read_only);
48.       auto sum_acc =
49.         sum_buf[3 * i + 2]->get_access<sycl::access::mode::write>(h);
50.
51.       h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
52.         [=](sycl::nd_item<1> index) {
53.           size_t loc_id = index.get_local_id();
54.           sum_acc[loc_id] = 0;
55.           for (int j = 0; j < 1000; j++)
56.             for (size_t i = loc_id; i < array_size; i += wg_size) {
57.               sum_acc[loc_id] += a_acc[i] + b_acc[i];
58.             }
59.         });
60.     });
61.   }
62.   q1.wait();
63.   q2.wait();
64.   q3.wait();
65.   auto end = std::chrono::steady_clock::now();
66.   std::cout << "Vector add completed on device - took " << (end - start).count()
67.         << " u-secs\n";
68.   // 結果をチェック
69.   for (size_t i = 0; i < (3 * iter); i++)
70.     delete sum_buf[i];
71.   return ((end - start).count());
72. }// VectorAdd の終わり
```

## 冗長なキューの構築を避ける

デバイスでカーネルを実行するには、関連するコンテキスト、プラットフォーム、およびデバイスを参照するキューを作成する必要があります。コンテキスト、プラットフォーム、およびデバイスは自動的に選択されるか、ユーザーが指定できます。

コンテキストは、ユーザーが直接構築するか、またはキューの作成時に暗黙的に構築され、SYCL* ランタイムと SYCL* バックエンドがデバイス上の動作に必要なすべてのランタイム情報を保持します。コンテキストが省略されてキューが作成されると、デフォルトのコンストラクターを使用して暗黙的に新しいコンテキストが構築されます。

一般的に、新しいコンテキストを作成すると、カーネルがそのコンテキストでキューに送信されるたびに JIT がプログラムをコンパイルするため重い操作であると言えます。良好なパフォーマンスを得るには、アプリケーションで使用するコンテキストをできるだけ少なくする必要があります。

次の例では、ループ内でキューが作成され、カーネルはその新しいキューに送信されます。これは、ループ反復ごとに JIT コンパイラーが呼び出されます。

```
int reductionMultipleQMultipleC(std::vector<int> &data, int iter) {
  const size_t data_size = data.size();
  int sum = 0;

  int work_group_size = 512;
  int num_work_groups = 1;
  int num_work_items = work_group_size;

  const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};

  sycl::buffer<int> buf(data.data(), data_size, props);
  sycl::buffer<int> sum_buf(&sum, 1, props);

  sycl::queue q1{sycl::default_selector_v, exception_handler};
  // デバイス上のデータを初期化
  q1.submit([&](auto &h) {
    sycl::accessor buf_acc(buf, h, sycl::write_only, sycl::no_init);
    h.parallel_for(data_size, [=](auto index) { buf_acc[index] = 1; });
  });

  double elapsed = 0;
  for (int i = 0; i < iter; i++) {
    sycl::queue q2{sycl::default_selector_v, exception_handler};
    if (i == 0)
      std::cout << q2.get_device().get_info<sycl::info::device::name>() << "\n";
    // reductionMapToHWVector main の始まり
    Timer timer;
    q2.submit([&](auto &h) {
      sycl::accessor buf_acc(buf, h, sycl::read_only);
      sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
      sycl::local_accessor<int, 1> scratch(work_group_size, h);
      h.parallel_for(sycl::nd_range<1>{num_work_items, work_group_size},
        [=](sycl::nd_item<1> item) {
          size_t loc_id = item.get_local_id(0);
          int sum = 0;
          for (int i = loc_id; i < data_size; i += num_work_items)
            sum += buf_acc[i];
          scratch[loc_id] = sum;
          for (int i = work_group_size / 2; i > 0; i >>= 1) {
            item.barrier(sycl::access::fence_space::local_space);
            if (loc_id < i)
              scratch[loc_id] += scratch[loc_id + i];
          }
          if (loc_id == 0)
            sum_acc[0] = scratch[0];
        });
    });
    // reductionMapToHWVector main の終わり
    q2.wait();
    sycl::host_accessor h_acc(sum_buf);
    sum = h_acc[0];
    elapsed += timer.Elapsed();
  }
  elapsed = elapsed / iter;
  if (sum == sum_expected)
    std::cout << "SUCCESS: Time reductionMultipleQMultipleC   = " << elapsed
          << "s"
          << " sum = " << sum << "\n";
  else
    std::cout << "ERROR: reductionMultipleQMultipleC Expected " << sum_expected
          << " but got " << sum << "\n";
  return sum;
} // reductionMultipleQMultipleC の終わり
```

上記のプログラムは、キューの宣言をループ外に移動することで大幅にパフォーマンスを向上できます。

```
1.  int reductionSingleQ(std::vector<int> &data, int iter) {
2.    const size_t data_size = data.size();
3.    int sum = 0;
4.
5.    int work_group_size = 512;
6.    int num_work_groups = 1;
7.    int num_work_items = work_group_size;
8.
9.    const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
10.
11.   sycl::buffer<int> buf(data.data(), data_size, props);
12.   sycl::buffer<int> sum_buf(&sum, 1, props);
13.   sycl::queue q{sycl::default_selector_v, exception_handler};
14.   std::cout << q.get_device().get_info<sycl::info::device::name>() << "\n";
15.
16.   // デバイス上のデータを初期化
17.   q.submit([&](auto &h) {
18.     sycl::accessor buf_acc(buf, h, sycl::write_only, sycl::no_init);
19.     h.parallel_for(data_size, [=](auto index) { buf_acc[index] = 1; });
20.   });
21.
22.   double elapsed = 0;
23.   for (int i = 0; i < iter; i++) {
24.     // reductionIntBarrier main の始まり
25.     Timer timer;
26.     q.submit([&](auto &h) {
27.       sycl::accessor buf_acc(buf, h, sycl::read_only);
28.       sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
29.       sycl::local_accessor<int, 1 > scratch(work_group_size, h);
30.       h.parallel_for(sycl::nd_range<1>{num_work_items, work_group_size},
31.         [=](sycl::nd_item<1> item) {
32.           size_t loc_id = item.get_local_id(0);
33.           int sum = 0;
34.           for (int i = loc_id; i < data_size; i += num_work_items)
35.             sum += buf_acc[i];
36.           scratch[loc_id] = sum;
37.           for (int i = work_group_size / 2; i > 0; i >>= 1) {
38.             item.barrier(sycl::access::fence_space::local_space);
39.             if (loc_id < i)
40.               scratch[loc_id] += scratch[loc_id + i];
41.           }
42.           if (loc_id == 0)
43.             sum_acc[0] = scratch[0];
44.         });
45.     });
46.     // reductionSingleQ main の終わり
47.     q.wait();
48.     sycl::host_accessor h_acc(sum_buf);
49.     sum = h_acc[0];
50.     elapsed += timer.Elapsed();
51.   }
52.   elapsed = elapsed / iter;
53.   if (sum == sum_expected)
54.     std::cout << "SUCCESS: Time reductionSingleQ   = " << elapsed << "s"
55.           << " sum = " << sum << "\n";
56.   else
57.     std::cout << "ERROR: reductionSingleQ Expected " << sum_expected
58.           << " but got " << sum << "\n";
59.   return sum;
60. } // reductionSingleQ の終わり
```

複数のキューを作成する場合、キュー間でコンテキストを共有するようにします。これによりパフォーマンスが向上します。上記のカーネルは次のように書き直すことができ、ここではループ内で作成されたキューとループ外のキューがコンテキストを共有しています。この場合、パフォーマンスは単一のキューを使用する場合とほぼ同等です。

```
int reductionMultipleQSingleC(std::vector<int> &data, int iter) {
  const size_t data_size = data.size();
  int sum = 0;

  int work_group_size = 512;
  int num_work_groups = 1;
  int num_work_items = work_group_size;

  const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};

  sycl::buffer<int> buf(data.data(), data_size, props);
  sycl::buffer<int> sum_buf(&sum, 1, props);

  sycl::queue q1{sycl::default_selector_v, exception_handler};
  // デバイス上のデータを初期化
  q1.submit([&](auto &h) {
    sycl::accessor buf_acc(buf, h, sycl::write_only, sycl::no_init);
    h.parallel_for(data_size, [=](auto index) { buf_acc[index] = 1; });
  });

  double elapsed = 0;
  for (int i = 0; i < iter; i++) {
    sycl::queue q2{q1.get_context(), sycl::default_selector_v,
              exception_handler};
    if (i == 0)
      std::cout << q2.get_device().get_info<sycl::info::device::name>() << "\n";
    // reductionMultipleQSingleC main の始まり
    Timer timer;
    q2.submit([&](auto &h) {
      sycl::accessor buf_acc(buf, h, sycl::read_only);
      sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
      sycl::local_accessor<int, 1> scratch(work_group_size, h);
      h.parallel_for(sycl::nd_range<1>{num_work_items, work_group_size},
        [=](sycl::nd_item<1> item) {
          size_t loc_id = item.get_local_id(0);
          int sum = 0;
          for (int i = loc_id; i < data_size; i += num_work_items)
            sum += buf_acc[i];
          scratch[loc_id] = sum;
          for (int i = work_group_size / 2; i > 0; i >>= 1) {
            item.barrier(sycl::access::fence_space::local_space);
          if (loc_id < i)
            scratch[loc_id] += scratch[loc_id + i];
          }
          if (loc_id == 0)
            sum_acc[0] = scratch[0];
        });
    });
    // reductionMultipleQSingleC main の終わり
    q2.wait();
    sycl::host_accessor h_acc(sum_buf);
    sum = h_acc[0];
    elapsed += timer.Elapsed();
  }
  elapsed = elapsed / iter;
  if (sum == sum_expected)
    std::cout << "SUCCESS: Time reductionMultipleQSingleContext  = " << elapsed
          << "s"
          << " sum = " << sum << "\n";
  else
    std::cout << "ERROR: reductionMultipleQSingleContext Expected "
          << sum_expected << " but got " << sum << "\n";
  return sum;
} // reductionMultipleQSingleC の終わり
```

# SYCL* 結合行列拡張を使用したインテル® XMX のプログラミング

結合行列は、行列ハードウェア向けのプログラミングの新しい SYCL* 拡張機能です。これにより、CPU 向けのインテル® アドバンスト・マトリックス・エクステンション (インテル® AMX)、GPU 向けのインテル® Xe マトリクス・エクステンション (インテル® XMX)、NVIDIA* Tensor コアなどのターゲットを統合できます。一般に、TensorFlow* などのフレームワークや oneDNN などのライブラリーは、多くの種類の AI ユーザーとアプリケーションにとって有用です。しかし、独自のニューラル・ネットワーク・アプリケーションを構築するユーザーにとって、これらのフレームワークやライブラリーは、ユーザー独自の最適化を適用できない高レベルな実装であり、ライブラリーのサイズが大きく、負荷も大きくなります。さらに、マシンラーニングの分野では、常に新しい操作が導入され変化しているため、フレームワークやライブラリーがタイムリーにその解決策を提供できないこともあります。そのような場合、結合行列は、フレームワークよりも抽象化レベルが高く、パフォーマンス、生産性、そして融合機能を提供するため、単一ソースコードから異なる行列ハードウェアをターゲットにすることで移植性も高めます。

この拡張機能の詳細な仕様は、こちら (英語) で確認できます。

結合行列拡張は、新しいタイプの `joint_matrix`、明示的なメモリー操作の `joint_matrix_load` および `joint_matrix_store`、行列初期化用の `joint_matrix_fill`、乗算加算操作を行う `joint_matrix_mad`、そして要素ごとの操作を行う `joint_matrix_apply` で構成されます。

次のコードでは、カーネルは 3 つの `joint_matrix` 行列 (`tA`、`tB`、`tC`) を宣言することで結合行列インターフェイスを利用して、`tC += tA * tB` 操作を実行します。DPAS (Dot Product and Accumulate Systolic) は、インテル® XMX で実行される基本操作の名称です。`joint_matrix` の使用例については、インテルの llvm-test-suite リポジトリー (英語) を参照してください。

この例をインテル® XMX で正しく実行するには、GPU としてインテル® XMX ハードウェアが必要です。合成行列の実装には、エミュレーションやフォールバックの機能は提供されません。

```
size_t NDRangeM = M / TM;
size_t NDRangeN = N / TN;
sycl::buffer<bfloat16, 2> bufA(A.get_data(), sycl::range<2>(M, K));
sycl::buffer<bfloat16, 2> bufB(B.get_data(), sycl::range<2>(K, N));
sycl::buffer<float, 2> bufC((float *)C.get_data(), sycl::range<2>(M, N));

sycl::queue q;
q.submit([&](sycl::handler &cgh) {
   sycl::accessor accC(bufC, cgh, sycl::read_write);
   sycl::accessor accA(bufA, cgh, sycl::read_only);
   sycl::accessor accB(bufB, cgh, sycl::read_only);
   cgh.parallel_for(
       sycl::nd_range<2>({NDRangeM, NDRangeN * SG_SZ}, {1, 1 * SG_SZ}),
       [=](sycl::nd_item<2> spmd_item) [[intel::reqd_sub_group_size(SG_SZ)]]

       {
         // 合成行列 API は、sub-group 内のすべての work-item からアクセスできる必要があります。
         // これらの関数は、sub-group から一度呼び出されます。
         // work-item 間のコードには変更ありません。
         const auto global_idx = spmd_item.get_global_id(0);
         const auto global_idy = spmd_item.get_global_id(1);
         const auto sg_startx = global_idx - spmd_item.get_local_id(0);
         const auto sg_starty = global_idy - spmd_item.get_local_id(1);

         sycl::sub_group sg = spmd_item.get_sub_group();
         sycl::ext::oneapi::experimental::matrix::joint_matrix<
             sycl::sub_group, bfloat16, use::a, TM, TK, layout::row_major>
             sub_a;
         // B はすでに VNNI に変更されていると仮定します。
```

```
          sycl::ext::oneapi::experimental::matrix::joint_matrix<
              sycl::sub_group, bfloat16, use::b, TK, TN,
              sycl::ext::intel::experimental::matrix::layout::packed>
              sub_b;
          sycl::ext::oneapi::experimental::matrix::joint_matrix<
              sycl::sub_group, float, use::accumulator, TM, TN>
              sub_c;

          joint_matrix_fill(sg, sub_c, 1.0);
          for (int k = 0; k < K / TK; k += 1) {
            joint_matrix_load(
                sg, sub_a,
                accA.template get_multi_ptr<sycl::access::decorated::no>() +
                  (sg_startx * TM) * K + k * TK,
              K);
          joint_matrix_load(
              sg, sub_b,
              accB.template get_multi_ptr<sycl::access::decorated::no>() +
                  (k * TK / 2) * (N * 2) + sg_starty / SG_SZ * TN * 2,
              N * 2);
            sub_c = joint_matrix_mad(sg, sub_a, sub_b, sub_c);
          }
          joint_matrix_apply(sg, sub_c, [=](float &x) { x *= ALPHA; });
          joint_matrix_store(
              sg, sub_c,
              accC.template get_multi_ptr<sycl::access::decorated::no>() +
                  (sg_startx * TM) * N + sg_starty / SG_SZ * TN,
              N, layout::row_major);
    }); // parallel for
}).wait();
```

インテル® XMX を使用して最適なパフォーマンスを得るには、インテル® XMX に最大の乗算と操作/サイクルを維持するのに必要なデータを供給し続けるように GEMM カーネルを作成する必要があります。そのチューニング手法の一部を以下に示します。

- 1 つの sub-group が複数の DPAS 操作を実行できます。
- キャッシュ局所性のブロック化は、3 つの `i`、`j`、および `k` 次元で行う必要があります。最初の 2 つの次元のブロック化は、カーネルのグローバルレンジに含まれます。`k` レベルのキャッシュブロック化は、カーネル内で行われます。例えば、256 x 256 x 32 のブロック係数では、グローバルレンジは次のようになります。

  ```
  range<2> global{M / 256, N / 256 * SG_SIZE};
  ```

  次に、1 つの sub-group を選択して 64 x 32 x 32 要素を実行すると、ローカルレンジは次のようになります。

  ```
  range<2> local{256 / 64, 256 / 32 * SG_SIZE};
  ```

- スレッドごとにラージ GRF モードを使用すると、GEMM カーネルで最良の結果が得られます。これには、次のコンパイルコマンドを使用します。

  ```
  $ icpx -fsycl -fsycl-targets=spir64_gen -Xsycl-target-backend "-device pvc -options
  -ze-opt-large-register-file" joint-matrix.cpp
  ```

## カーネル内での I/O

`Print` 文は、プログラムの結果を確認する最も基本的な機能です。アクセラレーターでは、プリントはかなり困難であり、オーバーヘッドの面でも高コストです。

SYCL* は、このタスクが標準の I/O C/C++ プログラムと同様になるようにいくつかの機能を提供しますが、アクセラレーターの仕組み上、理解しておかなければならない点があります。SYCL* カーネルでは、ファイル I/O は許可されません。

SYCL* は、カーネル内からコンソールに情報を出力する `stream` クラスを提供します。これにより、デバッガーを使用することなく簡単なデバッグができます。`stream` クラスは、C++ STL `ostream` クラスとよく似た機能を提供し、使い方は STL クラスに類似しています。以下では、SYCL* `stream` クラスを使用して、キューに投入されたカーネルから情報を出力する方法について説明します。

クラスを使用するには、最初にクラスをインスタンス化する必要があります。`stream` コンストラクターのシグネチャーは次のとおりです。

```
stream(size_t BufferSize, size_t MaxStatementSize, handler &CGH);
```

コンストラクターには 3 つのパラメーターがあります。

- `BufferSize`: カーネルレンジ全体で出力する文字の総数
- `MaxStatementSize`: `stream` クラスの 1 回の呼び出しの最大文字数
- CGH: `sycl::queue::submit` 呼び出しの `sycl::handler` パラメーターへの参照

使用法は、C++ STL `ostream std::cout` クラスの使い方と類似しています。出力するメッセージまたはデータは、適切なオペレーター `<<` メソッドにより SYCL* ストリーム・インスタンスに送信されます。SYCL* は、すべてのビルトイン・データ・タイプ (`int`、`char`、`float` など) と一般的なクラス (`sycl::nd_range` や `sycl::group` など) の実装を提供します。

次に、SYCL `stream` インスタンスの使い方を示します。

```
void out1() {
  constexpr int N = 16;
  sycl::queue q;
  q.submit([&](auto &cgh) {
    sycl::stream str(8192, 1024, cgh);
    cgh.parallel_for(N, [=](sycl::item<1> it) {
      int id = it[0];
      /*  コンソールに表示するストリームに識別子を送信  */
      str << "ID=" << id << sycl::endl;
    });
  }).wait();
} // out1 の終わり
```

`sycl::endl` は、C++ STL の `std::endl ostream` に類似しています。これには、改行を挿入するだけでなく、ストリームをフラッシュする効果があります。

上記のカーネルをコンパイルして実行すると、次の出力が得られます。

```
ID=0
ID=1
ID=2
ID=3
ID=4
ID=5
ID=6
ID=7
ID=8
ID=9
ID=10
ID=11
ID=12
ID=13
ID=14
ID=15
```

`BufferSize` と `MaxStatementSize` パラメーターの選択には注意が必要です。サイズが足りない場合、ステートメントが出力されないか、欠落した情報が表示されることがあります。次のカーネルについて考えてみます。

```
void out2() {
  sycl::queue q;
  q.submit([&](auto &cgh) {
    sycl::stream str(8192, 4, cgh);
    cgh.parallel_for(1, [=](sycl::item<1>) {
      str << "ABC" << sycl::endl;  // Print 文 1
      str << "ABCDEFG" << sycl::endl; // Print 文 2
    });
  }).wait();
} // out2 の終わり
```

上記のカーネルをコンパイルして実行すると、次の出力が得られます。

```
ABC
```

表示される文字数が 4 (`sycl::endl` による改行を含む) であり、`MaxStatementSize` (`sycl::stream` の最大ステートメント・サイズ・パラメーターで指定) も 4 であるため、最初のステートメントは正しく表示されました。ただし、2 番目のステートメントの改行のみが出力されます。

次のカーネルは、許容される最大文字サイズを増やした場合の影響を示しています。

```
void out3() {
  sycl::queue q;
  q.submit([&](auto &cgh) {
    sycl::stream str(8192, 10, cgh);
    cgh.parallel_for(1, [=](sycl::item<1>) {
      str << "ABC" << sycl::endl;  // Print 文 1
      str << "ABCDEFG" << sycl::endl; // Print 文 2
    });
  }).wait();
} // out3 の終わり
```

上記のカーネルをコンパイルして実行すると、期待する出力が得られます。

```
ABC
ABCDEFG
```

この例では、単一の work-item を持つ単純なカーネルを使用しています。より現実に近いカーネルには、通常、複数の work-item が含まれます。その場合、コンソールに出力されるステートメントの順番に保証はありません。さまざまな work-item のステートメントが順不同で出力されることを理解している必要があります。次のカーネルについて考えてみます。

```
void out4() {
  sycl::queue q;
  q.submit([&](auto &cgh) {
    sycl::stream str(8192, 1024, cgh);
    cgh.parallel_for(sycl::nd_range<1>(32, 4), [=](sycl::nd_item<1> it) {
      int id = it.get_global_id();
      str << "ID=" << id << sycl::endl;
  });
}).wait(); // out4 の終わり
```

実行すると次のような結果が得られます。

```
ID=0
ID=1
ID=2
ID=3
ID=4
ID=5
[snip]
ID=26
ID=27
ID=28
ID=29
ID=30
ID=31
```

このプログラムを再度実行すると、スレッドが実行される順序に応じて、全く異なる出力が生成される可能性があります。

```
ID=4
ID=5
ID=6
ID=7
ID=0
ID=1
[snip]
ID=14
ID=15
ID=28
ID=29
ID=30
ID=31
```

`sycl::stream` からの出力は、カーネルが実行を完了してから出力されます。通常これは問題とはなりません。ただし、カーネルに障害が発生したり、例外がスローされる場合、ステートメントは表示されません。例外が発生する次のカーネルの例を考えてみます。

```
void out5() {
  int *m = NULL;
  sycl::queue q;
  q.submit([&](auto &cgh) {
    sycl::stream str(8192, 1024, cgh);
    cgh.parallel_for(sycl::nd_range<1>(32, 4), [=](sycl::nd_item<1> it) {
      int id = it.get_global_id();
      str << "ID=" << id << sycl::endl;
      if (id == 31)
        *m = id;
    });
  }).wait();
} // out5 の終わり
```

上記のコードをコンパイルして実行すると、NULL ポインターへの書き込みが原因でセグメンテーション違反が発生します。

```
Segmentation fault (core dumped)
```

いずれの `print` 文もコンソールに出力されません。代わりに、セグメンテーション違反を示すエラーメッセージが表示されます。これは、従来の C/C++ ストリームとは異なります。

# GPU オフロードにライブラリーを使用

oneAPI ツールキットでは、最適化されたアプリケーションで使用できる特殊な API を提供することで、プログラミングを簡素化できるライブラリーを用意しています。この章では、アプリケーションを加速するコードサンプルを含むライブラリーの利用手順について説明します。利用可能な API を含む各ライブラリーの詳細は、それぞれのライブラリーのドキュメントに記載されています。

- パフォーマンス・ライブラリーを使用
- SYCL* カーネルで共有ライブラリー関数を使用
- oneAPI マス・カーネル・ライブラリー (oneMKL) を使用した効率良いフーリエ相関の実装

## パフォーマンス・ライブラリーを使用

ここでは、明示的にコードを記述する代わりに、oneAPI マス・カーネル・ライブラリー (oneMKL) や、oneAPI ディープ・ニューラル・ネットワーク・ライブラリー (oneDNN) などのライブラリーの効率良い関数を使用する方法を説明します。特定分野の数学演算を研究している専門家でない限り、通常、演算を行う独自バージョンを作成することは推奨しません。例えば、行列乗算は一般的で単純な数学演算です。

$$C_{m,n} = A_{m,k} \times B_{k,n} = \sum^{k} A_{m,k} \times B_{k,n}$$

これは数行のコードで簡単に実装できます。

```
// A と B の行列乗算
for (m = 0; m < M; m++) {
  for (n = 0; n < N; n++) {
    C[m][n] = 0.0;
    for (k = 0; k < K; k++) {
      C[m][n] += A[m][k] * B[k][n];
    }
  }
} // 行列乗算の終わり
```

これは単純な実装ですが、最高のパフォーマンスを得られるとは限りません。内部ループをよく見ると、行列 B のメモリーアクセスが非連続であることが分かります。これは、キャッシュを再利用できずパフォーマンスが低下します。

このアルゴリズムを SYCL* に移植して、行列乗算カーネルをアクセラレーターで実行することは、それほど困難ではありません。次のコードは、キューを初期化してデフォルトデバイスにワークを送信し、統合共有メモリー (USM) に行列空間を割り当てています。

```
// SYCL* キューを初期化
sycl::queue Q(sycl::default_selector_v);
auto sycl_device = Q.get_device();
auto sycl_context = Q.get_context();
std::cout << "Running on: "
          << Q.get_device().get_info<sycl::info::device::name>() << std::endl;

// USM に行列 A, B, および C を割り当て
auto A = sycl::malloc_shared<float *>(M, sycl_device, sycl_context);
for (m = 0; m < M; m++)
  A[m] = sycl::malloc_shared<float>(K, sycl_device, sycl_context);

auto B = sycl::malloc_shared<float *>(K, sycl_device, sycl_context);
for (k = 0; k < K; k++)
  B[k] = sycl::malloc_shared<float>(N, sycl_device, sycl_context);

auto C = sycl::malloc_shared<float *>(M, sycl_device, sycl_context);
for (m = 0; m < M; m++)
  C[m] = sycl::malloc_shared<float>(N, sycl_device, sycl_context);

// 行列 A, B, および C の初期化
```

USM のデータは、SYCL* ランタイムによってホストメモリーとデバイスメモリー間で移動できます。明示的なバッファーは必要ありません。デフォルトのアクセラレーターにオフロードするため、計算は SYCL* カーネルに変換され、キューに送信されます。

```
// 行列乗算カーネルをオフロード
Q.parallel_for(sycl::range<2>{M, N}, [=](sycl::id<2> id) {
  unsigned int m = id[0];
  unsigned int n = id[1];

  float sum = 0.0;
  for (unsigned int k = 0; k < K; k++)
    sum += A[m][k] * B[k][n];

  C[m][n] = sum;
}).wait(); // 行列乗算の終わり
```

しかし、このようなコードをアクセラレーターにオフロードしても、パフォーマンスが向上する可能性はほとんどありません。実際には、パフォーマンスが低下します。適切に記述されていないコードは、ホストまたはデバイスで実行されているにかかわらず適切でないままです。

行列乗算など、計算量の多い一般的な演算は十分に研究されています。研究者は、基本的な数式を純粋に実装するよりも優れたパフォーマンスをもたらす多くのアルゴリズムを設計しました。それらは、キャッシュ・ブロッキングやループアンロールなどのチューニングにより、行列 `A` および `B` の形状に関係なく高いパフォーマンスを実現します。

oneMKL は、最適化済みの一般的な行列乗算関数 (`oneapi::mkl::blas::gemm`) を提供し、ホスト・プロセッサーまたは各種アクセラレーター・デバイスで高い性能を実現できます。行列は USM に割り当てられ、デバイス キュー、行列次元、およびそのほか各種オプションが `gemm` 関数に渡されます。

```
// 行列乗算をオフロード
float alpha = 1.0, beta = 0.0;
oneapi::mkl::transpose transA = oneapi::mkl::transpose::nontrans;
oneapi::mkl::transpose transB = oneapi::mkl::transpose::nontrans;
sycl::event gemm_done;
std::vector<sycl::event> gemm_dependencies;
gemm_done = oneapi::mkl::blas::gemm(Q, transA, transB, M, N, K, alpha, A, M,
                                    B, K, beta, C, M, gemm_dependencies);
gemm_done.wait();
```

ライブラリー関数は、純粋な実装よりも高いパフォーマンスを示し、広範囲の用途に利用できます。例えば、ライブラリー関数は、必要に応じて乗算の前に一方または両方の行列を転置できます。この簡単な例は、アプリケーション開発者とチューニングの熟練者の関心がかけ離れていることを示しています。アプリケーション開発者は、高度に最適化されたライブラリーに一般的な計算をカプセル化するため熟練者に依存する傾向があります。oneAPI 仕様では、アプリケーションの高速化に役立つ多くのライブラリーが定義されています。以下に例を示します。

- 数学演算向けの oneMKL
- データ解析とマシンラーニング向けの oneDAL
- マシンラーニング・フレームワークを開発するための oneDNN
- ビデオ処理向けのインテル® VPL

必要な操作をスクラッチから実装する前に、oneAPI ライブラリーに利用可能な関数がないか確認してください。

## SYCL* カーネルで共有ライブラリー関数を使用

すべてではありませんが、一部の標準 C++ 関数は SYCL* カーネル内から呼び出すことができます。関数の概要については、『データ並列 C++』(英語) 書籍の第 18 章 (ライブラリー) を参照してください。サポートされない関数が SYCL* カーネルから呼び出された際に何が発生するか説明するため、簡単な例を示します。次のプログラムは、`rand()` 関数を使用して乱数シーケンスを生成します。

```
1.  #include <CL/sycl.hpp>
2.  #include <iostream>
3.  #include <random>
4.
5.  constexpr int N = 5;
6.
7.  extern SYCL_EXTERNAL int rand(void);
8.
9.  int main(void) {
10. #if defined CPU
11.   sycl::queue Q(sycl::cpu_selector_v);
12. #elif defined GPU
13.   sycl::queue Q(sycl::gpu_selector_v);
```

```
#else
  sycl::queue Q(sycl::default_selector_v);
#endif

  std::cout << "Running on: "
            << Q.get_device().get_info<sycl::info::device::name>() << std::endl;

  // SYCL* カーネル内で rand() を利用してみる
  auto test1 = sycl::malloc_shared<float>(N, Q.get_device(), Q.get_context());

  srand((unsigned)time(NULL));
  Q.parallel_for(N, [=](auto idx) {
     test1[idx] = (float)rand() / (float)RAND_MAX;
   }).wait();

  // 乱数のシーケンスを表示
  for (int i = 0; i < N; i++)
    std::cout << test1[i] << std::endl;

  // クリーンアップ
  sycl::free(test1, Q.get_context());
}
```

プログラムは、CPU (`cpu_selector`)、または GPU (`gpu_selector`) デバイスで実行される SYCL* カーネルにコンパイルできます。2 つのデバイスに対しエラーなしでコンパイルされ、CPU では正しく実行できますが、GPU で実行すると失敗します。

```
$ icpx -fsycl -DCPU -std=c++17 external_rand.cpp -o external_rand
$ ./external_rand
Running on: Intel(R) Xeon(R) E-2176G CPU @ 3.70GHz
0.141417
0.821271
0.898045
0.218854
0.304283

$ icpx -fsycl -DGPU -std=c++17 external_rand.cpp -o external_rand
$ ./external_rand
Running on: intel® Graphics Gen9 [0x3e96]
terminate called after throwing an instance of 'cl::sycl::compile_program_error'
  what():  The program was built for 1 devices
Build program log for 'Intel(R) Graphics Gen9 [0x3e96]':

error: undefined reference to `rand()'

error: backend compiler failed build.
 -11 (CL_BUILD_PROGRAM_FAILURE)
Aborted
```

`rand()` への未定義参照があるため、ジャストインタイム (JIT) コンパイルで障害が発生します。この関数は `SYCL_EXTERNAL` として宣言されていますが、GPU デバイスの `rand()` 関数に相当する SYCL* 関数がありません。

幸い、SYCL* ライブラリーには乱数生成関数など、多くの標準 C++ 関数の代替が含まれています。次の例は、oneAPI DPC++ ライブラリー (oneDPL) (英語) と oneAPI マス・カーネル・ライブラリー (oneMKL) (英語) を使用した同等のコードを示しています。

```
#include <CL/sycl.hpp>
#include <iostream>
#include <oneapi/dpl/random>
#include <oneapi/mkl/rng.hpp>

int main(int argc, char **argv) {
  unsigned int N = (argc == 1) ? 20 : std::stoi(argv[1]);
```

```
8.    if (N < 20)
9.      N = 20;
10.
11.   // oneDPL と oneMKL を使用して、[0.0, 1.0] 間の乱数シーケンスを生成
12.   //
13.   sycl::queue Q(sycl::gpu_selector_v);
14.   std::cout << "Running on: "
15.             << Q.get_device().get_info<sycl::info::device::name>() << std::endl;
16.
17.   auto test1 = sycl::malloc_shared<float>(N, Q.get_device(), Q.get_context());
18.   auto test2 = sycl::malloc_shared<float>(N, Q.get_device(), Q.get_context());
19.
20.   std::uint32_t seed = (unsigned)time(NULL); // RNG シード値を取得
21.
22.   // GPU デバイス上の oneDPL 乱数生成器
23.   clock_t start_time = clock(); // タイマーを開始
24.
25.   Q.parallel_for(N, [=](auto idx) {
26.      oneapi::dpl::minstd_rand rng_engine(seed, idx); // RNG エンジンを初期化
27.      oneapi::dpl::uniform_real_distribution<float>
28.          rng_distribution;                        // RNG 分布を設定
29.      test1[idx] = rng_distribution(rng_engine); // RNG シーケンスを生成
30.    }).wait();
31.
32.   clock_t end_time = clock(); // タイマーを停止
33.   std::cout << "oneDPL took " << float(end_time - start_time) / CLOCKS_PER_SEC
34.             << " seconds to generate " << N
35.             << " uniformly distributed random numbers." << std::endl;
36.
37.   // GPU デバイス上の oneMKL 乱数生成器
38.   start_time = clock(); // タイマーを開始
39.
40.   oneapi::mkl::rng::mcg31m1 engine(
41.       Q, seed); // RNG エンジンを初期化し、RNG 分布を設定
42.   oneapi::mkl::rng::uniform<float, oneapi::mkl::rng::uniform_method::standard>
43.       rng_distribution(0.0, 1.0);
44.   oneapi::mkl::rng::generate(rng_distribution, engine, N, test2)
45.       .wait(); // RNG シーケンスを生成
46.
47.   end_time = clock(); // タイマーを停止
48.   std::cout << "oneMKL took " << float(end_time - start_time) / CLOCKS_PER_SEC
49.             << " seconds to generate " << N
50.             << " uniformly distributed random numbers."<< std::endl;
51.
52.   // 各メソッドの最初の 10 個の乱数を表示
53.   std::cout << std::endl
54.             << "oneDPL"
55.             << "\t"
56.             << "oneMKL" << std::endl;
57.   for (int i = 0; i < 10; i++)
58.     std::cout << test1[i] << " " << test2[i] << std::endl;
59.
60.   // 各メソッドの最後の 10 個の乱数を表示
61.   std::cout << "..." << std::endl;
62.   for (size_t i = N - 10; i < N; i++)
63.     std::cout << test1[i] << " " << test2[i] << std::endl;
64.
65.   // クリーンアップ
66.   sycl::free(test1, Q.get_context());
67.   sycl::free(test2, Q.get_context());
68. }
```

oneDPL および oneMKL 関数は、それぞれ `<oneapi/dpl/random>` と `<oneapi/mkl/rng.hpp>` に含まれています。oneDPL と oneMKL の例では同じ操作のシーケンスを実行します。クロックから乱数シードを取得して、乱数エンジンを初期化し、目的とする乱数分布を選択して乱数を生成します。oneDPL コードは、明示的に SYCL* カーネルを使用してデバイスのオフロードを実行します。oneMKL コードでは、`mkl::rng` 関数がデバイスのオフロードを暗黙的に処理します。

## oneAPI マス・カーネル・ライブラリー (oneMKL) を使用した効率良いフーリエ相関の実装

これまで oneMKL カーネル関数の簡単な使い方を説明しましたが、相互相関と呼ばれるより複雑な数学演算を見てみましょう。相互相関には多くの用途があります。例えば、2 つの 1D 信号の類似性を測定し、類似したイメージをオーバーレイする最適な変換方法を検出する医療イメージの体積セグメント化などがあります。

1 と 0 のベクトルとして表現される信号について考えてみます。

```
Signal 1:  0  0  0  0  0  1  1  0
Signal 2:  0  0  1  1  0  0  0  0
```

シグナルは循環シフトとして扱われるため、2 番目のシグナルを最初のシグナルに対して 3 要素シフトすると、最大相関は 2 となります。

```
Signal 1: 0 0 0 0 0 1 1 0
Signal 2:    0 0 1 1 0 0 0 0

Correlation: (1 * 1) + (1 * 1) = 2
```

2 または 4 要素のシフトは、1 の相関スコアを与えます。それ以外のシフトでは相関スコアはゼロになります。これは以下のように計算されます。

$$corr_{\alpha} = \sum_{i=0}^{N-1} sig1_i \times sig2_{i+\alpha}$$

ここで、`N` はシグナルベクトルの要素数、`i` は `sig1` に対する `sig2` のシフトです。

実際のシグナルには多くのデータ (およびノイズ) が含まれますが、1D シグナルの位置に合わせ、2D イメージのオーバーレイ、3D 立体イメージ・レジストレーションのいずれを実行する場合でも原理は同じです。ここでの目的は、相関を最大化する変換を見つけることです。ただし、上記の総当たりの合計では、`N` シフトごとに `N` 回の乗算と加算が必要になります。1D、2D、および 3D では、問題は $O(N^2)$、$O(N^3)$ 、および $O(N^4)$ となります。

フーリエ相関アルゴリズムは、フーリエ変換 $O(NlogN)$ の利点を活用するため、この計算を実行するはるかに効率良い方法と言えます。

```
corr = IDFT(DFT(sig1) * CONJG(DFT(sig2)))
```

ここで `DFT` は離散フーリエ変換、`IDFT` は逆 DFT、`CONJG` は複素共役です。フーリエ相関アルゴリズムは、最適化された順方向および逆方向の変換と、複素共役の乗算関数をサポートする oneMKL を使用して構成できます。つまり、計算全体をアクセラレーター・デバイスで実行できます。

多くのアプリケーションでは、最終的な結果のみが必要であるため、デバイスからホストに転送するデータはこれだけで済みます。

この例では、2 つの人工的なシグナルがデバイスで生成され、インプレースで変換された後に相関されます。ホストは最終結果を取得して、最善の変換と相関のストアをレポートします。これまでの経験では、バッファリングはホストとデバイス間のデータ転送を明示的に制御できるため、最高のパフォーマンスが得られると言われていました。

この仮説を検証するため、2 つの入力シグナルを生成してみます。

```
// シグナルデータのバッファーを作成。これはデバイスでのみ使用されます。
sycl::buffer<float> sig1_buf{N + 2};
sycl::buffer<float> sig2_buf{N + 2};

// 相関結果を保持するコンテナを宣言
// (デバイスで計算され、ホストで使用されます)
std::vector<float> corr(N + 2);
```

ニューラル・ネットワークの訓練中に過適合を防止するため、イメージに視覚効果を追加するため、または最適ではない検出器から取得したシグナルの検出能を向上させるため、シグナルにランダムノイズが追加されることがあります。ここでは、oneMKL の単純な乱数生成器を使用してバッファーをランダムノイズで初期化します。

```
// 新しいスコープを開いて、相関結果の更新をトリガー
  {
    sycl::buffer<float> corr_buf(corr);

    // 入力シグナルを初期化
    std::uint32_t seed = (unsigned)time(NULL); // RNG シード値を取得
    oneapi::mkl::rng::mcg31m1 engine(Q, seed); // RNG エンジンを初期化
                                               // RNG 分布を設定
    oneapi::mkl::rng::uniform<float, oneapi::mkl::rng::uniform_method::standard>
      rng_distribution(-0.00005, 0.00005);

    oneapi::mkl::rng::generate(rng_distribution, engine, N, sig1_buf); //ノイズ
    oneapi::mkl::rng::generate(rng_distribution, engine, N, sig2_buf);
```

新しいスコープが開かれ、相関結果に対し `corr_buf` バッファーが宣言されていることに注目してください。このバッファーがスコープ外になると、ホストで `corr` が更新されます。

上記の例と同様に、人工的なシグナルはそれぞれのバッファーの両端に配置されます。

```
    Q.submit([&](sycl::handler &h) {
      sycl::accessor sig1_acc{sig1_buf, h, sycl::write_only};
      sycl::accessor sig2_acc{sig2_buf, h, sycl::write_only};
      h.single_task<>([=]() {
        sig1_acc[N - N / 4 - 1] = 1.0;
        sig1_acc[N - N / 4] = 1.0;
        sig1_acc[N - N / 4 + 1] = 1.0; // シグナル
        sig2_acc[N / 4 - 1] = 1.0;
        sig2_acc[N / 4] = 1.0;
        sig2_acc[N / 4 + 1] = 1.0;
      });
    }); // シグナル初期化の終わり
```

シグナルが準備できたら、oneMKL の DFT 関数で変換します。

```
// FFT 記述子を初期化
oneapi::mkl::dft::descriptor<oneapi::mkl::dft::precision::SINGLE,
oneapi::mkl::dft::domain::REAL>
transform_plan(N);
transform_plan.commit(Q);

// 配列で順方向変換を実行
oneapi::mkl::dft::compute_forward(transform_plan, sig1_buf);
oneapi::mkl::dft::compute_forward(transform_plan, sig2_buf);
```

単精度の実数から複素数への順方向変換が SYCL* キューにコミットされ、両方のバッファーデータに対し、インプレース DFT が実行されます。`DFT (sig1)` の結果に `CONJG(DFT(sig2))` を乗算します。これは手動で記述されたカーネルで実行できます。

```
Q.submit([&](sycl::handler &h)
{
   sycl::accessor sig1_acc{sig1_buf, h, sycl::read_only};
   sycl::accessor sig2_acc{sig2_buf, h, sycl::read_only};
   sycl::accessor corr_acc{corr_buf, h, sycl::write_only};

   h.parallel_for<>(sycl::range<1>{N/2}, [=](auto idx)
   {
      corr_acc[idx*2+0] = sig1_acc[idx*2+0] * sig2_acc[idx*2+0] +
                          sig1_acc[idx*2+1] * sig2_acc[idx*2+1];
      corr_acc[idx*2+1] = sig1_acc[idx*2+1] * sig2_acc[idx*2+0] -
                          sig1_acc[idx*2+0] * sig2_acc[idx*2+1];
   });
});
```

ただし、この実装では、クロスアーキテクチャーの最適なパフォーマンスを得られる可能性はほとんどありません。幸い、このステップには oneMKL 関数 `oneapi::mkl::vm::mulbyconj` が利用できます。mulbyconj (英語) 関数は、`std::complex<float>` の入力を想定しますが、バッファーは `float` データタイプで初期化されています。順方向変換後に複素数データが含まれている場合、バッファーを再度キャストする必要があります。

```
auto sig1_buf_cplx =
sig1_buf.template reinterpret<std::complex<float>, 1>((N + 2) / 2);
auto sig2_buf_cplx =
sig2_buf.template reinterpret<std::complex<float>, 1>((N + 2) / 2);
auto corr_buf_cplx =
corr_buf.template reinterpret<std::complex<float>, 1>((N + 2) / 2);
oneapi::mkl::vm::mulbyconj(Q, N / 2, sig1_buf_cplx, sig2_buf_cplx,
corr_buf_cplx);
```

IDFT ステップは計算を完了します。

```
// 複素数相関配列で逆交換を実行
oneapi::mkl::dft::compute_backward(transform_plan, corr_buf);
```

この例の始めで開かれたスコープが閉じられると、相関結果を保持するバッファーはスコープ外となり、ホストコンテナが強制的に更新されます。これは、ホストとデバイス間の唯一のデータ転送です。

明示的なバッファリングを使用したフーリエ相関の実装を以下に示します。

```
1.  #include <CL/sycl.hpp>
2.  #include <iostream>
3.  #include <mkl.h>
4.  #include <oneapi/mkl/dfti.hpp>
```

```
#include <oneapi/mkl/rng.hpp>
#include <oneapi/mkl/vm.hpp>

int main(int argc, char **argv) {
  unsigned int N = (argc == 1) ? 32 : std::stoi(argv[1]);
  if ((N % 2) != 0)
    N++;
  if (N < 32)
    N = 32;

  // SYCL* キューを初期化
  sycl::queue Q(sycl::default_selector_v);
  std::cout << "Running on: "
            << Q.get_device().get_info<sycl::info::device::name>() << std::endl;

  // シグナルデータのバッファーを作成。これはデバイスでのみ使用されます。
  sycl::buffer<float> sig1_buf{N + 2};
  sycl::buffer<float> sig2_buf{N + 2};

  // 相関結果を保持するコンテナを宣言
  // (デバイスで計算され、ホストで使用されます)
  std::vector<float> corr(N + 2);

  // 新しいスコープを開いて、相関結果の更新をトリガー
  {
    sycl::buffer<float> corr_buf(corr);

    // 入力シグナルを初期化
    std::uint32_t seed = (unsigned)time(NULL); // RNG シード値を取得
    oneapi::mkl::rng::mcg31m1 engine(Q, seed); // RNG エンジンを初期化
                                               // RNG 分布を設定
    oneapi::mkl::rng::uniform<float, oneapi::mkl::rng::uniform_method::standard>
        rng_distribution(-0.00005, 0.00005);

    oneapi::mkl::rng::generate(rng_distribution, engine, N, sig1_buf);//ノイズ
    oneapi::mkl::rng::generate(rng_distribution, engine, N, sig2_buf);

    Q.submit([&](sycl::handler &h) {
      sycl::accessor sig1_acc{sig1_buf, h, sycl::write_only};
      sycl::accessor sig2_acc{sig2_buf, h, sycl::write_only};
      h.single_task<>([=]() {
        sig1_acc[N - N / 4 - 1] = 1.0;
        sig1_acc[N - N / 4] = 1.0;
        sig1_acc[N - N / 4 + 1] = 1.0; // シグナル
        sig2_acc[N / 4 - 1] = 1.0;
        sig2_acc[N / 4] = 1.0;
        sig2_acc[N / 4 + 1] = 1.0;
      });
    }); // シグナル初期化の終わり

    clock_t start_time = clock(); // タイマーを開始

    // FFT 記述子の初期化
    oneapi::mkl::dft::descriptor<oneapi::mkl::dft::precision::SINGLE,
                                 oneapi::mkl::dft::domain::REAL>
        transform_plan(N);
    transform_plan.commit(Q);

    // 配列で順方向変換を実行
    oneapi::mkl::dft::compute_forward(transform_plan, sig1_buf);
    oneapi::mkl::dft::compute_forward(transform_plan, sig2_buf);

```

```
    // 計算: DFT(sig1) * CONJG(DFT(sig2))
    auto sig1_buf_cplx =
        sig1_buf.template reinterpret<std::complex<float>, 1>((N + 2) / 2);
    auto sig2_buf_cplx =
        sig2_buf.template reinterpret<std::complex<float>, 1>((N + 2) / 2);
    auto corr_buf_cplx =
        corr_buf.template reinterpret<std::complex<float>, 1>((N + 2) / 2);
    oneapi::mkl::vm::mulbyconj(Q, N / 2, sig1_buf_cplx, sig2_buf_cplx,
                               corr_buf_cplx);

    // 複素数相関配列で逆交換を実行
    oneapi::mkl::dft::compute_backward(transform_plan, corr_buf);

    clock_t end_time = clock(); // タイマーを停止
    std::cout << "The 1D correlation (N = " << N << ") took "
              << float(end_time - start_time) / CLOCKS_PER_SEC << " seconds."
              << std::endl;

  } // 相関結果を保持するバッファーがスコープ外になり、
    // ホストコンテナの更新が強制されました

  // 最大相関値を与えるシフトを検出
  float max_corr = 0.0;
  int shift = 0;
  for (unsigned int idx = 0; idx < N; idx++) {
    if (corr[idx] > max_corr) {
      max_corr = corr[idx];
      shift = idx;
    }
  }
  int _N = static_cast<int>(N);
  shift =
      (shift > _N / 2) ? shift - _N : shift; // シグナルを循環的に処理
                                             // 互いのバージョンをシフト
  std::cout << "Shift the second signal " << shift
            << " elements relative to the first signal to get a maximum, "
               "normalized correlation score of "
            << max_corr / N << "."<< std::endl;
}
```

明示的なバッファリングと比較するため、統合共有メモリー (USM) を使用してフーリエ相関アルゴリズムを再実装します。2 つの実装の違いがハイライト表示されています。最初に、シグナルと相関配列が USM に割り当てられ、次に人工的なデータで初期化されます。

```
// シグナルと相関配列を初期化
  auto sig1 = sycl::malloc_shared<float>(N + 2, sycl_device, sycl_context);
  auto sig2 = sycl::malloc_shared<float>(N + 2, sycl_device, sycl_context);
  auto corr = sycl::malloc_shared<float>(N + 2, sycl_device, sycl_context);

  // 入力シグナルを初期化
  std::uint32_t seed = (unsigned)time(NULL); // RNG シード値を取得
  oneapi::mkl::rng::mcg31m1 engine(Q, seed); // RNG エンジンを初期化
                                             // RNG 分布を設定
  oneapi::mkl::rng::uniform<float, oneapi::mkl::rng::uniform_method::standard>
      rng_distribution(-0.00005, 0.00005);

  // 警告: この文はデバイスで実行されます。
  auto evt1 =
      oneapi::mkl::rng::generate(rng_distribution, engine, N, sig1); // ノイズ
  auto evt2 = oneapi::mkl::rng::generate(rng_distribution, engine, N, sig2);
```

```
  evt1.wait();
  evt2.wait();

  // 警告: これらの文はホスト上で実行されるため、sig1 と sig2 を
  // デバイス上で更新する必要があります.
  sig1[N - N / 4 - 1] = 1.0;
  sig1[N - N / 4] = 1.0;
  sig1[N - N / 4 + 1] = 1.0; // シグナル
  sig2[N / 4 - 1] = 1.0;
  sig2[N / 4] = 1.0;
  sig2[N / 4 + 1] = 1.0;
```

USM ポインターが SYCL* バッファーではなく oneMKL 関数に渡されることを除いて、残りの実装はほとんど同一です。

```
  // 配列で順方向変換を実行
  evt1 = oneapi::mkl::dft::compute_forward(transform_plan, sig1);
  evt2 = oneapi::mkl::dft::compute_forward(transform_plan, sig2);

  // 計算: DFT(sig1) * CONJG(DFT(sig2))
  oneapi::mkl::vm::mulbyconj(
      Q, N / 2, reinterpret_cast<std::complex<float> *>(sig1),
      reinterpret_cast<std::complex<float> *>(sig2),
      reinterpret_cast<std::complex<float> *>(corr), {evt1, evt2})
      .wait();

  // 複素数相関配列で逆交換を実行
  oneapi::mkl::dft::compute_backward(transform_plan, corr).wait();
```

割り当てたメモリーを解放することも重要です。

```
sycl::free(sig1, sycl_context);
sycl::free(sig2, sycl_context);
sycl::free(corr, sycl_context);
```

USM 実装には、見慣れた構文が多くあります。また、SYCL* ランタイムによって処理される暗黙のデータ転送に依存するため、概念も単純です。ただし、プログラマーによる誤りはパフォーマンスを低下させます。

次のコードの警告メッセージに注目してください。oneMKL 乱数生成エンジンはデバイスで初期化されるため、sig1 と sig2 はデバイスでランダムノイズを使用して初期化されます。人工的なシグナルを追加するコードはホストで実行されることから、SYCL* ランタイムはホストとデバイス間でデータの一貫性を維持する必要があります。フーリエ相関で使用されるシグナルは、3D イメージのアプリケーションでは特に巨大化するため、不要なデータ転送はパフォーマンスの低下につながります。

デバイス上でシグナルデータを直接更新すると、データの一貫性が維持されるため不要なデータ転送を回避できます。

```
Q.single_task<>([=]() {
sig1[N - N / 4 - 1] = 1.0;
sig1[N - N / 4] = 1.0;
sig1[N - N / 4 + 1] = 1.0; // シグナル
sig2[N / 4 - 1] = 1.0;
sig2[N / 4] = 1.0;
sig2[N / 4 + 1] = 1.0;
}).wait();
```

明示的なバッファリングと USM の実装は同等のパフォーマンスを達成でき、(プログラマーがデータの一貫性を考慮していれば) SYCL* ランタイムが不要なデータ転送を回避できることを示しています。

USM を使用したフーリエ相関の実装を以下に示します。

```
#include <CL/sycl.hpp>
#include <iostream>
#include <mkl.h>
#include <oneapi/mkl/dfti.hpp>
#include <oneapi/mkl/rng.hpp>
#include <oneapi/mkl/vm.hpp>

int main(int argc, char **argv) {
  unsigned int N = (argc == 1) ? 32 : std::stoi(argv[1]);
  if ((N % 2) != 0)
    N++;
  if (N < 32)
    N = 32;

  // SYCL* キューを初期化
  sycl::queue Q(sycl::default_selector_v);
  auto sycl_device = Q.get_device();
  auto sycl_context = Q.get_context();
  std::cout << "Running on: "
            << Q.get_device().get_info<sycl::info::device::name>() << std::endl;

  // シグナルと相関配列を初期化
  auto sig1 = sycl::malloc_shared<float>(N + 2, sycl_device, sycl_context);
  auto sig2 = sycl::malloc_shared<float>(N + 2, sycl_device, sycl_context);
  auto corr = sycl::malloc_shared<float>(N + 2, sycl_device, sycl_context);

  // 入力シグナルを初期化
  std::uint32_t seed = (unsigned)time(NULL); // RNG シード値を取得
  oneapi::mkl::rng::mcg31m1 engine(Q, seed); // RNG エンジンを初期化
                                             // RNG 分布を設定
  oneapi::mkl::rng::uniform<float,
   oneapi::mkl::rng::uniform_method::standard>
       rng_distribution(-0.00005, 0.00005);

  auto evt1 =
       oneapi::mkl::rng::generate(rng_distribution, engine, N, sig1); // ノイズ
  auto evt2 = oneapi::mkl::rng::generate(rng_distribution, engine, N, sig2);
  evt1.wait();
  evt2.wait();

  Q.single_task<>([=]() {
     sig1[N - N / 4 - 1] = 1.0;
     sig1[N - N / 4] = 1.0;
     sig1[N - N / 4 + 1] = 1.0; // シグナル
     sig2[N / 4 - 1] = 1.0;
     sig2[N / 4] = 1.0;
     sig2[N / 4 + 1] = 1.0;
   }).wait();

  clock_t start_time = clock(); // タイマーを開始

  // FFT 記述子の初期化
  oneapi::mkl::dft::descriptor<oneapi::mkl::dft::precision::SINGLE,
                               oneapi::mkl::dft::domain::REAL>
      transform_plan(N);
  transform_plan.commit(Q);

  // 配列で順方向変換を実行
  evt1 = oneapi::mkl::dft::compute_forward(transform_plan, sig1);
  evt2 = oneapi::mkl::dft::compute_forward(transform_plan, sig2);

```

```
61.   // 計算: DFT(sig1) * CONJG(DFT(sig2))
62.   oneapi::mkl::vm::mulbyconj(
63.       Q, N / 2, reinterpret_cast<std::complex<float> *>(sig1),
64.       reinterpret_cast<std::complex<float> *>(sig2),
65.       reinterpret_cast<std::complex<float> *>(corr), {evt1, evt2})
66.       .wait();
67.
68.   // 複素数相関配列で逆変換を実行
69.   oneapi::mkl::dft::compute_backward(transform_plan, corr).wait();
70.
71.   clock_t end_time = clock(); // タイマーを停止
72.   std::cout << "The 1D correlation (N = " << N << ") took "
73.             << float(end_time - start_time) / CLOCKS_PER_SEC << " seconds."
74.             << std::endl;
75.
76.   // 最大相関値を与えるシフトを検出
77.   float max_corr = 0.0;
78.   int shift = 0;
79.   for (unsigned int idx = 0; idx < N; idx++) {
80.     if (corr[idx] > max_corr) {
81.       max_corr = corr[idx];
82.       shift = idx;
83.     }
84.   }
85.   int _N = static_cast<int>(N);
86.   shift =
87.       (shift > _N / 2) ? shift - _N : shift; // シグナルを循環的に処理
88.                                              // 互いのバージョンをシフト
89.   std::cout << "Shift the second signal " << shift
90.             << " elements relative to the first signal to get a maximum, "
91.                "normalized correlation score of "
92.             << max_corr / N << "."<< std::endl;
93.
94.   // クリーンアップ
95.   sycl::free(sig1, sycl_context);
96.   sycl::free(sig2, sycl_context);
97.   sycl::free(corr, sycl_context);
98. }
```

最大相関値の場所を特定する最後のステップは、ホストで実行されることに注意してください。入力データが大量にある場合、この計算をデバイスで実行することを推奨します。幸いなことに、`maxloc` リダクションは SYCL* で実装できる汎用的な並列パターンです。これは皆さんへの演習として残しておきますが、『データ並列 C++』(英語) の図 14-11 に役立つ例が示されています。

# ホスト/デバイスメモリー、バッファー、および USM

アクセラレーターは、いくつかのメモリー階層にアクセスできます。最高のパフォーマンスを得るには、階層内の適切なレベルを使用することが重要です。

ここでは、メモリー階層の宣言、移動、およびアクセスに関連するトピックについて説明します。


- 統合共有メモリーの割り当て
- USM とバッファーのパフォーマンスへの影響
- ホストとデバイス間でデータ移動の往復を避ける
- ループ内でのバッファー宣言を避ける
- バッファー・アクセサー・モード


## 統合共有メモリーの割り当て

統合共有メモリー (USM) を利用すると、プログラムは C/C++ ポインターを使用してメモリーにアクセスできます。SYCL* でメモリーを割り当てるには、3 つの方法があります。

**malloc_device:**

- 割り当てられたメモリーは、指定されたデバイスからのみアクセスできますが、コンテキスト内のほかのデバイスやホストからはアクセスできません。
- このデータは常にデバイス上にあるため、カーネル実行の最速の選択肢です。
- コンテキスト内のホストやほかのデバイスにデータを転送するには、明示的なコピー操作が必要です。

**malloc_host:**

- 割り当てられたメモリーは、ホストとコンテキスト内のほかのデバイスからアクセスできます。
- データは常にホスト上にあり、デバイスからは PCIe* 経由でアクセスされます。
- ホストとデバイス間のデータを同期するのに明示的なコピー操作は必要ありません。

**malloc_shared:**

- 割り当てられたメモリーのアクセスは、ホストと指定されたデバイスのみが許可されます。
- ホストとデバイス間でデータを移行して (ゼロレベルドライバーによって行われる)、アクセスを高速化できます。
- ホストとデバイス間の同期には明示的なコピー操作は必要ありませんが、コンテキスト内のデバイスには必要です。

3 つのメモリー割り当てとその特徴を以下にまとめます。

**メモリー割り当てタイプと特徴**

| メモリー割り当てタイプ | 説明 | ホストからアクセス可能 | デバイスからアクセス可能 | 場所 |
|---|---|---|---|---|
| host | ホストメモリーの割り当て | はい | はい<br><br>PCIe* またはファブリック・リンクを介してリモートアクセス可能 | ホスト |
| device | デバイスメモリーの割り当て | いいえ | はい | デバイス |
| shared | ホストとデバイス間で共有割り当て | はい | はい | ホストとデバイス間で動的に移行 |

マルチスタック、マルチ C スライス GPU 環境では、デバイスと共有される USM 割り当てがルートデバイスに関連付けられていることに注意してください。したがって、同じデバイス上のすべてのスタックと C スライスからアクセスできます。混乱を避けるため、プログラムは `malloc_device` と `malloc_shared` 割り当てにルートデバイスを使用する必要があります。

## OpenMP* USM 割り当て API

SYCL* USM モデルに合わせて、開発者がアプリケーション、メモリーサイズ、およびパフォーマンス要件に基づいてメモリー割り当てを行えるよう、インテル拡張機能として 3 つの新しい OpenMP* API が追加されています。それらのセマンティクスとパフォーマンス特性については、次の節で詳しく説明します。

## ホストメモリー割り当て

ホスト割り当ては、ホストとデバイスの間でデータを手動でコピーする必要がないため、デバイス割り当てよりも容易に利用できます。ホスト割り当ては、ホストとデバイスの両方からアクセス可能なホストメモリーへの割り当てです。この割り当ては、デバイスからアクセスできますが、デバイスに接続されたメモリーに移行することはできません。代わりに、このメモリーに対する読み書きを行うオフロード領域は、PCIe* バスまたはファブリック・リンクを介してリモートで実行します。これには利便性とパフォーマンスのトレードオフがあります。ホスト割り当てによって生じるアクセスコストは高くなりますが、ホスト割り当てを利用する利点はほかにもあります。例えば、ほとんどアクセスされないデータや、デバイスに接続されたメモリーに収まらない大きなデータセットが含まれます。ホストメモリー割り当てを要求する API を次に示します。

```
extern void *omp_target_alloc_host(size_t size, int device_num)
```

## デバイスメモリー割り当て

この割り当ては、GDDR/DDR やデバイス上の HBM など、デバイスに接続されたメモリーへのポインターアクセスを利用する際に必要となります。デバイスに割り当てられたメモリーは、デバイス上で実行される領域をオフロードすることで読み書きできますが、ホスト上で実行されるコードからは直接アクセスすることはできません。ホスト上のデバイス割り当てにアクセスすると、誤ったデータを取得したり、プログラムがクラッシュすることがあります。デバイスメモリー割り当てを要求する API を次に示します。

```
extern void *omp_target_alloc_device(size_t size, int device_num)
```

## 共有メモリー割り当て

ホスト割り当てと同様に、共有割り当ては、ホストとデバイスの両方からアクセスできます。ホスト割り当てとの違いは、共有割り当てでは、ホストメモリーとデバイスに接続されたメモリー間で、プログラマーからは透過に自動でデータを移行できることです。割り当てがデバイスに移行されると、デバイスにアクセスするオフロード領域は、ホストからリモートでアクセスするよりも高いパフォーマンスで実行されます。ただし、共有割り当てには、ページの移行コストやピンポン効果などのコストが生じるため、すべてのケースで利点が得られるわけではありません。

```
extern void *omp_target_alloc_shared(size_t size, int device_num)
```

## omp_target_alloc API の USM サポート

ターゲットメモリー割り当て用の OpenMP* API は、以下のようになります。

```
extern void *omp_target_alloc_device(size_t size, int device_num)
```

## USM とバッファーのパフォーマンスへの影響

SYCL* は、デバイスのメモリーを管理するいくつかの選択肢を提供します。ここでは、パフォーマンスのトレードオフについて説明します。始めに概念を簡単に紹介します。詳しい説明は、『データ並列 C++』(英語) を参照してください。

他の言語と同様に動作は仕様によって定義されますが、実装は定義されないため、パフォーマンス特性はソフトウェアのバージョンとデバイスで異なる可能性があります。これについて考えてみます。

**バッファー:** バッファーはデバイスとホストからアクセスできるデータのコンテナです。SYCL* ランタイムは、メモリーの割り当て、読み取り、書き込み向けの API を提供することでメモリーを管理します。ランタイムは、ホストとデバイス間でデータを移動し、データアクセスを同期する役割を持ちます。

**統合共有メモリー (USM):** データへのアクセスが API でのみ行われるバッファーとは異なり、USM では従来の C/C++ ポインターを使用してデータを読み書きできます。USM には 2 つのバリアントがあります。デバイスに割り当てられたメモリーにはデバイスからのみアクセスできるため、ホストとデバイス間で明示的にデータを移動する必要があります。**共有割り当て**は、ランタイムが自動的にメモリーを移動することで、デバイスまたはホストから参照できます。

3 つのモデルのモデルで作成された同じサンプルプログラムを比較して、選択肢のトレードオフを確認します。問題を解決するため、GPU とホストが連携して計算するプログラムを使用するにはデータを送受信する必要があります。

最初にシリアル計算から見ていきましょう。7 行目のループを GPU で実行し、11 行目のループを CPU で実行すると考えます。両方のループでデータ配列の読み書きを行うため、6 行目のループ反復ごとにホストと GPU 間でデータを移動する必要があります。

```
1.  void serial(int stride) {
2.  // data の割り当てと初期化
3.  float *data = new float[data_size];
4.  init(data);
5.  timer it;
6.  for (int i = 0; i < time_steps; i++) {
7.    for (int j = 0; j < data_size; j++) {
8.      for (int k = 0; k < device_steps; k++)
9.        data[j] += 1.0;
10.   }
11.   for (int j = 0; j < data_size; j += stride)
12.     data[j] += 1.0;
13. }
14. put_elapsed_time(it);
15. check(data);
16. delete[] data;
17. } // シリアル
```

## バッファー

次に、バッファーを使用してデータを管理する計算を示します。3 行目でバッファーを作成して `init` 関数で初期化します。`init` 関数はここでは示されませんが、アクセサーまたはポインターを受け入れます。`parallel_for` は 11 行目で定義されるカーネルを実行します。カーネルは、`device_data` アクセサーを使用して `buffer_data` のデータを読み書きしますが、ここでは、コードがデータの位置を指定していないことに注目してください。アクセサーはデータが必要なタイミングと位置を明示し、SYCL* ランタイムがデータをデバイスに移動して (必要であれば)、カーネルを起動します。16 行目の `host_accessor` は、データがホスト上で読み取り/書き込みされることを示します。カーネルは `buffer_data` も読み取り/書き込みするため、`host_accessor` コンストラクターはカーネルが完了するのを待機して、データをホストに移動した後、20 行目で読み取り/書き込みを実行します。ループの次の反復では、9 行目のアクセサー・コンストラクターがデバイスにデータが戻されるまで待機します。これにより、カーネルの起動が遅延します。

```
1.  void buffer_data(int stride) {
2.  // ホストでバッファーを割り当てて初期化
3.  sycl::buffer<float> buffer_data{data_size};
4.  init(sycl::host_accessor(buffer_data, sycl::write_only, sycl::no_init));
5.  timer it;
6.  for (int i = 0; i < time_steps; i++) {
7.    // デバイスで計算
8.    q.submit([&](auto &h) {
9.      sycl::accessor device_data(buffer_data, h);
10.     auto compute = [=](auto id) {
11.     for (int k = 0; k < device_steps; k++)
12.       device_data[id] += 1.0;
13.   };
14.   h.parallel_for(data_size, compute); });
15.   // ホストで計算
16.   sycl::host_accessor host_data(buffer_data);
17.   for (int i = 0; i < data_size; i += stride)
18.     host_data[i] += 1.0; }
19.   put_elapsed_time(it);
20.   const sycl::host_accessor h(buffer_data);
21.   check(h);
22. } // buffer_data
```

## パフォーマンス上の考察

11 行目と 17 行目のデータアクセスは単純な配列参照のように見えますが、C++ オペレーターのオーバーロードと SYCL* ランタイムによって実装されています。アクセサーによる配列参照の効率は実装によって異なりますが、デバイスコードはダイレクトメモリー参照と比較してもオーバーロードに対するコストは生じません。ランタイムは、バッファーのどのデータがアクセスされるか事前に認識できないため、カーネルが開始される前にすべてのデータがデバイス上に存在することを確認する必要があります。これは、今後変わる可能性があります。

同様のことが `host_accessor` にも当てはまります。ランタイムはすべてのデータをホストに移動するわけではありません。配列参照は複雑なコードで実装されており、ネイティブ C++ による配列参照に比べるとかなり低速です。少量のデータ参照は許容されますが、`host_accessor` を使用する計算量の多いアルゴリズムは、パフォーマンスの観点から回避する必要があります。

もう 1 つの問題は並列性です。`host_accessor` は、アクセサーがデータの読み取りと書き込みにあまり使用されていなくても、同じバッファーを参照するカーネルの起動をブロックできます。`host_accessor` を含むスコープを可能な限り最小限にします。この例では、4 行目のホストアクセサーは `init` 関数からリターンした後に破棄され、20 行目のホストアクセサーはそれぞれのループ反復が終了すると破棄されます。

## 共有割り当て

次に、共有割り当てにより実装される同様のアルゴリズムを示します。データは 2 行目で割り当てられます。USM に割り当てられるデータは、従来のポインターでアクセスできるためアクセサーを必要としません。したがって、8 行目と 12 行目の配列参照は、単純なインデックス操作で実装できます。10 行目の `parallel_for` は、ホストが 12 行目でデータにアクセスする前にカーネルが完了するのを待機して終了します。バッファーと同様に、SYCL* ランタイムはカーネルを起動する前にすべてのデータがデバイス上に存在することを確認します。また、共有割り当ては参照されない限りホストにコピーされません。ホストで最初にデータを参照すると、オペレーティング・システムのページフォルトが発生して、データページがデバイスからホストにコピーされ実行が続けられます。後続のコードが同じページのデータを参照する場合、遅延なくアクセスできます。カーネルが起動されると、ホスト上の常駐ページはすべてデバイスにフラッシュバックされます。

```
1.  void shared_usm_data(int stride) {
2.    float *data = sycl::malloc_shared<float>(data_size, q);
3.    init(data);
4.    timer it;
5.    for (int i = 0; i < time_steps; i++) {
6.      auto compute = [=](auto id) {
7.        for (int k = 0; k < device_steps; k++)
8.          data[id] += 1.0;
9.      };
10.   q.parallel_for(data_size, compute).wait();
11.   for (int k = 0; k < data_size; k += stride)
12.     data[k] += 1.0;
13.   }
14.   q.wait();
15.   put_elapsed_time(it);
16.   check(data);
17.   sycl::free(data, q);
18. } // shared_usm_data
```

## パフォーマンス上の考察

バッファーと比較すると、データ参照は単純なポインターであり、良好なパフォーマンスを提供します。ただし、データをホストに転送するためページフォルトが処理されると、データ転送のコストに加えオーバーヘッドが発生します。アプリケーションへの影響は参照パターンによって異なります。スパース・ランダム・アクセスはオーバーヘッドが最も高く、データの線形スキャンはページフォルトの影響が低いです。

すべての同期はプログラマーによる明示的な制御下にあるため、うまく設計されたプログラムでは同時実行性は問題になりません。

## デバイス割り当て

デバイス割り当てを使用するプログラムを以下に示します。デバイスに割り当てられたデータは、デバイス上でのみ直接アクセス可能であり、17 行目にあるようにホストに明示的にコピーする必要があります。デバイスとホスト間の同期はすべて明示的です。17 行目は待機してから終了するため、非同期コピーが完了するまでホストコードの実行は開始されません。キュー定義は示されていませんが、インオーダーでキューを使用するため 17 行目の `memcpy` は 15 行目の `parallel_for` が完了するのを待機します。

```
1.  void device_usm_data(int stride) {
2.    // ホストデータの割り当てと初期化
3.    float *host_data = new float[data_size];
4.    init(host_data);
5.    // デバイスデータを割り当て
6.    float *device_data = sycl::malloc_device<float>(data_size, q);
```

```
7.    timer it;
8.    for (int i = 0; i < time_steps; i++) {
9.    // デバイスへデータをコピーして計算
10.   q.memcpy(device_data, host_data, sizeof(float) * data_size);
11.   auto compute = [=](auto id) {
12.     for (int k = 0; k < device_steps; k++)
13.       device_data[id] += 1.0;
14.   };
15.   q.parallel_for(data_size, compute);
16.   // ホストへデータをコピーして計算
17.   q.memcpy(host_data, device_data, sizeof(float) * data_size).wait();
18.   for (int k = 0; k < data_size; k += stride)
19.     host_data[k] += 1.0;
20.   }
21.   q.wait();
22.   put_elapsed_time(it);
23.   check(host_data);
24.   sycl::free(device_data, q);
25.   delete[] host_data;
26. } // device_usm_data
```

### パフォーマンス上の考察

データの移動と同期はどちらも明示的であり、プログラマーの完全な制御下にあります。配列参照は、ホスト上で行われるため、共有割り当てのページフォルトおよびバッファーに関連するオーバーロードのオーバーヘッドもありません。共有割り当ては、ホストが実際に参照するデータだけをメモリーページ単位で転送します。理論的には、デバイス割り当ては任意の粒度によるオンデマンドの移動を可能にします。データを細粒度で非同期移動することは可能ですが、複雑になる可能性があるため大部分のプログラマーはデータ構造全体を一度だけ転送して済ませます。明示的なデータ移動と同期によりコードは複雑になりますが、デバイス割り当てにより高いパフォーマンスを得ることができます。

## ホストとデバイス間でデータ移動の往復を避ける

特にディスクリート・アクセラレーター・デバイスでは、ホストとデバイス間でデータを移動するコストは高くなります。そのため、ホストとデバイス間のデータ転送は可能な限り回避することが重要です。状況によっては、アクセラレーター上のカーネルによって計算されたデータをホストに移動して、何らかの操作を行い、処理を続行するためデバイスに送り返すことが必要な場合があります。そのような場合、デバイスからホストへの転送、およびホストからデバイスへの転送コストが生じることになります。

次の例を考えてみます。ここでは、カーネルが何らかの操作 (この場合ベクトル加算) により新しいベクトルに結果を生成します。次に、このベクトルは、それぞれの値に関数を適用することで別のベクトルに変換され、追加の計算を行うため別のカーネルの入力として供給されます。このような計算は非常に一般的であり、多くの分野で使用されます。ここでは、アルゴリズムは反復的で、1 つの計算の出力を別の計算の入力として供給する必要があります。典型的な例として、計算レイヤーとして構造化されるマシンラーニング・モデルがありますが、この場合 1 つのレイヤーの出力が次のレイヤーの入力となります。

```
1.  double myFunc1(sycl::queue &q, AlignedVector<int> &a, AlignedVector<int> &b,
2.                 AlignedVector<int> &c, AlignedVector<int> &d,
3.                 AlignedVector<int> &res, int iter) {
4.    sycl::range num_items{a.size()};
5.    VectorAllocator<int> alloc;
6.    AlignedVector<int> sum(a.size(), alloc);
7.
8.    const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
9.    sycl::buffer a_buf(a, props);
10.   sycl::buffer b_buf(b, props);
11.   sycl::buffer c_buf(b, props);
```

```
  sycl::buffer d_buf(b, props);
  sycl::buffer res_buf(res, props);
  sycl::buffer sum_buf(sum.data(), num_items, props);

  Timer timer;
  for (int i = 0; i < iter; i++) {
    // カーネル 1
    q.submit([&](auto &h) {
      // 入力アクセサー
      sycl::accessor a_acc(a_buf, h, sycl::read_only);
      sycl::accessor b_acc(b_buf, h, sycl::read_only);
      // 出力アクセサー
      sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

      h.parallel_for(num_items,
                     [=](auto id) { sum_acc[id] = a_acc[id] + b_acc[id]; });
    });

    {
      sycl::host_accessor h_acc(sum_buf);
      for (int j = 0; j < a.size(); j++)
        if (h_acc[j] > 10)
          h_acc[j] = 1;
        else
          h_acc[j] = 0;
    }

    // カーネル 2
    q.submit([&](auto &h) {
      // 入力アクセサー
      sycl::accessor sum_acc(sum_buf, h, sycl::read_only);
      sycl::accessor c_acc(c_buf, h, sycl::read_only);
      sycl::accessor d_acc(d_buf, h, sycl::read_only);
      // 出力アクセサー
      sycl::accessor res_acc(res_buf, h, sycl::write_only, sycl::no_init);

      h.parallel_for(num_items, [=](auto id) {
        res_acc[id] = sum_acc[id] * c_acc[id] + d_acc[id];
      });
    });
    q.wait();
  }
  double elapsed = timer.Elapsed() / iter;
  return (elapsed);
} // myFunc1 の終わり
```

データをホストに移動して関数でそのデータを処理し、デバイス上の次のカーネルにデータを送る代わりに、**カーネル 3** を作成してデバイス自身でこの関数を実行できます。この手法を次の例に示します。これは、機能的には前のコードと同じです。ここでは、**カーネル 1** と**カーネル 2** の間の `accum_buf` の中間データを操作する、第 3 のカーネルである**カーネル 3** について説明します。これには、デバイスからホストへのデータの送り戻しを回避できる利点があります。

```
double myFunc2(sycl::queue &q, AlignedVector<int> &a, AlignedVector<int> &b,
               AlignedVector<int> &c, AlignedVector<int> &d,
               AlignedVector<int> &res, int iter) {
  sycl::range num_items{a.size()};
  VectorAllocator<int> alloc;
  AlignedVector<int> sum(a.size(), alloc);

  const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
  sycl::buffer a_buf(a, props);
  sycl::buffer b_buf(b, props);
  sycl::buffer c_buf(b, props);
  sycl::buffer d_buf(b, props);
```

```
  sycl::buffer res_buf(res, props);
  sycl::buffer sum_buf(sum.data(), num_items, props);

  Timer timer;
  for (int i = 0; i < iter; i++) {
    // カーネル 1
    q.submit([&](auto &h) {
      // 入力アクセサー
      sycl::accessor a_acc(a_buf, h, sycl::read_only);
      sycl::accessor b_acc(b_buf, h, sycl::read_only);
      // 出力アクセサー
      sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

      h.parallel_for(num_items,
                     [=](auto i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
    });

    // カーネル 3
    q.submit([&](auto &h) {
      sycl::accessor sum_acc(sum_buf, h, sycl::read_write);
      h.parallel_for(num_items, [=](auto id) {
        if (sum_acc[id] > 10)
          sum_acc[id] = 1;
        else
          sum_acc[id] = 0;
      });
    });

    // カーネル 2
    q.submit([&](auto &h) {
      // 入力アクセサー
      sycl::accessor sum_acc(sum_buf, h, sycl::read_only);
      sycl::accessor c_acc(c_buf, h, sycl::read_only);
      sycl::accessor d_acc(d_buf, h, sycl::read_only);
      // 出力アクセサー
      sycl::accessor res_acc(res_buf, h, sycl::write_only, sycl::no_init);

      h.parallel_for(num_items, [=](auto i) {
        res_acc[i] = sum_acc[i] * c_acc[i] + d_acc[i];
      });
    });
    q.wait();
  }
  double elapsed = timer.Elapsed() / iter;
  return (elapsed);
} // myFunc2 の終わり
```

この例を最適化する方法はほかにもあります。例えば、**カーネル 3** のクリッピング操作を次のように**カーネル 1** の計算にマージします。これはカーネルの融合と呼ばれ、3 番目のカーネルを起動しない利点もあります。SYCL* コンパイラーは、このような最適化を行いません。マシンラーニングなど特定のドメインには、ML モデルを操作して同じ影響を及ぼす操作を融合するグラフ・コンパイラーがあります。

```
double myFunc3(sycl::queue &q, AlignedVector<int> &a, AlignedVector<int> &b,
               AlignedVector<int> &c, AlignedVector<int> &d,
               AlignedVector<int> &res, int iter) {
  sycl::range num_items{a.size()};
  VectorAllocator<int> alloc;
  AlignedVector<int> sum(a.size(), alloc);

  const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
  sycl::buffer a_buf(a, props);
  sycl::buffer b_buf(b, props);
  sycl::buffer c_buf(b, props);
```

```
  sycl::buffer d_buf(b, props);
  sycl::buffer res_buf(res, props);
  sycl::buffer sum_buf(sum.data(), num_items, props);

  Timer timer;
  for (int i = 0; i < iter; i++) {
    // カーネル 1
    q.submit([&](auto &h) {
      // 入力アクセサー
      sycl::accessor a_acc(a_buf, h, sycl::read_only);
      sycl::accessor b_acc(b_buf, h, sycl::read_only);
      // 出力アクセサー
      sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

      h.parallel_for(num_items, [=](auto i) {
        int t = a_acc[i] + b_acc[i];
        if (t > 10)
          sum_acc[i] = 1;
        else
          sum_acc[i] = 0;
      });
    });

    // カーネル 2
    q.submit([&](auto &h) {
      // 入力アクセサー
      sycl::accessor sum_acc(sum_buf, h, sycl::read_only);
      sycl::accessor c_acc(c_buf, h, sycl::read_only);
      sycl::accessor d_acc(d_buf, h, sycl::read_only);
      // 出力アクセサー
      sycl::accessor res_acc(res_buf, h, sycl::write_only, sycl::no_init);

      h.parallel_for(num_items, [=](auto i) {
        res_acc[i] = sum_acc[i] * c_acc[i] + d_acc[i];
      });
    });
    q.wait();
  }
  double elapsed = timer.Elapsed() / iter;
  return (elapsed);
} // myFunc3 の終わり
```

次のコードに示すように、このカーネルの融合をさらに進めて**カーネル 1** と**カーネル 2** の両方を融合します。これにより、カーネルの起動が排除されることに加えて、中間 `accum_buf` のメモリーを排除することで高いパフォーマンスが得られます。この場合、パフォーマンス上の利点の大部分は、メモリー参照の局所性の改善によるものです。

```
double myFunc4(sycl::queue &q, AlignedVector<int> &a, AlignedVector<int> &b,
               AlignedVector<int> &c, AlignedVector<int> &d,
               AlignedVector<int> &res, int iter) {
  sycl::range num_items{a.size()};
  VectorAllocator<int> alloc;

  const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
  sycl::buffer a_buf(a, props);
  sycl::buffer b_buf(b, props);
  sycl::buffer c_buf(b, props);
  sycl::buffer d_buf(b, props);
  sycl::buffer res_buf(res, props);

  Timer timer;
  for (int i = 0; i < iter; i++) {
    // カーネル 1
    q.submit([&](auto &h) {
```

```
      // 入力アクセサー
      sycl::accessor a_acc(a_buf, h, sycl::read_only);
      sycl::accessor b_acc(b_buf, h, sycl::read_only);
      sycl::accessor c_acc(c_buf, h, sycl::read_only);
      sycl::accessor d_acc(d_buf, h, sycl::read_only);
      // 出力アクセサー
      sycl::accessor res_acc(res_buf, h, sycl::write_only, sycl::no_init);

      h.parallel_for(num_items, [=](auto i) {
        int t = a_acc[i] + b_acc[i];
        if (t > 10)
          res_acc[i] = c_acc[i] + d_acc[i];
        else
          res_acc[i] = d_acc[i];
      });
    });
    q.wait();
  }
  double elapsed = timer.Elapsed() / iter;
  return (elapsed);
} // myFunc4 の終わり
```

## ループ内でのバッファー宣言を避ける

カーネルが `for` ループ内で繰り返し起動される場合、ループ外部でバッファーを宣言することでバッファーの割り当てと解放の繰り返しを回避できます。ループ内でバッファーを宣言すると、ホストからデバイスへのメモリーコピーとデバイスからホストへのメモリーコピーが繰り返されます。

次の例では、`for` ループ内でカーネルが繰り返し起動されています。バッファー `CBuf` は一時配列として反復内で値を保持するために使用されていますが、1 回の反復で割り当てられた値は他の反復では使用されません。バッファー `CBuf` はループ内で宣言されているため、ループ反復ごとに割り当てと解放が行われます。バッファーの割り当てと解放に加えて、バッファーに関連付けられたメモリーが反復ごとにホストからデバイスへ、そしてデバイスからホストへ冗長的に転送されます。

```
#include <CL/sycl.hpp>
#include <stdio.h>

constexpr int N = 25;
constexpr int STEPS = 100000;

int main() {

  int AData[N];
  int BData[N];
  int CData[N];

  sycl::queue Q;

  // それぞれ N 個の整数を保持する 2 つのバッファーを作成
  sycl::buffer<int> ABuf(&AData[0], N);
  sycl::buffer<int> BBuf(&BData[0], N);

  Q.submit([&](auto &h) {
    // デバイスアクセサーを作成。
    // プロパティー no_init は、バッファーの以前の
    // 内容を破棄できることをランタイムに通知します。
    sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);
    sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);
    h.parallel_for(N, [=](auto i) {
      aA[i] = 10;
```

```
27.       aB[i] = 20;
28.     });
29.   });
30.
31.   for (int j = 0; j < STEPS; j++) {
32.     sycl::buffer<int> CBuf(&CData[0], N);
33.     Q.submit([&](auto &h) {
34.       // デバイスアクセサーを作成
35.       sycl::accessor aA(ABuf, h);
36.       sycl::accessor aB(BBuf, h);
37.       sycl::accessor aC(CBuf, h);
38.       h.parallel_for(N, [=](auto i) {
39.         aC[i] = (aA[i] < aB[i]) ? -1 : 1;
40.         aA[i] += aC[i];
41.         aB[i] -= aC[i];
42.       });
43.     });
44.   } // for の終わり
45.
46.   // ホストアクセサーを作成
47.   const sycl::host_accessor haA(ABuf);
48.   const sycl::host_accessor haB(BBuf);
49.   printf("%d %d\n", haA[N / 2], haB[N / 2]);
50.
51.   return 0;
52. }
```

より適切なアプローチは、`for` ループの前で `CBuf` を宣言することです。これにより、`CBuf` は一度だけ割り当てられループが終了するときに解放されます。さらに、ホストとデバイス間の冗長なデータ転送を排除できるためパフォーマンスが向上します。次のカーネルではこの変更を適用しています。

```
1.  #include <CL/sycl.hpp>
2.  #include <stdio.h>
3.
4.  constexpr int N = 25;
5.  constexpr int STEPS = 100000;
6.
7.  int main() {
8.
9.    int AData[N];
10.   int BData[N];
11.   int CData[N];
12.
13.   sycl::queue Q;
14.
15.   // それぞれ N 個の整数を保持する 3 つのバッファーを作成
16.   sycl::buffer<int> ABuf(&AData[0], N);
17.   sycl::buffer<int> BBuf(&BData[0], N);
18.   sycl::buffer<int> CBuf(&CData[0], N);
19.
20.   Q.submit([&](auto &h) {
21.     // デバイスアクセサーを作成。
22.     // プロパティー no_init は、バッファーの以前の
23.     // 内容を破棄できることをランタイムに通知します。
24.     sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);
25.     sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);
26.     h.parallel_for(N, [=](auto i) {
27.       aA[i] = 10;
28.       aB[i] = 20;
29.     });
30.   });
31.
32.   for (int j = 0; j < STEPS; j++) {
33.     Q.submit([&](auto &h) {
```

```
34.        // デバイスアクセサーを作成
35.        sycl::accessor aA(ABuf, h);
36.        sycl::accessor aB(BBuf, h);
37.        sycl::accessor aC(CBuf, h);
38.        h.parallel_for(N, [=](auto i) {
39.          aC[i] = (aA[i] < aB[i]) ? -1 : 1;
40.          aA[i] += aC[i];
41.          aB[i] -= aC[i];
42.        });
43.      });
44.    } // for の終わり
45.
46.    // ホストアクセサーを作成
47.    const sycl::host_accessor haA(ABuf);
48.    const sycl::host_accessor haB(BBuf);
49.    printf("%d %d\n", haA[N / 2], haB[N / 2]);
50.
51.    return 0;
52. }
```

## バッファー・アクセサー・モード

SYCL* では、バッファーはホストまたはデバイスからアクセスできるメモリーへの抽象化されたビューを提供します。バッファー・オブジェクトから直接バッファーにアクセスすることはできません。代わりに、バッファーデータにアクセスできるアクセサー・オブジェクトを作成します。

アクセスモードは、プログラム内でアクセサーに関連付けられたメモリーをどのように使用するかを示します。アクセサーのアクセスモードは、ランタイムがカーネルの実行順序を決定し、データの移動を行うために使用されます。これにより、プログラマーが意図した順番でカーネルを実行できるようになります。依存関係違反や競合状態が発生しない場合、ベースとなるハードウェアの機能に応じて、ランタイムはカーネルを同時に実行できます。

パフォーマンスを向上させるには、アクセサーのアクセスモードが、カーネルによる実行操作に対応していることを確認します。`read_only` として宣言されているアクセサーに書き込みを行うと、コンパイル時にエラーとなります。しかし、カーネルが書き込みを行わない場合、コンパイラーは `read_write` 形式で宣言されたアクセサーを `read` に変更することはありません。

次の例には 3 つのカーネルがあります。最初のカーネルは `ABuf`、`BBuf`、および `CBuf` を初期化するため、バッファーのアクセスモードを `write_only` としています。2 番目のカーネルは `ABuf` と `BBuf` を読み取り、`CBuf` を読み書きするため、`ABuf` と `BBuf` のアクセスモードを `read_only`、そして `CBuf` のアクセスモードを `read_write` に指定します。

`read_only` アクセスモードは、カーネルが実行を開始する前にデータがデバイスで利用可能でなければならないことをランタイムに通知しますが、計算が完了してもデータをデバイスからホストにコピーする必要はありません。

2 番目のカーネルで `ABuf` と `BBuf` が `read_only` でなく `read_write` である場合、デバイスでデータが更新されていなくてもカーネルの終了時に `ABuf` と `BBuf` に関連するメモリーがデバイスからホストにコピーされます。さらに、`read_write` にすると依存関係も生じるようになります。`ABuf` または `BBuf` を読み取る別のカーネルがこのブロック内で送信されると、新しいカーネルは 2 番目のカーネルが完了するまで実行を開始できません。

```
1.  constexpr int N = 100;
2.
3.  int main() {
4.
5.    int AData[N];
6.    int BData[N];
```

```
7.    int CData[N];
8.
9.    sycl::queue Q;
10.
11.   // カーネル 1
12.   {
13.     // それぞれ N 個の整数を保持する 3 つのバッファーを作成
14.     sycl::buffer<int> ABuf(&AData[0], N);
15.     sycl::buffer<int> BBuf(&BData[0], N);
16.     sycl::buffer<int> CBuf(&CData[0], N);
17.
18.     Q.submit([&](auto &h) {
19.       // デバイスアクセサーを作成。
20.       // プロパティー no_init は、バッファーの以前の
21.       // 内容を破棄できることをランタイムに通知します。
22.       sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);
23.       sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);
24.       sycl::accessor aC(CBuf, h, sycl::write_only, sycl::no_init);
25.
26.       h.parallel_for(N, [=](auto i) {
27.         aA[i] = 11;
28.         aB[i] = 22;
29.         aC[i] = 0;
30.       });
31.     });
32.   } // Kernel1 の終わり
33.
34.   // カーネル 2
35.   {
36.     // それぞれ N 個の整数を保持する 3 つのバッファーを作成
37.     sycl::buffer<int> ABuf(&AData[0], N);
38.     sycl::buffer<int> BBuf(&BData[0], N);
39.     sycl::buffer<int> CBuf(&CData[0], N);
40.
41.     Q.submit([&](auto &h) {
42.       // デバイスアクセサーを作成
43.       sycl::accessor aA(ABuf, h, sycl::read_only);
44.       sycl::accessor aB(BBuf, h, sycl::read_only);
45.       sycl::accessor aC(CBuf, h);
46.       h.parallel_for(N, [=](auto i) { aC[i] += aA[i] + aB[i]; });
47.     });
48.   } // Kernel 2 の終わり
49.
50.   // バッファーは破棄され、CData が更新されアクセスできるようになりました
51.   for (int i = 0; i < N; i++) {
52.     printf("%d\n", CData[i]);
53.   }
54.
55.   return 0;
56. }
```

`read_write` アクセスモードの代わりに `read_only` を指定するのは、カーネルが `for` ループ内で繰り返し起動される場合に有用です。アクセスモードが `read_write` である場合、カーネルが計算を終了し次のカーネルを起動する前にデータの準備ができている必要があるため、起動されたカーネルはシリアル化されます。一方、アクセスモードが `read_only` の場合、ランタイムはカーネルを並列に起動できます。

バッファーの宣言とカーネルの起動は、ブロック内にあることに注意してください。これにより、最初のカーネルが完了するとバッファーはスコープ外になります。そしてデバイスからホストへのコンテンツのコピーがトリガーされます。2 番目のカーネルは、新しいバッファーが同じメモリーに宣言される別のブロック内にあるため、ホストからデバイスへの同一メモリーのコピーが再びトリガーされます。ホストとデバイス間の往復コピーは、バッファーを宣言してから、このバッファーが示すメモリーが存続する間にスコープ内にあることを確認することで回避できます。不要なメモリー転送を回避する方法を以下に示します。

```
#include <CL/sycl.hpp>
#include <stdio.h>

constexpr int N = 100;

int main() {

  int AData[N];
  int BData[N];
  int CData[N];

  sycl::queue Q;

  // それぞれ N 個の整数を保持する 3 つのバッファーを作成
  sycl::buffer<int> ABuf(&AData[0], N);
  sycl::buffer<int> BBuf(&BData[0], N);
  sycl::buffer<int> CBuf(&CData[0], N);

  // カーネル 1
  Q.submit([&](auto &h) {
    // デバイスアクセサーを作成。
    // プロパティー no_init は、バッファーの以前の
    // 内容を破棄できることをランタイムに通知します。
    sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);
    sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);
    sycl::accessor aC(CBuf, h, sycl::write_only, sycl::no_init);

    h.parallel_for(N, [=](auto i) {
      aA[i] = 11;
      aB[i] = 22;
      aC[i] = 0;
    });
  });

  // カーネル 2
  Q.submit([&](auto &h) {
    // デバイス sycl::accessors を作成
    sycl::accessor aA(ABuf, h, sycl::read_only);
    sycl::accessor aB(BBuf, h, sycl::read_only);
    sycl::accessor aC(CBuf, h);
    h.parallel_for(N, [=](auto i) { aC[i] += aA[i] + aB[i]; });
  });

  // ホストアクセサーを作成することで、カーネルの終了待機がトリガーされ、
  // デバイスからホストへデータがコピーされます
  sycl::host_accessor h_acc(CBuf);
  for (int i = 0; i < N; i++) {
    printf("%d\n", h_acc[i]);
  }

  return 0;
}
```

次の例は、異なるスコープのブロック化による別の手法を示します。この例では、**カーネル 1** の終わりと**カーネル 2** の始めにホストからデバイスへのバッファーコピーはありません。これら 3 つのバッファーのコピーは、バッファーがスコープ外になった**カーネル 2** の最後で発生します。

```
#include <CL/sycl.hpp>
#include <stdio.h>

constexpr int N = 100;

int main() {

  int AData[N];
  int BData[N];
  int CData[N];

  sycl::queue Q;

  {
    // それぞれ N 個の整数を保持する 3 つのバッファーを作成
    sycl::buffer<int> ABuf(&AData[0], N);
    sycl::buffer<int> BBuf(&BData[0], N);
    sycl::buffer<int> CBuf(&CData[0], N);

    // カーネル 1
    Q.submit([&](auto &h) {
      // デバイスアクセサーを作成。
      // プロパティー no_init は、バッファーの以前の
      // 内容を破棄できることをランタイムに通知します。
      sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);
      sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);
      sycl::accessor aC(CBuf, h, sycl::write_only, sycl::no_init);

      h.parallel_for(N, [=](auto i) {
        aA[i] = 11;
        aB[i] = 22;
        aC[i] = 0;
      });
    });

    // カーネル 2
    Q.submit([&](auto &h) {
      // デバイスアクセサーを作成
      sycl::accessor aA(ABuf, h, sycl::read_only);
      sycl::accessor aB(BBuf, h, sycl::read_only);
      sycl::accessor aC(CBuf, h);
      h.parallel_for(N, [=](auto i) { aC[i] += aA[i] + aB[i]; });
    });
  }
  // バッファーはスコープ外となるため、デバイスからホストへコピーし戻す必要があります。
  // これには、すべてのカーネルが終了するまで待機する必要がありますが、
  // 明示的な待機は必要ありません
  for (int i = 0; i < N; i++) {
    printf("%d\n", CData[i]);
  }

  return 0;
}
```

以下に示すように、カーネルを定義するラムダ関数の変数キャプチャーの一部として、ホスト上の `read-only` 変数のコピーにデバイスがアクセスできるカーネルを作成する方法もあります。しかしこの方法では、カーネルの呼び出しごとにベクトル `AData` と `BData` に関連付けられたデータをデバイスにコピーする必要があります。

```
1.  #include <CL/sycl.hpp>
2.  #include <stdio.h>
3.
4.  constexpr int N = 100;
5.  constexpr int iters = 100;
6.
7.  int main() {
8.
9.    int AData[N];
10.   int BData[N];
11.   int CData[N];
12.
13.   sycl::queue Q;
14.   sycl::buffer<int> CBuf(&CData[0], N);
15.
16.   {
17.     // それぞれ N 個の整数を保持する 2 つのバッファーを作成
18.     sycl::buffer<int> ABuf(&AData[0], N);
19.     sycl::buffer<int> BBuf(&BData[0], N);
20.
21.     // カーネル 1
22.     Q.submit([&](auto &h) {
23.       // デバイスアクセサーを作成。
24.       // プロパティー no_init は、バッファーの以前の
25.       // 内容を破棄できることをランタイムに通知します。
26.       sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);
27.       sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);
28.       sycl::accessor aC(CBuf, h, sycl::write_only, sycl::no_init);
29.
30.       h.parallel_for(N, [=](auto i) {
31.         aA[i] = 11;
32.         aB[i] = 22;
33.         aC[i] = 0;
34.       });
35.     });
36.   }
37.
38.   for (int it = 0; it < iters; it++) {
39.     // カーネル 2
40.     Q.submit([&](auto &h) {
41.       // デバイスアクセサーを作成
42.       sycl::accessor aC(CBuf, h);
43.       h.parallel_for(N, [=](auto i) { aC[i] += AData[i] + BData[i]; });
44.     });
45.   }
46.
47.   sycl::host_accessor h_acc(CBuf);
48.   for (int i = 0; i < N; i++) {
49.     printf("%d\n", h_acc[i]);
50.   }
51.
52.   return 0;
53. }
```

ベクトルがホストからデバイスに一度だけコピーされるように、バッファーと `read-only` アクセサーを定義することを推奨します。次のカーネルでは、`AData` と `Bdata` へのアクセスは 24 行目と 25 行目の `aA(ABuf)` と `aB(BBuf)` を介して行われ、36 行目と 37 行目で `read-only` として定義されているため、スコープ外になってもデバイスからホストへのコピーは発生しません。

```
1.  #include <CL/sycl.hpp>
2.  #include <stdio.h>
3.  constexpr int N = 100;
4.  constexpr int iters = 100;
5.  int main() {
6.    int AData[N];
7.    int BData[N];
8.    int CData[N];
9.    sycl::queue Q;
10.   sycl::buffer<int> CBuf(&CData[0], N);
11.   {
12.   // それぞれ N 個の整数を保持する 2 つのバッファーを作成
13.   sycl::buffer<int> ABuf(&AData[0], N);
14.   sycl::buffer<int> BBuf(&BData[0], N);
15.   // カーネル 1
16.   Q.submit([&](auto &h) {
17.     // デバイスアクセサーを作成。
18.     // プロパティー no_init は、バッファーの以前の
19.     // 内容を破棄できることをランタイムに通知します。
20.     sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);
21.     sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);
22.     sycl::accessor aC(CBuf, h, sycl::write_only, sycl::no_init);
23.     h.parallel_for(N, [=](auto i) {
24.       aA[i] = 11;
25.       aB[i] = 22;
26.       aC[i] = 0;
27.     });
28.   });
29.   }
30.   sycl::buffer<int> ABuf(&AData[0], N);
31.   sycl::buffer<int> BBuf(&BData[0], N);
32.   for (int it = 0; it < iters; it++) {
33.     // カーネル 2
34.     Q.submit([&](auto &h) {
35.       // デバイスアクセサーを作成
36.       sycl::accessor aA(ABuf, h, sycl::read_only);
37.       sycl::accessor aB(BBuf, h, sycl::read_only);
38.       sycl::accessor aC(CBuf, h);
39.       h.parallel_for(N, [=](auto i) { aC[i] += aA[i] + aB[i]; });
40.     });
41.   }
42.   sycl::host_accessor h_acc(CBuf);
43.   for (int i = 0; i < N; i++) {
44.     printf("%d\n", h_acc[i]);
45.   }
46.   return 0;
47. }
```

# ホスト/デバイスの調整

ホストとアクセラレーター・デバイス間には大量の計算および通信リソースがあるため、それらを効率良く利用することを考慮しなければなりません。

ここでは、ホストとアクセラレーターの処理の調整に関連するトピックについて説明します。

- ホストとデバイス間のデバイス間の非同期および重複データ転送

## ホストとデバイス間のデバイス間の非同期および重複データ転送

アクセラレーターはホスト CPU とは別のデバイスであり、PCIe* や CXL* などのバスに接続されます。このバスは、そのタイプに応じてホストとデバイス間で転送できるデータ帯域幅が定められます。アクセラレーターは計算に必要なデータをホストから転送しなければならず、システム全体のパフォーマンスは、この転送がどれだけ高速に行われるかに依存します。

### ホストとデバイス間の帯域幅

ほとんどのアクセラレーターは、PCIe* を介してホストと接続されます。次の表に示すように、PCIe* の世代が異なると帯域幅の性能は異なります。

**世代ごとの PCIe* 帯域幅**

| PCIe* バージョン | 転送レート | スループット |
|---|---|---|
| 1.0 | 2.5GT/秒 | 0.250GB/秒 |
| 2.0 | 5.0GT/秒 | 0.500GB/秒 |
| 3.0 | 8.0GT/秒 | 0.985GB/秒 |
| 4.0 | 16.0GT/秒 | 1.969GB/秒 |
| 5.0 | 32.0GT/秒 | 3.938GB/秒 |

アクセラレーターが搭載するローカルメモリーの帯域幅は、PCIe* リンクを介したホストからデバイスへの帯域幅よりも 1 桁高くなります。例えば、最新 GPU の HBM (High Bandwidth Memory) は、x16 PCIe* の 63GB/秒の帯域幅と比べると、最大 900GB/秒にも達します。そのため、データをローカルメモリーに保持して、ホストとデバイス間のデータ転送を可能な限り回避することが不可欠です。例えば、一部のカーネルがアクセラレーターで効率良く実行できない場合でも、アクセラレーター間、またはホストとアクセラレーター間でのデータ移動を回避するため、アクセラレーターですべてのカーネルを実行するほうが最終的に良い結果となることもあります。

また、中間データ構造を使用する場合、ホストでデータを作成してアクセラレーター間で移動を行うのではなく、デバイス上でデータを作成して使用します。ここでは、デバイス上で中間結果を生成してリダクション操作を行うカーネルについて考えます。カーネル `ComputeParallel1` では、一時アキュムレーターがホスト上に作成され、すべての work-item が中間結果をホストに配置します。このアキュムレーターはホストに戻され、さらにレデュースされます (37 行目)。

```
1.  float ComputeParallel1(sycl::queue &q, std::vector<float> &data) {
2.    const size_t data_size = data.size();
3.    float sum = 0;
4.    static float *accum = 0;
5.
6.    if (data_size > 0) {
7.      const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
8.      int num_EUs =
9.          q.get_device().get_info<sycl::info::device::max_compute_units>();
```

```
    int vec_size =
        q.get_device()
            .get_info<sycl::info::device::native_vector_width_float>();
    int num_processing_elements = num_EUs * vec_size;
    int BATCH = (N + num_processing_elements - 1) / num_processing_elements;
    sycl::buffer<float> buf(data.data(), data.size(), props);
    sycl::buffer<float> accum_buf(accum, num_processing_elements, props);

    if (!accum)
      accum = new float[num_processing_elements];

    q.submit([&](auto &h) {
      sycl::accessor buf_acc(buf, h, sycl::read_only);
      sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
      h.parallel_for(num_processing_elements, [=](auto index) {
        size_t glob_id = index[0];
        size_t start = glob_id * BATCH;
        size_t end = (glob_id + 1) * BATCH;
        if (end > N)
          end = N;
        float sum = 0.0;
        for (size_t i = start; i < end; i++)
          sum += buf_acc[i];
        accum_acc[glob_id] = sum;
      });
    });
    q.wait();
    sycl::host_accessor h_acc(accum_buf);
    for (int i = 0; i < num_processing_elements; i++)
      sum += h_acc[i];
  }
  return sum;
} // ComputeParallel1 の終わり
```

別のアプローチとして、この一時アキュムレーターをアクセラレーターに保持して、1 つの work-item のみで別のカーネルを起動する方法があります。ここでは、`ComputeParallel2` カーネルの 43 行目でデバイスが最終的なリダクション操作を実行します。このカーネルにはそれほど多くの並列処理がないため、1 つの work-item によってのみ実行されることに注意してください。一部のプラットフォームでは、これはデータをホストに転送してからホストでリダクションを行うより良好な場合があります。

```
float ComputeParallel2(sycl::queue &q, std::vector<float> &data) {
  const size_t data_size = data.size();
  float sum = 0;
  static float *accum = 0;

  if (data_size > 0) {
    const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
    int num_EUs =
        q.get_device().get_info<sycl::info::device::max_compute_units>();
    int vec_size =
        q.get_device()
            .get_info<sycl::info::device::native_vector_width_float>();
    int num_processing_elements = num_EUs * vec_size;
    int BATCH = (N + num_processing_elements - 1) / num_processing_elements;
    sycl::buffer<float> buf(data.data(), data.size(), props);
    sycl::buffer<float> accum_buf(accum, num_processing_elements, props);
    sycl::buffer<float> res_buf(&sum, 1, props);
    if (!accum)
      accum = new float[num_processing_elements];

    q.submit([&](auto &h) {
      sycl::accessor buf_acc(buf, h, sycl::read_only);
      sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
      h.parallel_for(num_processing_elements, [=](auto index) {
        size_t glob_id = index[0];
        size_t start = glob_id * BATCH;
```

```
27.         size_t end = (glob_id + 1) * BATCH;
28.         if (end > N)
29.           end = N;
30.         float sum = 0.0;
31.         for (size_t i = start; i < end; i++)
32.           sum += buf_acc[i];
33.         accum_acc[glob_id] = sum;
34.       });
35.     });
36.
37.     q.submit([&](auto &h) {
38.       sycl::accessor accum_acc(accum, h, sycl::read_only);
39.       sycl::accessor res_acc(res_buf, h, sycl::write_only, sycl::no_init);
40.       h.parallel_for(1, [=](auto index) {
41.         res_acc[index] = 0;
42.         for (size_t i = 0; i < num_processing_elements; i++)
43.           res_acc[index] += accum_acc[i]; // 最終的なリダクション
44.       });
45.     });
46.   }
47.   // バッファーがスコープ外となるため、データはデバイスからホストへ転送されます
48.   return sum;
49. } // ComputeParallel2 の終わり
```

## デバイスでの計算とホスト/デバイス間のオーバーラップ・データ転送

一部の GPU には、ホストからデバイスにデータをコピーする専用エンジンを持つものがあります。それらを効果的に利用することで、ホストからデバイスへのデータ転送をデバイスの実行とオーバーラップさせることができます。次の例では、メモリーブロックがいくつかのチャンクに分割され、それぞれのチャンクがアクセラレーターに転送されて (49 行目)、処理され (53 行目)、結果がホストに戻されます (54 行目)。この 3 つのタスクのチャンクは独立しているため、ハードウェア・リソースの状況に応じて並列に処理できます。ホストとデバイス間のデータ転送に利用できるコピーエンジンが存在するシステムでは、異なるループ反復から操作を並列に実行できることが分かります。次の 2 つの方法で並列実行できます。

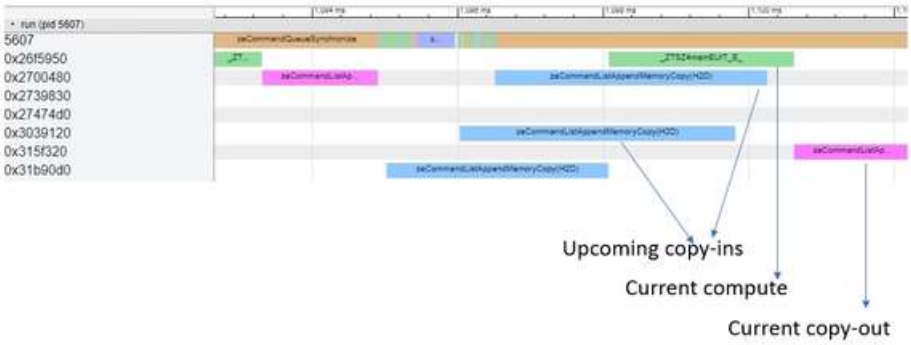
- 1 つは GPU EU で実行され、もう 1 つはコピーエンジンによって実行される 2 つのメモリーコピー間
- コピーはコピーエンジンによって実行され、計算カーネルは GPU EU で実行される、メモリーコピーと計算カーネル間

```
1.  #include <CL/sycl.hpp>
2.  #define NITERS 10
3.  #define KERNEL_ITERS 10000
4.  #define NUM_CHUNKS 10
5.  #define CHUNK_SIZE 10000000
6.  class Timer {
7.    public:
8.    Timer() : start_(std::chrono::steady_clock::now()) {}
9.    double Elapsed() {
10.     auto now = std::chrono::steady_clock::now();
11.     return std::chrono::duration_cast<Duration>(now - start_).count();
12.   }
13.   private:
14.   using Duration = std::chrono::duration<double>;
15.   std::chrono::steady_clock::time_point start_;
16. };
17. int main() {
18.   const int num_chunks = NUM_CHUNKS;
19.   const int chunk_size = CHUNK_SIZE;
20.   const int iter = NITERS;
21.   sycl::queue q;
22.   // ホストデータの割り当てと初期化
23.   float *host_data[num_chunks];
```

```
  for (int c = 0; c < num_chunks; c++) {
    host_data[c] = sycl::malloc_host<float>(chunk_size, q);
    float val = c;
    for (int i = 0; i < chunk_size; i++)
      host_data[c][i] = val;
  }
  std::cout << "Allocated host data\n";
  // デバイスメモリーに配列を割り当てて初期化
  float *device_data[num_chunks];
  for (int c = 0; c < num_chunks; c++) {
    device_data[c] = sycl::malloc_device<float>(chunk_size, q);
    float val = 1000.0;
    q.fill<float>(device_data[c], val, chunk_size);
  }
  q.wait();
  std::cout << "Allocated device data\n";
  Timer timer;
  for (int it = 0; it < iter; it++) {
    for (int c = 0; c < num_chunks; c++) {
      auto add_one = [=](auto id) {
        for (int i = 0; i < KERNEL_ITERS; i++)
        device_data[c][id] += 1.0;
      };
      // 以前のイベントに依存しないコピーイン
      auto copy_in =
      q.memcpy(device_data[c], host_data[c], sizeof(float) * chunk_size);
      // 計算はコピーインを待機
      auto compute = q.parallel_for(chunk_size, copy_in, add_one);
      auto cg = [=](auto &h) {
        h.depends_on(compute);
        h.memcpy(host_data[c], device_data[c], sizeof(float) * chunk_size);
      };
      // コピーアウトは計算を待機
      auto copy_out = q.submit(cg);
    }
    q.wait();
  }
  auto elapsed = timer.Elapsed() / iter;
  for (int c = 0; c < num_chunks; c++) {
    for (int i = 0; i < chunk_size; i++) {
      if (host_data[c][i] != (float)((c + KERNEL_ITERS * iter))) {
        std::cout << "Mismatch for chunk: " << c << " position: " << i
                  << " expected: " << c + 10000 << " got: " << host_data[c][i]
                  << "\n";
        break;
      }
    }
  }
  std::cout << "Time = " << elapsed << " usecs\n";
}
```

`onetrace` で収集された次の図のタイムラインでは、コピーインが計算カーネルの実行とオーバーラップしていることが分かります。また、複数のコピーインが複数のコピーエンジンで並列に実行されていることが分かります。

**計算カーネルの実行とコピーインのオーバーラップ**




上記の例では、GPU が 1 つしかないため (独立していても) 2 つのカーネルを同時に実行することはできません (GPU を小さなチャンクに分割して、それらで異なるカーネルを同時に実行することは可能です)。

# 複数のヘテロジニアス・デバイスを使用

通常ほとんどのアクセラレーターは、大量の計算リソースを持つサーバーに搭載されます。例えば、一般的なサーバーには最大 8 つのソケットがあり、それぞれのソケットには 50 以上のコアがあります。SYCL* は、CPU とアクセラレーターを均一に扱い、それらにワークを分散する機能を提供します。プラットフォーム上のヘテロジニアス・コンピューティングのリソース間にワークをバランスよく分散するのは、プログラマーの責任です。

## 各種デバイスで計算をオーバーラップ

SYCL* は、デバイスセレクターの抽象化によりさまざまなデバイスへのアクセスを可能にします。デバイスごとにキューを作成して、カーネルをデバイスに送信し実行することができます。SYCL* におけるすべてのカーネル送信は非ブロッキングです。つまり、実行のためカーネルがキューに送信されると、キューの待機が明示的に要求されない限り、ホストはカーネルの終了を待機しません。これにより、カーネルがアクセラレーターで実行されている間に、ホストはワークを実行したり、他のデバイスでワークを開始することができます。

ホスト CPU はアクセラレーターとして扱うことができ、SYCL* はカーネルをホスト CPU に送信して実行することができます。これは完全に独立しており、カーネルの送信と作成を調整するためホストが実行するジョブと直交しています。ベースとなるオペレーティング・システムは、CPU アクセラレーターに送信されたカーネルを別のプロセスとして管理し、同じ **OpenCL*/レベル 0** ランタイムの機能を使用してホストデバイスと情報を交換します。

以下では、単一の GPU デバイスで機能する単純なベクトル加算の例を示します。

```
1.  size_t VectorAdd1(sycl::queue &q, const IntArray &a, const IntArray &b,
2.                    IntArray &sum, int iter) {
3.    sycl::range num_items{a.size()};
4.
5.    sycl::buffer a_buf(a);
6.    sycl::buffer b_buf(b);
7.    sycl::buffer sum_buf(sum.data(), num_items);
8.    auto start = std::chrono::steady_clock::now();
9.    for (int i = 0; i < iter; i++) {
10.
11.     auto e = q.submit([&](auto &h) {
12.       // 入力アクセサー
13.       sycl::accessor a_acc(a_buf, h, sycl::read_only);
14.       sycl::accessor b_acc(b_buf, h, sycl::read_only);
15.       // 出力アクセサー
16.       sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
17.
18.       h.parallel_for(num_items,
19.                      [=](auto i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
20.     });
21.   }
22.   q.wait();
23.   auto end = std::chrono::steady_clock::now();
24.   std::cout << "Vector add1 completed on device - took "
25.             << (end - start).count() << " u-secs\n";
26.   return ((end - start).count());
27. } // VectorAdd1 の終わり
```

次のカーネルでは、入力ベクトルが 2 つに分割され、同時に実行可能な 2 つの異なるアクセラレーター (CPU と GPU) で計算されます。この並列処理では、送信されるだけではなく、カーネルが実際にデバイスで起動されるようにする必要があります。カーネルの起動に必要な時間は、ホストから送信される場合よりも大幅に長くなる可能性があります。利用率、スループット、待機時間などのメトリックを最大化するため、実装のヒューリスティックに基づいてカーネルを起動する時間を決定します。例えば、OpenCL* バックエンドの特定のプラットフォームでは、キューで `clFlush(cq);` を明示的に発行してから (41 行目)、アクセラレーターでカーネルを起動する必要があります。

```
1.  size_t VectorAdd2(sycl::queue &q1, sycl::queue &q2, const IntArray &a,
2.                    const IntArray &b, IntArray &sum, int iter) {
3.    sycl::range num_items{a.size() / 2};
4.
5.    auto start = std::chrono::steady_clock::now();
6.    {
7.      sycl::buffer a1_buf(a.data(), num_items);
8.      sycl::buffer b1_buf(b.data(), num_items);
9.      sycl::buffer sum1_buf(sum.data(), num_items);
10.
11.     sycl::buffer a2_buf(a.data() + a.size() / 2, num_items);
12.     sycl::buffer b2_buf(b.data() + a.size() / 2, num_items);
13.     sycl::buffer sum2_buf(sum.data() + a.size() / 2, num_items);
14.     for (int i = 0; i < iter; i++) {
15.
16.       q1.submit([&](auto &h) {
17.         // 入力アクセサー
18.         sycl::accessor a_acc(a1_buf, h, sycl::read_only);
19.         sycl::accessor b_acc(b1_buf, h, sycl::read_only);
20.         // 出力アクセサー
21.         sycl::accessor sum_acc(sum1_buf, h, sycl::write_only, sycl::no_init);
22.
23.         h.parallel_for(num_items,
24.                        [=](auto i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
25.       });
```

```
26.       // ホストでワークを実行
27.       q2.submit([&](auto &h) {
28.         // 入力アクセサー
29.         sycl::accessor a_acc(a2_buf, h, sycl::read_only);
30.         sycl::accessor b_acc(b2_buf, h, sycl::read_only);
31.         // 出力アクセサー
32.         sycl::accessor sum_acc(sum2_buf, h, sycl::write_only, sycl::no_init);
33.
34.         h.parallel_for(num_items,
35.                        [=](auto i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
36.       });
37.     }
38.     // 一部のプラットフォームでは、CPU と GPU 間の実行がオーバーラップすることを
39.     // 確実にするため、明示的なキューのフラッシュが必要になります
40.     // cl_command_queue cq = q1.get();
41.     // clFlush(cq);
42.     // cq=q2.get();
43.     // clFlush(cq);
44.   }
45.   q1.wait();
46.   q2.wait();
47.   auto end = std::chrono::steady_clock::now();
48.   std::cout << "Vector add2 completed on device - took "
49.             << (end - start).count() << " u-secs\n";
50.   return ((end - start).count());
51. } // VectorAdd2 の終わり
```

上記 2 つのカーネルの実行時間を確認すると、問題解決に利用できる専用ハードウェアがあるため、アプリケーションは以前のおよそ 2 倍高速に実行されていることが分かります。良好なバランスを達成するため、上記の例のようにワークを均等に分割する代わりに、アクセラレーターの機能に応じて分割することを考えなければなりません。

# コンパイル

oneAPI には、異なる複数のコンパイル方法があります。アプリケーションの `main` のソースがコンパイルされ、オフロードされるカーネルがコンパイルされます。カーネルは、事前 (AOT) またはジャストイン (JIT) コンパイルされる場合があります。

ここでは、コンパイルに関連するトピックと、実行効率に与える影響について説明します。

- ジャストインタイム・コンパイル
- 事前コンパイル
- 特殊化定数
- 浮動小数点計算における精度とパフォーマンスのトレードオフ

## ジャストインタイム・コンパイル

インテル® oneAPI の SYCL* コンパイラーは、SYCL* プログラムを SPIR-V* と呼ばれる中間言語に変換し、コンパイルで生成されるバイナリー内に格納します。バイナリーの代わりに中間ファイルを生成する利点は、実行時に SPIR-V* コードをプラットフォームのバイナリーコードに変換することで、このコードを任意のハードウェア・プラットフォームで実行できることです。中間コードを実行時にバイナリーに変換するプロセスは、JIT (ジャストインタイム) コンパイルと呼ばれます。JIT コンパイルは実行時に要求に応じて行われます。この JIT コンパイルを制御するにはいくつかの方法があります。デフォルトでは、バイナリー内の SPIR-V* コードは、最初にオフロードされたカーネルの実行開始前に変換されます。

```
#include <CL/sycl.hpp>
#include <array>
#include <chrono>
#include <iostream>

// この例の配列タイプとデータサイズ
constexpr size_t array_size = (1 << 16);
typedef std::array<int, array_size> IntArray;

void VectorAdd1(sycl::queue &q, const IntArray &a, const IntArray &b,
                IntArray &sum) {
  sycl::range num_items{a.size()};

  sycl::buffer a_buf(a);
  sycl::buffer b_buf(b);
  sycl::buffer sum_buf(sum.data(), num_items);

  auto e = q.submit([&](auto &h) {
    // 入力アクセサー
    sycl::accessor a_acc(a_buf, h, sycl::read_only);
    sycl::accessor b_acc(b_buf, h, sycl::read_only);
    // 出力アクセサー
    sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

    h.parallel_for(num_items,
                   [=](auto i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
  });
  q.wait();
}

void VectorAdd2(sycl::queue &q, const IntArray &a, const IntArray &b,
                IntArray &sum) {
  sycl::range num_items{a.size()};

  sycl::buffer a_buf(a);
  sycl::buffer b_buf(b);
  sycl::buffer sum_buf(sum.data(), num_items);

  auto e = q.submit([&](auto &h) {
    // 入力アクセサー
    sycl::accessor a_acc(a_buf, h, sycl::read_only);
    sycl::accessor b_acc(b_buf, h, sycl::read_only);
    // 出力アクセサー
    sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

    h.parallel_for(num_items,
                   [=](auto i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
  });
  q.wait();
}

void InitializeArray(IntArray &a) {
  for (size_t i = 0; i < a.size(); i++)
    a[i] = i;
}

int main() {
  IntArray a, b, sum;

  InitializeArray(a);
  InitializeArray(b);

  sycl::queue q(sycl::default_selector_v,
                sycl::property::queue::enable_profiling{});

  std::cout << "Running on device: "
            << q.get_device().get_info<sycl::info::device::name>() << "\n";
```

```
68.   std::cout << "Vector size: " << a.size() << "\n";
69.   auto start = std::chrono::steady_clock::now();
70.   VectorAdd1(q, a, b, sum);
71.   auto end = std::chrono::steady_clock::now();
72.   std::cout << "Initial Vector add1 successfully completed on device - took "
73.             << (end - start).count() << " nano-secs\n";
74.
75.   start = std::chrono::steady_clock::now();
76.   VectorAdd1(q, a, b, sum);
77.   end = std::chrono::steady_clock::now();
78.   std::cout << "Second Vector add1 successfully completed on device - took "
79.             << (end - start).count() << " nano-secs\n";
80.
81.   start = std::chrono::steady_clock::now();
82.   VectorAdd2(q, a, b, sum);
83.   end = std::chrono::steady_clock::now();
84.   std::cout << "Initial Vector add2 successfully completed on device - took "
85.             << (end - start).count() << " nano-secs\n";
86.
87.   start = std::chrono::steady_clock::now();
88.   VectorAdd2(q, a, b, sum);
89.   end = std::chrono::steady_clock::now();
90.   std::cout << "Second Vector add2 successfully completed on device - took "
91.             << (end - start).count() << " nano-secs\n";
92.   return 0;
93. }
```

上記のプログラムを次のコマンドでコンパイルします (ソースファイル名は `example.cpp`)。

```
$ icpx -fsycl -O3 -o example example.cpp
```

実行すると、生成される出力から `VectorAdd1` の最初の呼び出しは JIT コンパイルのコストが加算されるため、プログラム内のほかの `VectorAdd1` カーネルの呼び出しよりもはるかに時間がかかることが分かります。

アプリケーションに複数のカーネルが含まれる場合、コンパイル時にオプションを指定して、eager または lazy JIT コンパイルを強制できます。eager JIT コンパイルでは、実行時にバイナリー内のすべてのカーネルが JIT を呼び出しますが、lazy JIT コンパイルでは、実行中に実際にカーネルが呼び出されたときにのみ JIT を有効にします。一部のカーネルが呼び出されない状況では、実行されないコードを変換しないという利点があり、不要な JIT コンパイルを回避できます。このモードは、次のオプションで有効にできます。

```
-fsycl-device-code-split=<value>
```

ここで、`value` は次のいずれかです。

- `per_kernel`: カーネルが呼び出されたときにのみ JIT コンパイルを行うコードを生成します。
- `per_source`: ソースファイル内のいずれかのカーネルが呼び出されたときに、ソースファイル内のすべてのカーネルを JIT コンパイルするコードを生成します。
- `off`: アプリケーション内のすべてのカーネルを eager JIT コンパイルします。
- `auto`: デフォルト。コンパイラーはヒューリスティックに従って JIT コンパイル用にデバイスコードを分割する最適な方法を選択します。

上記のプログラムを次のオプションでコンパイルします。

```
$ icpx -fsycl -O3 -o example vec1.cpp vec2.cpp main.cpp -fsycl-device-code-
split=per_kernel
```

プログラムを実行すると、カーネル実行のタイミングから、`VectorAdd1` と `VectorAdd2` の最初の起動まで時間がかかりますが、2 回目の呼び出しには JIT コンパイルのコストがないため、時間がかかっていないことが分かります。

上記の例では、`VectorAdd1` と `VectorAdd2` を異なるファイルに配置して `per_source` オプションを指定 (または未指定) して、カーネルの実行時間への影響を確認できます。次のようにコンパイルします。

```
$ icpx -fsycl -O3 -o example vec1.cpp vec2.cpp main.cpp -fsycl-device-code-
split=per_source
```

生成されたバイナリーを実行すると、カーネルの実行時間は、最初のカーネル呼び出しでは JIT コンパイルのコストが生じているのに対し、後続のカーネル呼び出しではそのコストがないことが分かります。次に、以下のようにコンパイルします。

```
$ icpx -fsycl -O3 -o example vec1.cpp vec2.cpp main.cpp
```

生成されたバイナリーを実行すると、カーネルの実行時間は、最初のカーネル呼び出しですべての JIT コンパイルのコストが発生し、後続のすべてのカーネル呼び出しでそのコストがないことが分かります。

## 事前コンパイル

実行時の JIT コンパイルのオーバーヘッドは、事前 (AOT) コンパイルによって回避できます。AOT コンパイルでは、SPIR-V* 中間コードに代わって、コンパイル時に選択されたプラットフォームの実際のアセンブリー・コードがバイナリーに出力されます。AOT コンパイルの利点は、実行中に SPIR-V* からアセンブリー・コードに JIT コンパイルする必要がないため、コードの実行が高速化されることです。また欠点は、バイナリーは特定のプラットフォーム専用であるため、それ以外のプラットフォームでは実行できないことです。

インテル® GPU 向けに事前 (AOT) モードでコンパイルを行う場合、特定の GPU ターゲットを明確にするため追加のコンパイルオプション (`-Xs`) が必要です。

### SYCL* 向けの AOT コンパイラー・オプション

インテル® データセンター GPU マックス・シリーズ:

```
$ icpx -fsycl -fsycl-targets=spir64_gen -Xs "-device 0x0BD5 -revision_id 0x2f"
source.c
```

### OpenMP* 向けの AOT コンパイラー・オプション

インテル® データセンター GPU マックス・シリーズ:

```
$ icx -fiopenmp -fopenmp-targets=spir64_gen -Xs "-device 0x0BD5 -revision_id 0x2F"
source.c
```

**注:**

- 上記のコンパイルオプションは、OpenMP* Fortran プログラムを AOT モード (`ifx` を使用) でコンパイルする場合にも利用できます。
- JIT モードでは、インテル® グラフィックス・コンパイラーがハードウェアのタイプを認識して、自動的に調整されます。`-revision_id` オプションは、AOT モードでのみ必要です。

## 特殊化定数

SYCL* には、特殊化定数と呼ばれる機能があり、JIT コンパイルを明示的にトリガーして、特殊化定数の実行時の値をベースに SPIR-V 中間コードからバイナリーを生成できます。この JIT コンパイルは、定数値が判明していればプログラムの実行中に行われます。これは、`-fsycl-device-code-split` オプションによってトリガーされる JIT コンパイルとは異なります。

次の例では、`set_specialization_constant` 呼び出しにより、7 行目で定義される `get_value` 関数呼び出しで返される値が SYCL* カーネルにバインドされます。カーネルバンドルが最初にコンパイルされるときにこの値が不明であるため、最適化に利用できません。実行時に、`get_value` 関数が呼び出されると、値は既知となり、コマンド・グループ・ハンドラーで使用され、この値を基に特殊化されたカーネルの JIT コンパイルがトリガーされます。

```
1.  #include <CL/sycl.hpp>
2.  #include <vector>
3.  class specialized_kernel;
4.  // 特殊化定数の const static 識別子
5.  const static sycl::specialization_id<float> value_id;
6.  //    ランタイムで値をフェッチ
7.  float get_value() { return 10; };
8.  int main() {
9.    sycl::queue queue;
10.   std::vector<float> vec(1);
11.   {
12.   sycl::buffer<float> buffer(vec.data(), vec.size());
13.   queue.submit([&](auto &cgh) {
14.     sycl::accessor acc(buffer, cgh, sycl::write_only, sycl::no_init);
15.     // 特殊化定数値を設定
16.     cgh.template set_specialization_constant<value_id>(get_value());
17.     // ランタイムは、特殊化定数を前の
18.     // `set_specialization_constant<value_id>` の呼び出しで
19.     // 与えられたリテラル値に置き換えてカーネルを構築します
20.     cgh.template single_task<specialized_kernel>(
21.       [=](sycl::kernel_handler kh) {
22.         const float val = kh.get_specialization_constant<value_id>();
23.         acc[0] = val;
24.     });
25.   });
26.   }
27.   queue.wait_and_throw();
28.   std::cout << vec[0] << std::endl;
29.   return 0;
30. }
```

22 行目の専用カーネルは最終的に次のようなコードになります。

```
cgh.single_task<specialized_kernel>( [=]() { acc[0] = 10; });
```

この JIT コンパイルは、カーネルの実行時間にも影響します。以下に例を示します。

```
1.  #include <CL/sycl.hpp>
2.  #include <chrono>
3.  #include <vector>
4.
5.  class specialized_kernel;
6.  class literal_kernel;
7.
8.  // 特殊化定数の const static 識別子
9.  const static sycl::specialization_id<float> value_id;
```

```
10.
11. //   ランタイムで値をフェッチ
12. float get_value() { return 10; };
13.
14. int main() {
15.   sycl::queue queue;
16.
17.   // カーネルクラス修飾子からカーネル ID を取得
18.   sycl::kernel_id specialized_kernel_id =
19.       sycl::get_kernel_id<specialized_kernel>();
20.
21.   // 入力状態の specialized_kernel のみでカーネルハンドルを構築
22.   sycl::kernel_bundle kb_src =
23.       sycl::get_kernel_bundle<sycl::bundle_state::input>(
24.           queue.get_context(), {specialized_kernel_id});
25.   // 特殊化定数値を設定
26.   kb_src.set_specialization_constant<value_id>(get_value());
27.
28.   auto start = std::chrono::steady_clock::now();
29.   // 設定値のカーネルハンドルを構築
30.   sycl::kernel_bundle kb_exe = sycl::build(kb_src);
31.   auto end = std::chrono::steady_clock::now();
32.   std::cout << "specialization took - " << (end - start).count()
33.             << " nano-secs\n";
34.
35.   std::vector<float> vec{0, 0, 0, 0, 0};
36.   sycl::buffer<float> buffer1(vec.data(), vec.size());
37.   sycl::buffer<float> buffer2(vec.data(), vec.size());
38.   start = std::chrono::steady_clock::now();
39.   {
40.     queue.submit([&](auto &cgh) {
41.       sycl::accessor acc(buffer1, cgh, sycl::write_only, sycl::no_init);
42.
43.       // 事前コンパイルされたカーネルバンドルを実行状態で使用
44.       cgh.use_kernel_bundle(kb_exe);
45.
46.       cgh.template single_task<specialized_kernel>(
47.           [=](sycl::kernel_handler kh) {
48.             float v = kh.get_specialization_constant<value_id>();
49.             acc[0] = v;
50.           });
51.     });
52.     queue.wait_and_throw();
53.   }
54.   end = std::chrono::steady_clock::now();
55.
56.   {
57.     sycl::host_accessor host_acc(buffer1, sycl::read_only);
58.     std::cout << "result1 (c): " << host_acc[0] << " " << host_acc[1] << " "
59.               << host_acc[2] << " " << host_acc[3] << " " << host_acc[4]
60.               << std::endl;
61.   }
62.   std::cout << "execution took : " << (end - start).count() << " nano-secs\n";
63.
64.   start = std::chrono::steady_clock::now();
65.   {
66.     queue.submit([&](auto &cgh) {
67.       sycl::accessor acc(buffer2, cgh, sycl::write_only, sycl::no_init);
68.       cgh.template single_task<literal_kernel>([=]() { acc[0] = 20; });
69.     });
70.     queue.wait_and_throw();
71.   }
72.   end = std::chrono::steady_clock::now();
73.
74.   {
75.     sycl::host_accessor host_acc(buffer2, sycl::read_only);
```

```
76.     std::cout << "result2 (c): " << host_acc[0] << " " << host_acc[1] << " "
77.               << host_acc[2] << " " << host_acc[3] << " " << host_acc[4]
78.               << std::endl;
79.   }
80.   std::cout << "execution took - " << (end - start).count() << " nano-secs\n";
81. }
```

タイミングメッセージでレポートされるランタイムの情報を見ると、カーネルの初期変換に時間を要し、JIT コンパイルされたカーネルの実行時間は短いことが分かります。特殊化定数なしで実行される同じカーネルは、実行される前にランタイムによって JIT コンパイルされるため、時間は長くなります。

特殊化定数の簡単な使用例とアプリケーションを以下に示します。

## 簡単なトリップカウントの例

次の例は、合計を実行し、特殊化定数によってトリップカウントを設定します。

```
1.  #include <CL/sycl.hpp>
2.
3.  class SpecializedKernel;
4.
5.  // 特殊化定数を特定
6.  constexpr sycl::specialization_id<int> nx_sc;
7.
8.  int main() {
9.    sycl::queue queue;
10.
11.   std::cout << "Running on "
12.             << queue.get_device().get_info<sycl::info::device::name>() << "\n";
13.
14.   std::vector<float> vec(1);
15.   {
16.     sycl::buffer<float> buf(vec.data(), vec.size());
17.
18.     // アプリケーションはここで実行を停止し、ユーザーの入力を求めます
19.     int nx;
20.     std::cout << "Enter input number ..." << std::endl;
21.     std::cin >> Nx;
22.
23.     queue.submit([&](sycl::handler &h) {
24.       sycl::accessor acc(buf, h, sycl::write_only, sycl::no_init);
25.
26.       // ランタイム変数で特殊化定数を設定
27.       h.set_specialization_constant<nx_sc>(Nx);
28.
29.       h.single_task<SpecializedKernel>([=](sycl::kernel_handler kh) {
30.         // ここでの nx_sc 値は、実行時に提供される入力値であり、
31.         // JIT コンパイラーが定数として使用できるため最適化可能です
32.         int runtime_const_trip_count = kh.get_specialization_constant<nx_sc>();
33.         int accum = 0;
34.         for (int i = 0; i < runtime_const_trip_count; i++) {
35.           accum = accum + i;
36.         }
37.         acc[0] = accum;
38.       });
39.     });
40.   }
41.   std::cout << vec[0] << std::endl;
42.   return 0;
43. }
```

ここでの目標は、カーネル内のループのトリップカウント変数 `Nx` を特殊化することです。プログラムの実行開始時にユーザーがトリップカウントを入力するため、ホスト・コンパイラーは `Nx` の値を認識できません。入力を特殊化定数として JIT コンパイラーに渡すことができるため、JIT コンパイラーはループアンロールなどの最適化を行うことができます。

特殊化定数を使用しない場合、同等の最適化を行うには変数 `Nx` はプログラム全体の定数式である必要があります。特殊化定数によってランタイム変数から定数値を取得することで、最適化が可能となり、カーネルコードを高速化できます。

ホスト・コンパイラーはは、トリップカウントが定数ではない以下のループの例では、効果的な最適化が行えません。

```
for (int i = 0; i < Nx; i++) {
  // Nx が定数でない場合、最適化は制限されます
}
```

## STREAM Triad アプリケーションを変更

次の例 (STREAM Triad の変更例) では、入力変数によって乗算器と乗算・加算の回数が決定される、いくつかの乗算・加算操作を持つ古典的な STREAM Triad を示します。

この例は、デバイスで実行されるオリジナルコードの一部です。ランタイム変数 `inner_loop_size` は、ループの上限を設定します。

```
1.  auto q0_event = q.submit([&](sycl::handler &h) {
2.        h.parallel_for<non_specialized_kernel>(array_size / 2, [=](auto idx) {
3.          // トリップカウントをランタイム変数に設定
4.          auto runtime_trip_count_const = inner_loop_size;
5.          auto accum = 0;
6.          for (size_t j = 0; j < runtime_trip_count_const; j++) {
7.            auto multiplier = scalar * j;
8.            accum = accum + A0[idx] + B0[idx] * multiplier;
9.          }
10.         C0[idx] = accum;
11.       });
12.     });
13.
14.     q.wait();
15.
16.     cl_ulong exec_time_ns0 =
17.         q0_event
18.             .get_profiling_info<sycl::info::event_profiling::command_end>() -
19.         q0_event
20.             .get_profiling_info<sycl::info::event_profiling::command_start>();
21.
22.     std::cout << "Execution time (iteration " << i
23.               << ") [sec]: " << (double)exec_time_ns0 * 1.0E-9 << "\n";
24.     min_time_ns0 = std::min(min_time_ns0, exec_time_ns0);
```

パフォーマンス向上のため、特殊化定数を使用して変数 `inner_loop_size` を特殊化します。以下は、特殊化定数を使用して、デバイスで実行されるカーネルコードの一部です。

```
1.  auto q0_event = q.submit([&](sycl::handler &h) {
2.        // ランタイム変数を使用して特殊化定数を設定
3.        h.set_specialization_constant<trip_sc>(inner_loop_size);
4.        h.parallel_for<specialized_kernel>(
5.            array_size / 2, [=](auto idx, sycl::kernel_handler kh) {
6.              // トリップカウントを特殊化定数に設定
7.              auto runtime_trip_count_const =
```

```
                  kh.get_specialization_constant<trip_sc>();
              auto accum = 0;
              for (size_t j = 0; j < runtime_trip_count_const; j++) {
                auto multiplier = scalar * j;
                accum = accum + A0[idx] + B0[idx] * multiplier;
              }
              C0[idx] = accum;
            });
      });

      q.wait();

      cl_ulong exec_time_ns0 =
          q0_event
              .get_profiling_info<sycl::info::event_profiling::command_end>() -
          q0_event
              .get_profiling_info<sycl::info::event_profiling::command_start>();

      std::cout << "Execution time (iteration " << i
                << ") [sec]: " << (double)exec_time_ns0 * 1.0E-9 << "\n";
```

最後に、特殊化されたトリップカウントの値と、通常の定数値を使用する例を比較します。以下は、通常の定数を使用して、デバイスで実行されるカーネルコードの一部です。

```
  auto q0_event = q.submit([&](sycl::handler &h) {
         h.parallel_for<regular_constant_kernel>(array_size / 2, [=](auto idx) {
           // トリップカウントを通常の定数に設定
           size_t runtime_trip_count_const = 10;
           auto accum = 0;
           for (size_t j = 0; j < runtime_trip_count_const; j++) {
             auto multiplier = scalar * j;
             accum = accum + A0[idx] + B0[idx] * multiplier;
           }
           C0[idx] = accum;
         });
      });

      q.wait();

      cl_ulong exec_time_ns0 =
          q0_event
              .get_profiling_info<sycl::info::event_profiling::command_end>() -
          q0_event
              .get_profiling_info<sycl::info::event_profiling::command_start>();

      std::cout << "Execution time (iteration " << i
                << ") [sec]: " << (double)exec_time_ns0 * 1.0E-9 << "\n";
      min_time_ns0 = std::min(min_time_ns0, exec_time_ns0);
```

上記 3 つのバージョンの実行タイミングを以下に示します。ストリームサイズは、配列 `A0`、`B0` および `C0` のサイズを表わします。内部トリップカウントは、特殊化定数を使用して設定された `runtime_trip_count_const` 変数の値を表わします。

次の表は、入力として 134217728 要素 (1024MB) のストリームサイズと 10 回の内部トリップカウントを使用して実行したタイミング出力です。

- ランタイム変数で実行: 秒単位の時間 (最速): 0.00161008
- 特殊化定数で実行: 秒単位の時間 (最速): 0.00156256
- 定数で実行: 秒単位の時間 (最速): 0.00155104

結果は予想されたように、特殊化定数を使用すると、デバイス上の計算パフォーマンスは、ランタイム変数を使用した実行時間から、定数を使用した実行時間に近くなったことが確認できます。さらに、生成されたコードを確認すると、特殊化されたアプリケーションでは、メインループがアンロールされていることが分かります。これは、ループのトリップカウントを特殊化したことで、既知の定数として JIT コンパイルする機能が適用されているためです。特殊化定数を使用するプログラムのメインループは、ランタイム変数を使用する例と比較して、およそ `inner_loop_size` 倍 (この例では 10 倍) の浮動小数点加算命令が生成されます。

## 浮動小数点計算における精度とパフォーマンスのトレードオフ

浮動小数点を使用するアプリケーションの開発者は、通常、次の 2 つのことを意識します。

- **精度:** 正確な計算結果に「近い」結果を生成します。
- **パフォーマンス:** 可能な限り高速に実行できるアプリケーションを作成します。

通常、これら 2 つの目的は相反します。しかし、適切なプログラミング手法と適切なコンパイラー・オプションを使用することで、そのトレードオフを制御できます。

詳細は、『インテル® コンパイラーの浮動小数点演算における結果の一貫性』(Dr. Martyn J. Corden & David Kreitzer 2018) を参照してください。

この節では、プログラマーがホストとデバイスで浮動小数点計算のセマンティクス (精度とパフォーマンス) を制御するメカニズム (コンパイラー・オプションとソースレベルの変更) について説明します。最初に OpenMP* と SYCL* プログラムのコンパイラー・オプションについて、次に SYCL* のソースレベルの変更について説明します。

### OpenMP*

OpenMP* では、`-fp-model` オプションを使用してホストとデバイスの浮動小数点計算のセマンティクスを制御できます。

- `-fp-model=precise`

このオプションは、浮動小数点計算を実装する際に、安全な値を保証する最適化を行うようコンパイラーに指示します。浮動小数点計算の結果を変える可能性のある最適化 (再関連付け、逆数による乗算、ゼロ・フォールディングなど) を無効にします。`-fp-model=precise` によって精度が向上すると、パフォーマンスが低下する可能性があります。

- `-fp-model=fast`

このオプションは、`-O2` 以上の最適化におけるホストとデバイス両方のコンパイルのデフォルトです。このオプションは、浮動小数点計算を実装する際に、積極的な最適化を行うようコンパイラーに指示します。この最適化で速度は向上しますが、浮動小数点計算の精度と再現性に影響する可能性があります。

C/C++ では、`-fp-model=fast` オプションと `-ffast-math` オプションは同等です。このオプション (`-O2` 以上) を使用すると、C/C++ フロントエンドによって 7 つの高速演算フラグ (`nnan`、`ninf`、`nsz`、`arcp`、`contract`、`afn`、`reassoc`) が設定されます。(LLVM における `fast-math` オプションについては、https://llvm.org/docs/LangRef.html#fast-math-flags (英語) を参照してください。)

一方、Fortran では、言語規則によってデフォルトで `nnan` フラグ (NaN なし) を設定できないことが規定されています。したがって、`-fp-model=fast` オプション (`-O2` 以上) では、6 つの高速演算フラグ (`ninf`、`nsz`、`arcp`、`contract`、`afn`、`reassoc`) が設定されます。Fortran で 7 つのフラグを設定するには、`-ffast-math` オプションを使用します。

`-fp-model=fast` (または `-ffast-math`) は、インテル® GPU (インテル® データセンター GPU マックス・シリーズ) ではネイティブの数学命令を有効にしないことに注意してください。ネイティブの数学命令では、`-fp-model=fast` よりも低い精度が得られます。インテル® GPU でネイティブの数学命令を有効にするには、次のように `-cl-fast-relaxed-math` オプションを使用します。

- `-Xopenmp-target-backend "-options -cl-fast-relaxed-math"`

`-fp-model=fast` (または `-ffast-math`) は、インテル® GPU (インテル® データセンター GPU マックス・シリーズ) ではネイティブの数学命令を有効にしないことに注意してください。GPU でネイティブの数学命令を有効にするには、`-Xopenmp-target-backend "-options -cl-fast-relaxed-math"` オプションでコンパイルする必要があります。ネイティブの数学命令では、`-fp-model=fast` よりも低い精度が得られます。

`-Xopenmp-target-backend "-options -cl-fast-relaxed-math"` は、デバイス向けコンパイル・ツール・チェーンのバックエンドに `-cl-fast-relaxed-math` オプションを渡します。`-cl-fast-relaxed-math` を指定すると、デバイスで一般的に使用される数学関数の精度が緩和されます。ネイティブ数学命令による精度要件が満たされている限り、多くの数学ライブラリー関数呼び出しでカーネルのパフォーマンスを向上する簡単な方法が提供されます。

`-cl-fast-relaxed-math` オプションは、プログラム全体のコンパイルに影響するため、結果の数値精度を細かく制御できなくなることに注意してください。

インテル® GPU は、単精度 (`float`、`real`) のネイティブ命令のみをサポートすることを忘れないでください。

**注 (OpenMP*):**

- コンパイルオプションに `-fp-model` が指定される場合 (`-fopenmp-targets=spir64="…"` オプション以外で)、`-fp-model` オプションはホストとデバイスの両方のコンパイルに適用されます。以下に示すように、ホストとデバイスのコンパイルに異なる `-fp-model` キーワードを指定できます。例えば、次のコンパイルコマンドでは、ホストとデバイスの両方のコンパイルに `-fp-model=precise` を指定しています。

  ```
  icx/icpx/ifx -O2 -fiopenmp -fopenmp-targets=spir64 -fp-model=precise
  ```

  以下に示すように、ホストとデバイスのコンパイルに異なる `-fp-model` キーワードを指定できます。

- ホストのコンパイルに `-fp-model=fast` を指定し、デバイスのコンパイルに `-fp-model=precise` を指定するには、次のようにします。

  ```
  icx/icpx/ifx -O2 -fiopenmp -fopenmp-targets=spir64="-fp-model=precise" -fp-model=fast
  ```

  または

  ```
  icx/icpx/ifx -O2 -fiopenmp -fopenmp-targets=spir64="-fp-model=precise"
  ```

  (`-O2` 以上ではデフォルトであるため、`-fp-model=fast` は必要ありません。)

- また、ホストのコンパイルに `-fp-model=precise` を指定し、デバイスのコンパイルに `-fp-model=fast` を指定するには、次のようにします。

```
icx/icpx/ifx -O3 -fiopenmp -fopenmp-targets=spir64=”-fp-model=fast” -fp-model=precise
```

- ホストのコンパイルに `-fp-model=fast` を指定し、デバイスのコンパイルに `relaxed-math` を指定するには、次のようにします。

```
icx/icpx/ifx -O2 -fiopenmp -fopenmp-targets=spir64 -Xopenmp-target-backend “-options -cl-fast-relaxed-math” -fp-model=fast
```

  または

```
icx/ifx -O2 -fiopenmp -fopenmp-targets=spir64 -Xopenmp-target-backend “-options -cl-fast-relaxed-math”
```

  (`-O2` 以上ではデフォルトであるため、`-fp-model=fast` は必要ありません。)

- ホストでの精度を調整するため、`fast`、`precise`、`relaxed-math` オプションと `-fimf-precision=low` (`medium` または `high`) を組み合わせることもできます。現在、デバイス向けに `-fimf-precision` オプションはサポートされていません。

以下の表では、OpenMP* の各種オプション (`precise`、`fast-math`、`relaxed math`) を (a) ホストとデバイスの両方 (2 列目) 、(b) ホストのみ (3 列目)、(c) デバイスのみ (4 列目) に設定する方法を示しています。

### OpenMP* - オプションのまとめ

| 浮動小数点セマンティクス | ホストとデバイスのコンパイルに適用 | ホストのコンパイルにのみ適用 | デバイスのコンパイルにのみ適用 |
|---|---|---|---|
| precice | -fp-model=precise | -fp-model=precise と -fiopenmp -fopenmp-targets=spir64="-fp-model=fast" を指定 | -fiopenmp -fopenmp-targets=spir64="-fp-model=precise" |
| fast-math | -fp-model=fast (デフォルト) | -fp-model=fast と -fiopenmp -fopenmp-targets=spir64="-fp-model= precise" を指定 | -fp-model=precise と -fiopenmp -fopenmp-targets=spir64="-fp-model=fast" を指定 |
| relaxed-math (ネイティブ命令) | デバイスにのみ適用 | デバイスにのみ適用 | -Xopenmp-target-backend "-options -cl-fast-relaxed-math" |

## SYCL*

SYCL* では、OpenMP* と同様に、ホストとデバイス両方のコンパイルに `-fp-model=fast` オプションと `-fp-model=precise` オプションを使用できます。

SYCL* では、`-fp-model=fast` オプションと `-ffast-math` オプションは同等であり、`-O2` 以上ではホストとデバイスの両方のコンパイルでデフォルトです。

デバイスコンパイル向けに `relaxed-math` を有効にするには、`-Xsycl-target-backend “-options -cl-fast-relaxed-math”` オプションを使用します。GPU でネイティブ数学命令を使用するには、このオプションを指定する必要があります。

SYCL* (およびインテル® GPU) は、単精度 (`float`、`real`) のネイティブ命令のみをサポートします。

**注 (SYCL*):**

- (`-Xsycl-target` オプション以外で) `-fp-model` コンパイルオプションを指定する場合、`-fp-model` オプションはホストとデバイスの両方のコンパイルに適用されます。以下に示すように、ホストとデバイスのコンパイルに異なる `-fp-model` キーワードを指定できます。例えば、次のコンパイルコマンドは、ホストとデバイスの両方のコンパイルに `-fp-model=precise` を指定しています。

  ```
  icx/icpx/ifx -fsycl -fp-model=precise
  ```

  以下に示すように、ホストとデバイスのコンパイルに異なる `-fp-model` キーワードを指定できます。

- ホストのコンパイルに `-fp-model=fast` を指定し、デバイスのコンパイルに `-fp-model=precise` を指定するには、次のようにします。

  ```
  icx/icpx -fsycl -Xsycl-target-frontend "-fp-model=precise" -fp-model=fast
  ```

  または

  ```
  icx/icpx -fsycl -Xsycl-target-frontend "-fp-model=precise"
  ```

  (`-O2` 以上ではデフォルトであるため、`-fp-model=fast` は必要ありません。)

- また、ホストのコンパイルに `-fp-model=precise` を指定し、デバイスのコンパイルに `-fp-model=fast` を指定するには、次のようにします。

  ```
  icx/icpx -fsycl -Xsycl-target-frontend "-fp-model=fast" -fp-model=precise
  ```

- ホストのコンパイルに `-fp-model=fast` を指定し、デバイスのコンパイルに `relaxed-math` を指定するには、次のようにします。

  ```
  icx/icpx -fsycl -Xsycl-target-backend “-options -cl-fast-relaxed-math” -fp-model=fast
  ```

  または

  ```
  icx/icpx -fsycl -Xsycl-target-backend “-options -cl-fast-relaxed-math”
  ```

  (`-O2` 以上ではデフォルトであるため、`-fp-model=fast` は必要ありません。)

以下の表では SYCL* の各種オプション (`precise`、`fast-math`、`relaxed math`) を (a) ホストとデバイスの両方 (2 列目) 、(b) ホストのみ (3 列目)、(c) デバイスのみ (4 列目) に設定する方法を示しています。

### SYCL* - オプションのまとめ

| 浮動小数点のセマンティクス | ホストとデバイスのコンパイルに適用 | ホストのコンパイルにのみ適用 | デバイスのコンパイルにのみ適用 |
|---|---|---|---|
| `precise` | `-fp-model=precise` | `-fp-model=precise` と `-Xsycl-target-frontend "-fp-mode=fast"` を指定 | `-Xsycl-target-frontend "-fp-model=precise"` |
| `fast-math` | `-fp-model=fast` (デフォルト) | `-fp-model=fast` と `-Xsycl-target-frontend "-fp-model=precise"` を指定 | `-fp-model=precise` と `-Xsycl-target-frontend "-fp-model=fast"` を指定 |
| `relaxed-math` (ネイティブ命令) | デバイスにのみ適用 | デバイスにのみ適用 | `-Xsycl-target-backend "-options -cl-fast-relaxed-math"` |

## ガイドライン

使用するオプションを決定するガイドラインを示します。

- 矛盾するオプションを指定しない。結果は予測できません。
- ホストとデバイスで最も一般的なオプションは、`-fp-model=fast` です。
- デバイス上で最良のパフォーマンスを得るには `relaxed-math` を使用します。
- 最も高い精度が必要な場合 `-fp-model=precise` を使用します。

## 例: log 関数

数学ライブラリー (プログラミング言語の標準ライブラリーのコンポーネント) には、指数関数、対数関数、累乗関数、三角関数など、一般的な数学関数 (またはサブルーチン) が含まれます。

しかし、数学ライブラリーの実装が異なると、精度が異なったり、同じ方法で丸めができない場合があります。数学ライブラリー関数が返す値は、アルゴリズムと最適化が変更されると、コンパイラーのリリースごとに異なる場合があります。

数学ライブラリー関数の精度は、コンパイルオプションやソースコードで制御できます。log (自然対数) 数学関数の例を使用してこれを説明します。

## OpenMP* / C++ (test_log_omp.cpp)

以下は、デバイス上で (OpenMP* ターゲット領域内で) `std::log` 関数を呼び出す OpenMP* プログラムです。このプログラムでは、一般的な数学関数の定義を含む `cmath` ヘッダーをインクルードします。

```
1.  #include <iostream>
2.  #include <assert.h>
3.  #include <chrono>
4.  #include <cmath>
5.  #if FP_SIZE == 32
6.    typedef float FP_TYPE;
7.    static constexpr FP_TYPE VALIDATION_THRESHOLD = 1e-3;
8.  #elif FP_SIZE == 64
9.    typedef double FP_TYPE;
10.   static constexpr FP_TYPE VALIDATION_THRESHOLD = 1e-6;
11. #endif
12. template<typename T>
13. void do_work (unsigned NELEMENTS, unsigned NREPETITIONS, T initial_value, T *res)
14. {
15. #pragma omp target teams distribute parallel for map(present,alloc:res[0:NELEMENTS])
16.   for (unsigned j = 0; j < NELEMENTS; j++)
17.   {
18.     T tmp = initial_value;
19.     for (unsigned i = 0; i < NREPETITIONS; ++i)
20.       tmp += std::log(tmp);
21.     res[j] = tmp;
22.   }
23. }
24. int main (int argc, char *argv[])
25. {
26.   static constexpr unsigned NELEMENTS = 64*1024*1024;
27.   static constexpr unsigned NREPETITIONS = 1024;
28. #pragma omp target
29.   { }
30.   FP_TYPE initial_value = 2;
31.   FP_TYPE ref_res = initial_value;
32.   for (unsigned i = 0; i < NREPETITIONS; ++i)
```

```
33.     ref_res += std::log(ref_res);
34.   std::cout << "reference result = " << ref_res << std::endl;
35.   {
36.     FP_TYPE * std_res = new FP_TYPE[NELEMENTS];
37.     assert (std_res != nullptr);
38.     std::chrono::duration<float, std::micro> elapsed;
39.     #pragma omp target data map(std_res[0:NELEMENTS])
40.     {
41.       auto tbegin = std::chrono::system_clock::now();
42.       do_work<FP_TYPE> (NELEMENTS, NREPETITIONS, initial_value, std_res);
43.       auto tend = std::chrono::system_clock::now();
44.       elapsed = tend - tbegin;
45.     }
46.     std::cout << "std::log result[0] = " << std_res[0] << std::endl;
47.     bool allequal = true;
48.     for (auto i = 1; i < NELEMENTS; ++i)
49.       allequal = allequal and std_res[0] == std_res[i];
50.     if (allequal)
51.     {
52.       if (std::abs(ref_res - std_res[0])/std::abs(ref_res) <
53.                                                 std::abs(VALIDATION_THRESHOLD))
54.         std::cout << "std::log validates. Total execution time is "<<elapsed.count()
55.                   << " us." << std::endl;
56.       else
57.         std::cout << "std::log does not validate (ref=" << ref_res
58.                   << " std_res=" << std_res[0]
59.                   << " mix=" << std::abs(ref_res - std_res[0])/std::abs(ref_res)<<")"
60.                   << std::endl;
61.     }
62.     else
63.       std::cout << "std::log does not validate, results are not equal." << std::endl;
64.     delete [] std_res;
65.   }
66.   return 0;
67. }
```

`test_log_omp.cpp` サンプルのコンパイルと実行コマンド:

```
$ icpx -O2 -fiopenmp -fopenmp-targets=spir64 test_log_omp.cpp -DREAL_ELEMENT
-DFP_SIZE=64 -fp-model=fast -fopenmp-version=51
$ export ZE_AFFINITY_MASK=0.0
$ ./a.out
```

## OpenMP* / Fortran (test_log_omp_f_mod.f90)

以下は、デバイス上で (OpenMP* ターゲット領域から) Fortran 組込み数学 `log` 関数を呼び出す OpenMP* Fortran モジュールです。

```
1.  MODULE test
2.
3.      USE ISO_C_BINDING
4.
5.  CONTAINS
6.
7.    SUBROUTINE log_real_sp (nelements, nrepetitions, initial_value, res)
    bind(C,NAME='log_real_sp')
8.      IMPLICIT NONE
9.      INTEGER(KIND=C_INT), VALUE :: nelements, nrepetitions
10.     REAL(C_FLOAT), VALUE :: initial_value
11.     REAL(C_FLOAT) :: res(0:nelements-1), tmp
12.     INTEGER :: i, j
13.
14.     !$OMP TARGET TEAMS DISTRIBUTE PARALLEL DO PRIVATE(tmp)
15.     DO j = 0, nelements-1
16.       tmp = initial_value
```

```
17.       DO i = 0, nrepetitions-1
18.         tmp = tmp + log(tmp)
19.       END DO
20.       res(j) = tmp
21.     END DO
22.     RETURN
23.   END SUBROUTINE log_real_sp
24.
25.   SUBROUTINE log_real_dp (nelements, nrepetitions, initial_value, res)
    bind(C,NAME='log_real_dp')
26.     IMPLICIT NONE
27.     INTEGER(KIND=C_INT), VALUE :: nelements, nrepetitions
28.     REAL(C_DOUBLE), VALUE :: initial_value
29.     REAL(C_DOUBLE) :: res(0:nelements-1), tmp
30.     INTEGER :: i, j
31.
32.     !$OMP TARGET TEAMS DISTRIBUTE PARALLEL DO PRIVATE(tmp)
33.     DO j = 0, nelements-1
34.       tmp = initial_value
35.       DO i = 0, nrepetitions-1
36.         tmp = tmp + log(tmp)
37.       END DO
38.       res(j) = tmp
39.     END DO
40.     RETURN
41.   END SUBROUTINE log_real_dp
42.
43. END MODULE test
```

次のコマンドで `test_log_omp_f.f90` をコンパイルします。

```
$ ifx -c -O2 -fiopenmp -fopenmp-targets=spir64 test_log_omp_f_mod.f90
```

## SYCL* (test_log_sycl.cpp)

SYCL* では、呼び出す数学関数を選択することで、ソースレベルで浮動小数点セマンティクスを制御できます。例えば、SYCL* プログラムで次の 3 つの異なる `log` 関数を呼び出す例を考えてみます。

- **`std::log`:** C++ 標準ライブラリーの `log` 関数を参照します。実装の選択は、コンパイルオプション (`-fp-model` と `-cl-fast-relaxed-math`) の規定によって決定されます。例えば、ネイティブ数学命令を使用する実装を選択するには、`-cl-fast-relaxed-math` オプションを指定してコンパイルします。
- **`sycl::log`:** SYCL* が提供する `sycl` 名前空間の `log` 関数を参照します。`-cl-fast-relaxed-math` オプションが指定されていなくても、この関数はネイティブ命令を使用する場合があります。精度については SYCL* 仕様で詳しく説明されています。
- **`sycl::native::log`:** SYCL* が提供する `sycl` 名前空間のネイティブ `log` 関数を参照します。この関数はネイティブ命令を使用するため、`-cl-fast-relaxed-math` オプションは必要ありません。SYCL* (およびインテル® GPU) は、単精度 (`float`、`real`) のネイティブ命令のみをサポートします。精度については SYCL* 仕様で詳しく説明されています。

```
1.  #include <CL/sycl.hpp>
2.  #include <iostream>
3.  #include <assert.h>
4.  #include <chrono>
5.  #include <cmath>
6.  #if FP_SIZE == 32
7.    typedef float FP_TYPE;
8.    static constexpr FP_TYPE VALIDATION_THRESHOLD = 1e-3;
9.  #elif FP_SIZE == 64
10.   typedef double FP_TYPE;
11.   static constexpr FP_TYPE VALIDATION_THRESHOLD = 1e-6;
12. #endif
```

```
13. template<typename T>
14. void do_work_std (sycl::queue &q, unsigned NELEMENTS, unsigned NREPETITIONS, T
    initial_value, T *res)
15. {
16.   q.submit([&](sycl::handler& h) {
17.     h.parallel_for(NELEMENTS, [=] (auto j)
18.     {
19.     FP_TYPE tmp = initial_value;
20.     for (unsigned i = 0; i < NREPETITIONS; ++i)
21.       tmp += std::log(tmp);
22.       res[j] = tmp;
23.     });
24.   }).wait();
25. }
26. template<typename T>
27. void do_work_sycl (sycl::queue &q, unsigned NELEMENTS, unsigned NREPETITIONS, T
    initial_value, T *res)
28. {
29.   q.submit([&](sycl::handler& h) {
30.     h.parallel_for(NELEMENTS, [=] (auto j)
31.     {
32.       FP_TYPE tmp = initial_value;
33.       for (unsigned i = 0; i < NREPETITIONS; ++i)
34.         tmp += sycl::log(tmp);
35.       res[j] = tmp;
36.     });
37.   }).wait();
38. }
39. # if FP_SIZE == 32
40. template<typename T>
41. void do_work_sycl_native (sycl::queue &q, unsigned NELEMENTS, unsigned NREPETITIONS,
    T initial_value, T *res)
42. {
43.   q.submit([&](sycl::handler& h) {
44.     h.parallel_for(NELEMENTS, [=] (auto j)
45.     {
46.       FP_TYPE tmp = initial_value;
47.       for (unsigned i = 0; i < NREPETITIONS; ++i)
48.         tmp += sycl::native::log(tmp);
49.       res[j] = tmp;
50.     });
51.   }).wait();
52. }
53. # endif
54. int main (int argc, char *argv[])
55. {
56.   static constexpr unsigned NELEMENTS = 64*1024*1024;
57.   static constexpr unsigned NREPETITIONS = 1024;
58.   sycl::device d (sycl::gpu_selector_v);
59.   sycl::queue q (d);
60.   q.submit([&](sycl::handler& h) {
61.     h.single_task ([=]() { });
62.   }).wait();
63.   FP_TYPE initial_value = 2;
64.   FP_TYPE ref_res = initial_value;
65.   for (unsigned i = 0; i < NREPETITIONS; ++i)
66.   ref_res += std::log(ref_res);
67.   std::cout << "reference result = " << ref_res << std::endl;
68.   {
69.   FP_TYPE * std_res = new FP_TYPE[NELEMENTS];
70.   assert (std_res != nullptr);
71.   std::chrono::duration<float, std::micro> elapsed;
72.   {
73.   auto * res = sycl::malloc_device<FP_TYPE>(NELEMENTS, q);
74.   auto tbegin = std::chrono::system_clock::now();
75.   do_work_std<FP_TYPE>(q, NELEMENTS, NREPETITIONS, initial_value, res);
76.   auto tend = std::chrono::system_clock::now();
77.   elapsed = tend - tbegin;
78.   q.memcpy (std_res, res, NELEMENTS*sizeof(FP_TYPE)).wait();
```

```
79.   sycl::free (res, q);
80.   }
81.   std::cout << "std::log result[0] = " << std_res[0] << std::endl;
82.   bool allequal = true;
83.   for (auto i = 1; i < NELEMENTS; ++i)
84.     allequal = allequal and std_res[0] == std_res[i];
85.   if (allequal)
86.   {
87.     if (std::abs(ref_res - std_res[0])/std::abs(ref_res) <
    std::abs(VALIDATION_THRESHOLD))
88.       std::cout << "std::log validates.Total execution time is "
89.                 << elapsed.count() << " us."<< std::endl;
90.     else
91.       std::cout << "std::log does not validate (ref=" << ref_res << " std_res="
92.                 << std_res[0] << " mix="
93.                 << std::abs(ref_res - std_res[0])/std::abs(ref_res) << ")"
94.                 << std::endl;
95.   }
96.   else
97.     std::cout << "std::log does not validate, results are not equal."<< std::endl;
98.   delete [] std_res;
99.   }
100.  {
101.  FP_TYPE * sycl_res = new FP_TYPE[NELEMENTS];
102.  assert (sycl_res != nullptr);
103.
104.  std::chrono::duration<float, std::micro> elapsed;
105.  {
106.  auto * res = sycl::malloc_device<FP_TYPE>(NELEMENTS, q);
107.  auto tbegin = std::chrono::system_clock::now();
108.  do_work_sycl<FP_TYPE>(q, NELEMENTS, NREPETITIONS, initial_value, res);
109.  auto tend = std::chrono::system_clock::now();
110.  elapsed = tend - tbegin;
111.  q.memcpy (sycl_res, res, NELEMENTS*sizeof(FP_TYPE)).wait();
112.  sycl::free (res, q);
113.  }
114.  std::cout << "sycl::log result[0] = " << sycl_res[0] << std::endl;
115.  bool allequal = true;
116.  for (auto i = 1; i < NELEMENTS; ++i)
117.    allequal = allequal and sycl_res[0] == sycl_res[i];
118.  if (allequal)
119.  {
120.    if (std::abs(ref_res - sycl_res[0])/std::abs(ref_res) <
    std::abs(VALIDATION_THRESHOLD))
121.      std::cout << "sycl::log validates. Total execution time is "
122.                << elapsed.count() << " us." << std::endl;
123.    else
124.      std::cout << "sycl::log does not validate (ref=" << ref_res
125.                << " sycl_res=" << sycl_res[0]
126.                << " mix=" << std::abs(ref_res - sycl_res[0])/std::abs(ref_res)
127.                << ")" << std::endl;
128.  }
129.  else
130.    std::cout << "sycl::log does not validate, results are not equal." << std::endl;
131.  delete [] sycl_res;
132.  }
133.# if FP_SIZE == 32
134.  {
135.  FP_TYPE * sycl_native_res = new FP_TYPE[NELEMENTS];
136.  assert (sycl_native_res != nullptr);
137.  std::chrono::duration<float, std::micro> elapsed;
138.  {
139.  auto * res = sycl::malloc_device<FP_TYPE>(NELEMENTS, q);
140.  auto tbegin = std::chrono::system_clock::now();
141.  do_work_sycl_native<FP_TYPE>(q, NELEMENTS, NREPETITIONS, initial_value, res);
142.  auto tend = std::chrono::system_clock::now();
143.  elapsed = tend - tbegin;
144.  q.memcpy (sycl_native_res, res, NELEMENTS*sizeof(FP_TYPE)).wait();
145.  sycl::free (res, q);
```

```
146.  }
147.  std::cout << "sycl::native::log result[0] = " << sycl_native_res[0] << std::endl;
148.  bool allequal = true;
149.  for (auto i = 1; i < NELEMENTS; ++i)
150.  allequal = allequal and sycl_native_res[0] == sycl_native_res[i];
151.  if (allequal)
152.  {
153.    if (std::abs(ref_res - sycl_native_res[0])/std::abs(ref_res) <
    std::abs(VALIDATION_THRESHOLD))
154.      std::cout << "sycl::native::log validates. Total execution time is "
155.                << elapsed.count() << " us."<< std::endl;
156.    else
157.      std::cout << "sycl::native::log does not validate (ref=" << ref_res
158.              << " sycl_native_res=" << sycl_native_res[0]
159.              << " mix=" << std::abs(ref_res - sycl_native_res[0])/std::abs(ref_res)
160.              << ")" << std::endl;
161.  }
162.  else
163.    std::cout << "sycl::native::log does not validate, results are not equal."
164.              << std::endl;
165.  delete [] sycl_native_res;
166.  }
167.# endif // FP_SIZE == 32
168.return 0;
169.}
```

`test_log_sycl.cpp` サンプルのコンパイルと実行コマンド:

```
$ icpx -fsycl -O2 test_log_sycl.cpp -DREAL_ELEMENT -DFP_SIZE=64 -fp-model=fast
$ export ZE_AFFINITY_MASK=0.0
$ ./a.out
```

## パフォーマンス検証

以下に、`log` 関数を呼び出す異なるプログラム (OpenMP* C++、OpenMP* Fortran および SYCL*) を実行した結果をまとめます。ここでは、インテル® データセンター GPU マックス・シリーズ (1 スタックのみ) を使用し、単精度の `log` 関数のパフォーマンスを取得しています。

### `log` のパフォーマンス - デフォルト精度 (Fast-math)

| バージョン | 時間 (秒) |
|---|---|
| OpenMP*/C++ (`std::log`) | 93,118 |
| OpenMP*/Fortran (`log`) | 94,342 |
| SYCL* (`std::log`) | 31,835 |
| SYCL* (`sycl::log`) | 31,644 |
| SYCL* (`sycl::native::log`) | 31,684 |

### `log` のパフォーマンス - Fast-math

| バージョン | 時間 (秒) |
|---|---|
| OpenMP*/C++ (`std::log`) | 93,181 |
| OpenMP*/Fortran (`log`) | 94,467 |
| SYCL* (`std::log`) | 31,657 |
| SYCL* (`sycl::log`) | 32,064 |
| SYCL* (`sycl::native::log`) | 31,452 |

### `log` のパフォーマンス – Precise

| バージョン | 時間 (秒) |
|---|---|
| OpenMP*/C++ (`std::log`) | 92,971 |
| OpenMP*/Fortran (`log`) | 94,444 |
| SYCL* (`std::log`) | 94,592 |
| SYCL* (`sycl::log`) | 94,852 |
| SYCL* (`sycl::native::log`) | 40,778 |

### `log` のパフォーマンス - Relaxed-math

| バージョン | 時間 (秒) |
|---|---|
| OpenMP*/C++ (`std::log`) | 35,251 |
| OpenMP*/Fortran (`log`) | 35,787 |
| SYCL* (`std::log`) | 31,314 |
| SYCL* (`sycl::log`) | 32,077 |
| SYCL* (`sycl::native::log`) | 32,141 |

**所見:**

- OpenMP* (C および Fortran) では、`std::log` はコンパイルオプション (`-fp-model`、`-cl-relaxed-math`) の規定に従います。`-cl-relaxed-math` オプションを指定すると、ネイティブ命令の `std::log` を呼び出します。
- SYCL* では、`sycl::log` は `-fp-model=fast` だけでもネイティブ命令を使用する場合があります。
- SYCL* では、`sycl::native::log` は常にネイティブマシン命令を使用します。`-cl-relaxed-math` オプションは必要ありません。
- OpenMP* と SYCL* では、`-cl-relaxed-math` オプションを指定すると、デバイスの `log` 関数にネイティブマシン命令が使用されます。
- `-fp-model=precise` は、正確な結果を生成しますが、パフォーマンスは低下します。

## 参考資料


1. 「インテル® コンパイラーの浮動小数点演算における結果の一貫性」(Dr. Martyn J. Corden and David Kreitzer (2018))
2. 『LLVM 言語リファレンス・マニュアル』 - 「Fast-Math フラグ」(英語)
3. 『DPC++ コンパイラー・ドキュメント』 - 「ユーザーズ・マニュアル」(英語)
4. 『インテル® oneAPI DPC++/C++ コンパイラー・デベロッパー・ガイドおよびリファレンス』 - 「サポートされるオプション」(英語)
5. 『インテル® oneAPI DPC++/C++ コンパイラー・デベロッパー・ガイドおよびリファレンス』 - 「Xopenmp-target」(英語)
6. 『インテル® Core™ プロセッサーおよびインテル® Xeon® プロセッサー向け OpenCL* デベロッパー・ガイド』(英語)
7. 『インテル® SDK for OpenCL* アプリケーション・デベロッパー・ガイド』(英語)
8. ターゲット・ツールチェーン・オプション (英語)
9. 『SYCL 2020 仕様』 - 「数学関数」
10. 『SYCL 2020 仕様』 - 「SYCL* ホストとデバイス向けの SYCL* ビルトイン関数」
11. 『OpenCL* 仕様バージョン 1.2』(英語) Khronos OpenCL* ワーキンググループ
12. 『インテル® プロセッサー・グラフィックス向け OpenCL* デベロッパー・ガイド』(英語)

# OpenMP* オフロードのチューニング・ガイド

LLVM ベースのインテル® C/C++ および Fortran コンパイラー (`icx`、`icpx`、および `ifx`) は、GPU への OpenMP* オフロードをサポートします。OpenMP* を使用する場合、プログラマーはソースコードにデバイス・ディレクティブを挿入して、アプリケーションの特定の領域を GPU にオフロードするようにコンパイラーに指示します。計算集約型のコードをオフロードすることで、適切なパフォーマンスを期待できます。

この節では、OpenMP* オフロードに関連する各種トピックと、オフロードされたコードのパフォーマンスを向上させる方法について説明します。


- OpenMP* ディレクティブ
- OpenMP* 実行モデル
- 用語
- OpenMP* アプリケーションのコンパイルと実行
- oneMKL の計算を GPU へオフロードする
- OpenMP* アプリケーションのパフォーマンスを解析するツール
- OpenMP* オフロードの最良の事例


## OpenMP* ディレクティブ

インテル® コンパイラー (`icx`、`icpx` および `ifx`) は、計算のオフロードとデバイスへのデータマッピングを制御するさまざまな OpenMP* デバイス・ディレクティブをサポートしています。これには以下のものがあります。

- `target`
- `teams`
- `distribute`
- `target data`
- `target enter data`
- `target exit data`
- `target update`
- `declare target`
- `dispatch`

`target` 構造は、デバイスで実行するコード領域と、データをデバイスにマッピングする方法を指定します。

`teams` 構造は、プライマリー・スレッドと複数のワーカースレッドで構成されるスレッドのチームを作成します。`teams` が `num_teams` 節なしで指定されると、チーム数は実装定義になります。

`distribute` 構造は、ループ反復を各チームのプライマリー・スレッドに分散させます。

`target data` 構造は、デバイスのデータ環境に変数をマップします。変数は、任意の `map` 節によって `target data` 領域の範囲にマッピングされます。

`target enter data` ディレクティブでは、デバイスへマップする変数を指定します。このディレクティブの `map` 節で指定されるマップタイプは `to` または `alloc` である必要があります。

`target exit data` ディレクティブは、変数がデバイスからマップ解除されることを指示します。このディレクティブの `map` 節で指定できるマップタイプは、`from`、`release` または `delete` です。

`target update` ディレクティブは、指定される `motion` 節にしたがって、デバイス上の変数値と元のホストの変数と一致させます。

`declare target` ディレクティブは、デバイスにマップされる変数、関数 (C/C++、Fortran) およびサブルーチン (Fortran) を指定します。

`declare variant` 指示句は、ベース関数の特殊バリアントを宣言し、そのバリアントが使用されるコンテキストを指定します。

`dispatch` 構造は、関数呼び出しに対しバリアントへの置き換えを行うかどうか制御します。

`map` 節は、元のホスト変数をデバイス上の対応する変数にどのようにマップするかを決定します。マップタイプには以下があります。

- `to`: ホストのオリジナル変数が `target` 領域の入り口でデバイスにコピーされます。
- `from`: デバイス上の変数の値は、`target` 領域の出口でデバイスから元のホスト変数にコピーされます。
- `tofrom`: 元のホスト変数の値は、`target` 領域の入り口でデバイスにコピーされ、`target` 領域の出口でホストにコピーバックされます。
- `alloc`: 元のホストの変数と同じタイプとサイズの変数をデバイスに割り当てます (値はホストからコピーされません)。

ディレクティブは組み合わせることができます。例えば、次のように組み合わせることができます。

- `target teams`
- `target teams distribute`
- `target teams distribute parallel for`
- `target teams distribute parallel for simd`

コンパイラーとランタイムがオフロードされたループ反復を GPU で実行するため最良の方法で分割できるよう、可能な限り複合ディレクティブを使用することを推奨します。

## OpenMP* 実行モデル

OpenMP* 実行モデルでは、ホストデバイスは単一ですが、複数のターゲットデバイスを持つことができます。デバイスは論理実行エンジンであり、固有のローカルストレージとデータ環境を持ちます。

インテル® データセンター GPU マックス・シリーズで実行する場合、2 つのタイルで構成される 1 つの GPU デバイス、または各タイルをそれぞれデバイスと見なすことができます。

OpenMP* ではホスト上で実行を開始します。ホストスレッドが `target` 構造に到達すると、データはホストからデバイスに転送され (例えば、`map` 節で指定されている場合)、構造内のコードはデバイスにオフロードされます。`target` 領域の終端では、データはデバイスからホストに転送されます (指定されていれば)。

デフォルトでは、ホストスレッドは `target` 領域が完了するのを待って処理を続行します。`target` 構造に `nowait` 節が指定されると、ホストスレッドは `target` 領域の完了を待機せずに処理を続行します。つまり、`nowait` 節を使用すると `target` 領域を非同期に実行できます。

非同期に実行されるコード領域間の同期は、`taskwait` ディレクティブ、`depend` 節、(暗黙および明示的な) バリア、または他の同期メカニズムによって行うことができます。

## 用語

ここでは、オフロードされた並列ループの反復処理を説明するため、OpenMP* と SYCL* の用語を同じ意味で使用しています。

「スレッドのマッピングと GPU 占有率」で説明したように、GPU にオフロードされた並列ループ (実行レンジ) の反復は、work-group、sub-group、そして work-item に分割されます。ND-range はすべての実行範囲を表しており、同じサイズの work-group に分割されます。work-group は、1 次元、2 次元、または 3 次元の work-item の集合です。各 work-group は sub-group に分割できます。sub-group は、SIMD ベクトルとして同時に処理される連続した work-item の短い範囲を表します。

次の表は、SYCL* の概念が OpenMP* と CUDA* の概念とどのように関連するか示しています。

| SYCL* | OpenMP* | CUDA* |
|---|---|---|
| ワーク項目 | OpenMP* スレッドまたは SIMD レーン | CUDA* スレッド |
| ワークグループ | チーム | スレッドブロック |
| ワークグループ・サイズ | チームサイズ | スレッド・ブロック・サイズ |
| ワークグループ数 | チーム数 | スレッドブロック数 |
| サブグループ | SIMD チャンク (`simdlen` = 8、16、32) | ワープ (`size` = 32) |
| ワークグループあたりのワーク項目の最大数 | スレッドリミット | スレッドブロックあたりの CUDA* スレッドの最大数 |

## OpenMP* アプリケーションのコンパイルと実行

インテル® GPU への OpenMP*オフロードを有効にするには、次のコンパイルオプションを使用します。C/C++ と Fortran の両方に適用できます。

```
-fiopenmp -fopenmp-targets=spir64
```

デフォルトでは、インテル® コンパイラーはプログラムを SPIR-V* と呼ばれる中間言語に変換し、コンパイルプロセスで生成されたバイナリーに格納します。実行時に SPIR-V* コードをプラットフォームのアセンブリー・コードに変換することで、いずれのハードウェア・プラットフォームでもコードを実行できます。このプロセスを JIT (Just-In-Time) コンパイルと呼びます。

次のコンパイルオプションを追加して、コンパイラー最適化レポートの出力を有効にします。

```
-qopt-report=3 -O3
```

**注:**

`-qopenmp` は、`-fiopenmp` と等価であり、この 2 つのオプションには互換性があります。

## 事前コンパイル (AOT)

インテル® データセンター GPU マックス・シリーズの事前コンパイル (AOT) コンパイルには、次のようなコンパイルオプション (`-Xs`) を使用します。このオプションは、C/C++ と Fortran 両方で使用できます。

```
-fiopenmp -fopenmp-targets=spir64_gen -Xs "-device pvc"
```

## OpenMP* ランタイムルーチン

デバイスに関連するランタイムルーチンには次のものがあります。

```
omp_target_alloc
omp_target_free
omp_target_memcpy
```

次のランタイムルーチンは、インテル® コンパイラーの拡張としてサポートされます。

```
omp_target_alloc_host
omp_target_alloc_device
omp_target_alloc_shared
```

`omp_target_free` を呼び出すと、インテル拡張のルーチンで割り当てたメモリーも解放できます。

`icx`、`icpx` および `ifx` コンパイラーでサポートされる OpenMP* 機能は以下を参照してください。

- インテル® oneAPI DPC++/C++ コンパイラーにおける OpenMP* 機能と拡張のサポート
- インテル® Fortran コンパイラーで実装される Fortran 言語と OpenMP* 機能

## 環境変数

ここでは、プログラムのデバッグやパフォーマンス向上に役立つ環境変数を紹介します。

環境変数に関する追加情報は以下を参照してください。

- 『インテル® oneAPI DPC++/C++ コンパイラー・デベロッパー・ガイドおよびリファレンス』 - 「サポートされる環境変数」 (英語)
- 『インテル® oneAPI プログラミング・ガイド』 - 「デバッグツール」
- LLVM* OpenMP* ランタイム (英語)
- レベルゼロプラグイン向けのデバッグ変数 (英語)

### LIBOMPTARGET_DEBUG=1

ライブラリーからのデバッグ情報出力を有効にします。

### LIBOMPTARGET_DEVICES=<DeviceKind>

サブデバイスをユーザーにどのように公開するか制御します。

```
<DeviceKind> := DEVICE | SUBDEVICE | SUBSUBDEVICE | device | subdevice | subsubdevice
```

`DEVICE/device`: OpenMP* デバイスとしてレポートされるのはトップレベルのデバイスのみです。`subdevice` 節はオプションとしてサポートされます。

`SUBDEVICE/subdevice`: 第 1 レベルのサブデバイスのみが OpenMP* デバイスとしてサポートされ、`subdevice` 節は無視されます。

`SUBSUBDEVICE/subsubdevice`: 第 2 レベルのサブデバイスのみが OpenMP* デバイスとしてサポートされ、`subdevice` 節は無視されます。レベルゼロ・バックエンドを使用するインテル® GPU で、サブのサブデバイスをタイル内の単一計算スライスとして制限するには、追加の GPU 計算ランタイム環境変数 `CFESingleSliceDispatchCCSMode=1` も設定する必要があります。

デフォルトは、`<DeviceKind>=device` です。

### LIBOMPTARGET_INFO=<Num>

ユーザーが libomptarget から各種タイプのランタイム情報を要求できるようにします。詳細については、

https://openmp.llvm.org/design/Runtimes.html#libomptarget-info (英語) を参照してください。

### LIBOMPTARGET_LEVEL0_MEMORY_POOL=<Option>

再利用可能なメモリープールの構成を制御します。

```
<Option>     := 0 | <PoolInfoList>
<PoolInfoList> := <PoolInfo>[,<PoolInfoList>]
<PoolInfo>   := <MemType>[,<AllocMax>[,<Capacity>[,<PoolSize>]]]
<MemType>    := all | device | host | shared
<AllocMax>   := 正の整数または空 (MB 単位の最大割り当てサイズ)
<Capacity>   := 正の整数または空 (単一ブロックからの割り当て数)
<PoolSize>   := 正の整数または空 (MB 単位の最大プールサイズ)
```

プールは、単一ブロックから `<AllocMax>` サイズまで、少なくとも `<Capacity>` 個の割り当てに対応できるメモリーブロックのリストであり、合計サイズが `<PoolSize>` を超えないようにします。

### LIBOMPTARGET_LEVEL0_STAGING_BUFFER_SIZE=<Num>

ステージのバッファーサイズを `<Num>` に KB 単位で設定します。ステージバッファーは、ホストメモリーが USM ではない場合に、ホストとデバイス間のコピー操作を最適化するために使用されます。ステージバッファーは、ディスクリート・デバイスのみに使用され、デフォルトのバッファーサイズは 16KB です。

### LIBOMPTARGET_LEVEL_ZERO_COMMAND_BATCH=copy

`target` 領域のデータ転送コマンドのバッチ処理を可能にします。

`target` 構造に `map(to:)` 節が指定されている場合、この環境変数はホストからデバイスへ複数のデータ転送を同時に行うことを許可します。同様に、`target` 構造に `map(from:)` 節がある場合、この環境変数はデバイスからホストへ複数のデータ転送を同時に行うことを許可します。`map(tofrom:)` や `map()` は `map(to:)` と `map(from:)` に分割されることに注意してください。

### LIBOMPTARGET_LEVEL_ZERO_USE_IMMEDIATE_COMMAND_LIST=<Bool>

カーネル送信に即時コマンドリストの使用を有効/無効にします。

```
<Bool> := 1 | T | t | 0 | F | f
```

デフォルトでは、即値コマンドリストは無効 (0) になっています。

### LIBOMPTARGET_PLUGIN=<Name>

使用するオフロードプラグインの名称を指定します。

```
<Name> := LEVEL0 | OPENCL | X86_64 |  level0 | opencl | x86_64
```

デフォルトでは、オフロードプラグインは `LEVEL0` です。

### LIBOMPTARGET_PLUGIN_PROFILE=<Enable>[,<Unit>]

基本プラグインのプロファイルを有効にし、プログラムの終了時に結果を表示します。

```
<Enable> := 1 | T
<Unit>   := usec | unit_usec
```

デフォルトではプラグインのプロファイルは無効になっています。

`<Unit>` が指定されていない場合、マイクロ秒 (`usec`) がデフォルト単位になります。

### LIBOMPTARGET_PROFILE=<FileName>

`libomptarget.so` が Clang の `-ftime-trace` オプションと同様の時間プロファイルの出力を生成できるようにします。

### OMP_TARGET_OFFLOAD=MANDATORY

デバイス構造またはデバイス・メモリー・ルーチンに到達した際に、デバイスが利用できない、または実装でサポートされていない場合、プログラムの実行を終了することを指定します。

## 暗黙的または明示的にスケーリングを制御する環境変数

暗黙のスケーリングを無効にして、単一の GPU タイルのみを利用するには、`ZE_AFFINITY_MASK=0.0` を設定します。

明示的なスケーリングを有効にするには、`LIBOMPTARGET_DEVICES=subdevice` を設定します。

インテル® データセンター GPU マックス・シリーズの場合、デフォルトで暗黙のスケーリングは有効になっています。

## SYCL* 向けの環境変数

OpenMP* だけではなく、SYCL* プログラムの開発とデバッグに有効な環境変数 (`SYCL_PI_LEVEL_ZERO`) があります。詳細については、「インテル® oneAPI ツールキット 2023 における DPC++ ランタイム環境変数」をご覧ください。

### 参考資料

1. インテル® oneAPI DPC++/C++ コンパイラーにおける OpenMP* 機能と拡張のサポート
2. インテル® Fortran コンパイラーで実装される Fortran 言語と OpenMP* 機能
3. 『インテル® oneAPI DPC++/C++ コンパイラー・デベロッパー・ガイドおよびリファレンス』 - 「サポートされる環境変数」 (英語)
4. 『インテル® oneAPI プログラミング・ガイド』 - 「デバッグツール」
5. LLVM* OpenMP* ランタイム (英語)
6. レベルゼロプラグイン向けのデバッグ変数 (英語)
7. インテル® oneAPI ツールキット 2023 における DPC++ ランタイム環境変数

## oneMKL の計算を GPU へオフロードする

oneAPI マス・カーネル・ライブラリー (oneMKL) は、大規模な計算問題を解決するソフトウェア・アプリケーションにおける数学ルーチンのパフォーマンスを向上させます。oneMKL は BLAS および LAPACK 線形代数ルーチン、高速フーリエ変換、ベクトル数学関数、乱数生成関数、およびその他の機能を提供します。

oneMKL の配布物には、oneMKL ルーチンのさまざまな呼び出し例を含む `examples` ディレクトリーが含まれています。

インテル® oneAPI マス・カーネル・ライブラリーの詳細については、以下を参照してください。

- インテル® oneAPI マス・カーネル・ライブラリー・デベロッパー・リファレンス - C (英語)
- インテル® oneAPI マス・カーネル・ライブラリー・デベロッパー・リファレンス - Fortran (英語)
- バッチ GEMM 操作の紹介 (英語)

### oneMKL OpenMP* オフロードを使用する際のコンパイルコマンドとリンクコマンド

この項で説明する内容は、Linux* 固有のものです。Windows* 向けの情報、および詳細については、インテル® oneAPI マス・カーネル・ライブラリーのリンクライン・アドバイザー (英語) を参照してください。

**注:**

- 次に示すリンクコマンドでは、oneMKL ライブラリーを動的リンクしています。
- oneMKL LP64 ライブラリーは 32 ビット整数型、oneMKL ILP64 ライブラリーは 64 ビット整数型 ($2^{31}$-1 要素を持つ大規模配列のインデックスに必要) で配列のインデックスを作成します。

### C/C++ (Linux*)

OpenMP* スレッドを使用して、**32 ビット整数**で oneMKL C/C++ API を呼び出す C/C++ プログラムのコンパイルおよびリンクコマンドは次のようになります。

**コンパイル:**
```
$ icpx -fiopenmp -fopenmp-targets=spir64 -qmkl -c source.cpp
```

**リンク:**
```
$ icpx -fiopenmp -fopenmp-targets=spir64 -qmkl -lOpenCL source.o
```

プログラムが **64 ビット整数**で oneMKL C/C++ API を呼び出す場合のコンパイルとリンクは次のようになります。

**コンパイル:**
```
$ icpx -fiopenmp -fopenmp-targets=spir64 -qmkl -DMKL_ILP64 -c source.cpp
```

**リンク:**
```
$ icpx -fiopenmp -fopenmp-targets=spir64 -qmkl -lOpenCL source.o
```

## Fortran (Linux*)

OpenMP* スレッドを使用して、**32 ビット整数**で oneMKL Fortran API を呼び出す Fortran プログラムのコンパイルおよびリンクコマンドは次のようになります。

**コンパイル:**
```
$ ifx -fiopenmp -fopenmp-targets=spir64 -qmkl -fpp -free -c source.f
```

**リンク:**
```
$ ifx -fiopenmp -fopenmp-targets=spir64 -qmkl -lOpenCL source.o
```

プログラムが **64 ビット整数**で oneMKL Fortran API を呼び出す場合のコンパイルとリンクは次のようになります。

**コンパイル:**
```
$ ifx -fiopenmp -fopenmp-targets=spir64 -qmkl -i8 -fpp -free -c source.f
```

**リンク:**
```
$ ifx -fiopenmp -fopenmp-targets=spir64 -qmkl -lOpenCL source.o
```

## `-qmkl` オプション使用時の注意

`-qmkl` オプション (`-qmkl=parallel` と同等) を使用して、指定するスレッド化オプションに応じて、特定の oneMKL スレッド化レイヤーとリンクします。

- `-fiopenmp`: インテル® コンパイラーの OpenMP* スレッド化レイヤー
- `-tbb`: oneTBB スレッド化レイヤー

`-qmkl=sequential` を使用して、oneMKL ランタイム変数のシリアルバージョンとリンクします。

`-qmkl=parallel/sequential` は、CPU のスレッド化のみに影響することに注意してください。オフロードされた oneMKL 計算は常に適切に並列化され、可能な限り多くの GPU ベクトルエンジンを占有します。

## oneMKL 計算をオフロードする OpenMP* ディレクティブ

OpenMP* ディレクティブを使用して、oneMKL 計算を GPU にオフロードするには 2 つの方法があります。

1 つは、インテル拡張機能の `target variant dispatch` ディレクティブを使用する方法です。次に示すように、oneMKL ルーチンの呼び出しを `target variant dispatch` 構造の中に配置します。この例では、配列 `A`、`B`、および `C` は oneMKL ルーチンの `cblas_dgemm` を呼び出す前に、デバイスにマップされています。`use_device_ptr(A,B,C)` 節は、`A`、`B`、および `C` がデバイス上のストレージ・オブジェクトを指すことを示すため、`target variant dispatch` ディレクティブに指定されています。

cblas_dgemm が呼び出されると、A、B、および C に対応するデバイスポインターが引数として渡され、A、B、および C のデバイスコピーが計算で使用されます。

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#include "mkl.h"
#include "mkl_omp_offload.h"

#define min(x,y) (((x) < (y)) ? (x) : (y))
#define EPSILON 0.0001

int main()
{
    double *A, *B, *C, *C_fl;
    int64_t m, n, k;
    double alpha, beta;
    double sum;
    int64_t i, j, q;
    int fail;

    printf ("\n This example computes real matrix C=alpha*A*B+beta*C using \n"
            " Intel oneMKL function dgemm, where A, B, and  C are matrices and \n"
            " alpha and beta are double precision scalars\n\n");

    m = 2000, k = 200, n = 1000;
    printf (" Initializing data for matrix multiplication C=A*B for matrix \n"
            " A(%li x %li) and matrix B(%li x %li)\n\n", m, k, k, n);
    alpha = 1.0; beta = 0.0;

    printf (" Allocating memory for matrices aligned on 64-byte boundary for better
\n"
            " performance \n\n");
    A = (double *)mkl_malloc( m * k * sizeof( double ), 64 );
    B = (double *)mkl_malloc( k * n * sizeof( double ), 64 );
    C = (double *)mkl_malloc( m * n * sizeof( double ), 64 );
    C_fl = (double *)mkl_malloc( m*n*sizeof( double ), 64 );

    if (A == NULL || B == NULL || C == NULL || C_fl == NULL) {
      printf( "\n ERROR: Cannot allocate memory for matrices. Exiting... \n\n");
      return 1;
    }

    printf (" Intializing matrices \n\n");
    for (i = 0; i < (m*k); i++) {
        A[i] = (double)(i+1);
    }

    for (i = 0; i < (k*n); i++) {
        B[i] = (double)(-i-1);
    }

    for (i = 0; i < (m*n); i++) {
        C[i] = 0.0;
        C_fl[i] = 0.0;
    }

    printf (" Computing matrix product using Intel oneMKL dgemm function via CBLAS
interface \n\n");

    #pragma omp target data map(to: A[0:m*k], B[0:k*n]) map(tofrom: C[0:m*n])
    {
       #pragma omp target variant dispatch use_device_ptr(A, B, C)
       {
          cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                      m, n, k, alpha, A, k, B, n, beta, C, n);
       }
```

```
64.     }
65.
66.     printf ("\n Top left corner of matrix C: \n");
67.     for (i=0; i<min(m,6); i++) {
68.         for (j=0; j<min(n,6); j++) {
69.             printf ("%12.5G", C[j+i*n]);
70.         }
71.         printf ("\n");
72.     }
73.
74.     printf (" Computing matrix product using for-loops \n");
75.
76.     for (i = 0; i < m; i++) {
77.         for (j = 0; j < n; j++) {
78.             sum = 0.0;
79.             for (q = 0; q < k; q++) {
80.                 sum += A[k*i+q] * B[n*q+j];
81.             }
82.             C_fl[n*i+j] = alpha * sum + beta * C_fl[n*i+j];
83.         }
84.     }
85.
86.     printf ("\n Top left corner of matrix C_fl: \n");
87.     for (i=0; i<min(m,6); i++) {
88.         for (j=0; j<min(n,6); j++) {
89.             printf ("%12.5G", C_fl[j+i*n]);
90.         }
91.         printf ("\n");
92.     }
93.
94.     printf (" Computations completed. Verifying... \n\n");
95.
96.     fail = 0;
97.     for (i = 0; i < (m*n); i++) {
98.         if (fabs(C[i] - C_fl[i]) > EPSILON) {
99.             fail = 1;
100.            break;
101.      }
102.    }
103.
104.    if (fail)
105.        printf ("\n **** FAIL **** \n");
106.    else
107.        printf ("\n **** PASS **** \n");
108.
109.    printf ("\n Deallocating memory \n\n");
110.    mkl_free(A);
111.    mkl_free(B);
112.    mkl_free(C);
113.
114.    return fail;
115.}
```

oneMKL の計算を GPU にオフロードすることをコンパイラーに知らせるもう 1 つの方法は、次のように OpenMP* 5.1 の `dispatch` ディレクティブを使用することです。この例では、配列 `A`、`B`、および `C` は oneMKL ルーチンの `cblas_dgemm` を呼び出す前に、デバイスにマップされています。`cblas_dgemm` が呼び出されると、`A`、`B`、および `C` に対応するデバイスポインターが引数として渡され、`A`、`B`、および `C` のデバイスコピーが計算で使用されます。

`dispatch` ディレクティブに `use_device_ptr` 節は必要ありません。OpenMP* 5.1 では、oneMKL ルーチンで必要なデバイスポインターのリストは、GPU バリアント関数が宣言される oneMKL OpenMP* オフロード・ヘッダー・ファイル `mkl_omp_offload.h` に含まれます。ユーザーは、oneMKL ルーチンを呼び出す前に、oneMKL ヘッダーファイルで使用するデバイスポインターのリストを確認し、デバイスから対応する配列にアクセスできることを確認する必要があります。

このサンプルをコンパイルするには、コンパイルオプション `-fopenmp-version=51` を追加する必要があります。

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#include "mkl.h"
#include "mkl_omp_offload.h"

#define min(x,y) (((x) < (y)) ? (x) : (y))
#define EPSILON 0.0001

int main()
{
    double *A, *B, *C, *C_fl;
    int64_t m, n, k;
    double alpha, beta;
    double sum;
    int64_t i, j, q;
    int fail;

    printf ("\n This example computes real matrix C=alpha*A*B+beta*C using \n"
            " Intel oneMKL function dgemm, where A, B, and  C are matrices and \n"
            " alpha and beta are double precision scalars\n\n");

    m = 2000, k = 200, n = 1000;
    printf (" Initializing data for matrix multiplication C=A*B for matrix \n"
            " A(%li x %li) and matrix B(%li x %li)\n\n", m, k, k, n);
    alpha = 1.0; beta = 0.0;

    printf (" Allocating memory for matrices aligned on 64-byte boundary for better
\n"
            " performance \n\n");
    A = (double *)mkl_malloc( m * k * sizeof( double ), 64 );
    B = (double *)mkl_malloc( k * n * sizeof( double ), 64 );
    C = (double *)mkl_malloc( m * n * sizeof( double ), 64 );
    C_fl = (double *)mkl_malloc( m*n*sizeof( double ), 64 );

    if (A == NULL || B == NULL || C == NULL || C_fl == NULL) {
      printf( "\n ERROR: Cannot allocate memory for matrices.Exiting...\n\n");
      return 1;
    }

    printf (" Intializing matrices \n\n");
    for (i = 0; i < (m*k); i++) {
        A[i] = (double)(i+1);
    }

    for (i = 0; i < (k*n); i++) {
        B[i] = (double)(-i-1);
    }

    for (i = 0; i < (m*n); i++) {
        C[i] = 0.0;
        C_fl[i] = 0.0;
    }

    printf (" Computing matrix product using Intel oneMKL dgemm function via CBLAS
interface \n\n");

    #pragma omp target data map(to: A[0:m*k], B[0:k*n]) map(tofrom: C[0:m*n])
    {
       #pragma omp target variant dispatch use_device_ptr(A, B, C)
       cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                   m, n, k, alpha, A, k, B, n, beta, C, n);
    }

    printf ("\n Top left corner of matrix C: \n");
```

```
65.     for (i=0; i<min(m,6); i++) {
66.         for (j=0; j<min(n,6); j++) {
67.             printf ("%12.5G", C[j+i*n]);
68.         }
69.         printf ("\n");
70.     }
71.
72.     printf (" Computing matrix product using for-loops \n");
73.
74.     for (i = 0; i < m; i++) {
75.         for (j = 0; j < n; j++) {
76.             sum = 0.0;
77.             for (q = 0; q < k; q++) {
78.                 sum += A[k*i+q] * B[n*q+j];
79.             }
80.             C_fl[n*i+j] = alpha * sum + beta * C_fl[n*i+j];
81.         }
82.     }
83.
84.     printf ("\n Top left corner of matrix C_fl: \n");
85.     for (i=0; i<min(m,6); i++) {
86.         for (j=0; j<min(n,6); j++) {
87.             printf ("%12.5G", C_fl[j+i*n]);
88.         }
89.         printf ("\n");
90.     }
91.
92.     printf (" Computations completed.Verifying...\n\n");
93.
94.     fail = 0;
95.     for (i = 0; i < (m*n); i++) {
96.         if (fabs(C[i] - C_fl[i]) > EPSILON) {
97.             fail = 1;
98.             break;
99.       }
100.    }
101.
102.    if (fail)
103.        printf ("\n **** FAIL **** \n");
104.    else
105.        printf ("\n **** PASS **** \n");
106.
107.    printf ("\n Deallocating memory \n\n");
108.    mkl_free(A);
109.    mkl_free(B);
110.    mkl_free(C);
111.
112.    return fail;
113.}
```

**注:**

- oneMKL ルーチンは、計算を行う前に配列/行列がデバイス上に存在することを前提としています。そのため、ユーザーは oneMKL ルーチンを呼び出す前に、データをデバイスにマップするか、デバイス上で直接データを割り当てる必要があります。
- oneMKL ルーチンが `target variant dispatch` (または `dispatch`) 領域から呼び出されていない、またはオフロードが無効である場合、oneMKL の計算は CPU で実行されます。
- OpenMP* の `target variant dispatch` (または `dispatch`) 領域から呼び出せる oneMKL ルーチンは 1 回のみです。oneMKL ルーチンの呼び出しが 2 回連続する場合、呼び出しは別々の `target variant dispatch` (または `dispatch`) 構造に配置する必要があります。

## Fortran

Fortran コードから oneMKL ルーチンを呼び出す場合は、次の `include` 文を必ず追加する必要があります。

```
include "mkl_omp_offload.f90"
```

また、oneMKL Fortran API を **32 ビット整数**で呼び出す場合は、次のモジュールの `use` 文を追加します。

```
use onemkl_blas_omp_offload_lp64
```

一方、oneMKL Fortran API を **64 ビット整数**で呼び出す場合は、次のモジュールの `use` 文を追加します。

```
use onemkl_blas_omp_offload_ilp64
```

次の Fortran の例では、Fortran プログラムから DGEMM を呼び出す方法と、前述の `include` 文と `use` 文について説明しています。

```
1.  include "mkl_omp_offload.f90"
2.
3.          program DGEMM_MAIN
4.
5.  #if defined(MKL_ILP64)
6.          use onemkl_blas_omp_offload_ilp64
7.  #else
8.          use onemkl_blas_omp_offload_lp64
9.  #endif
10.         use omp_lib
11.         use iso_fortran_env
12.         implicit none
13.
14.         integer, parameter :: m = 20
15.         integer, parameter :: k = 5
16.         integer, parameter :: n = 10
17.         double precision   a(m,k), b(k,n), c1(m,n), c2(m,n)
18.         double precision   alpha, beta
19.         integer            i, j
20.
21.         print*
22.         print*,'   DGEMM EXAMPLE PROGRAM'
23.
24.         ! Initialize
25.
26.         alpha = 1.025
27.         beta  = 0.75
28.
29.         do i = 1, m
30.           do j = 1, k
31.             a(i,j) = (i-1) - (0.25 * k)
32.           end do
33.         end do
34.
35.         do i = 1, k
36.           do j = 1, n
37.             b(i,j) = -((i-1) + j)
38.           end do
39.         end do
40.
41.         do i = 1, m
42.           do j = 1, n
43.             c1(i,j) = 0.2 + i - j
44.             c2(i,j) = 0.2 + i - j
45.           end do
46.         end do
```

```
47.
48.
49.         ! ホストで DGEMM を実行
50.
51.         call DGEMM('N','N',m,n,k,alpha,a,m,b,k,beta,c1,m)
52.
53.         print *
54.         print *, 'c1 - After DGEMM host execution'
55.
56.         do i=1,m
57.            print 110, (c1(i,j),j=1,n)
58.         end do
59.         print*
60.
61.
62.         ! デバイスで DGEMM を実行
63.
64. !$omp target data map(to: a, b) map(tofrom: c2)
65.
66. !$omp dispatch
67.         call DGEMM('N','N',m,n,k,alpha,a,m,b,k,beta,c2,m)
68.
69. !$omp end target data
70.
71.         print *
72.         print *, 'c2 - After DGEMM device execution'
73.
74.         do i=1,m
75.            print 110, (c2(i,j),j=1,n)
76.         end do
77.         print *
78.
79.  110    format(7x,10(f10.2,2x))
80.
81.         end
```

上記の Fortran プログラムを **32 ビット整数**でコンパイルおよびリンクするには、次のコマンドを使用します。

```
$ ifx -fiopenmp -fopenmp-targets=spir64 -qmkl -fpp -free -c dgemm_dispatch_f.f90
$ ifx -fiopenmp -fopenmp-targets=spir64 -qmkl -fsycl -L${MKLROOT}/lib/intel64 -liomp5
-lsycl -lOpenCL -lstdc++ -lpthread -lm -ldl -lmkl_sycl dgemm_dispatch_f.o
```

上記の Fortran プログラムを **64 ビット整数**でコンパイルおよびリンクするには、次のコマンドを使用します。

```
$ ifx -fiopenmp -fopenmp-targets=spir64 -qmkl -m64 -DMKL_ILP64 -i8 -fpp -free -c
dgemm_dispatch_f.f90
$ ifx -fiopenmp -fopenmp-targets=spir64 -qmkl -fsycl -L${MKLROOT}/lib/intel64 -liomp5
-lsycl -lOpenCL -lstdc++ -lpthread -lm -ldl -lmkl_sycl dgemm_dispatch_f.o
```

C/C++ または Fortran プログラムから実行ファイル (`a.out`) を生成し、その実行ファイルを `onetrace` で実行すると、生成されたトレースファイルから「`Device Timing Results`」という見出しを検索します。その見出しの下に、oneMKL カーネルがリストされているはずです。このように、oneMKL 計算が GPU にオフロードされていることを確認できます。

実行コマンドの例:

```
$ export OMP_TARGET_OFFLOAD=MANDATORY
$ export ZE_AFFINITY_MASK=0.0
$ onetrace -h -d ./a.out
```

## oneMKL GEMM 呼び出しのバッチ処理

oneMKL ライブラリーには、複数の oneMKL 呼び出しを 1 つの呼び出しにまとめる「バッチ」ルーチンが含まれています。実行時に oneMKL は、全体のパフォーマンスを最適化するため、すべての行列操作をインテリジェントに実行します。

例えば、`cblas_dgemm` ルーチンは、2 つの一般行列 `a` と `b` の「行列-行列積」を計算して、結果を行列 `c` に返します。`cblas_dgemm` のインターフェイスを以下に示します。

```
void cblas_dgemm (const CBLAS_LAYOUT layout,
const CBLAS_TRANSPOSE transa, const CBLAS_TRANSPOSE transb,
const MKL_INT m, const MKL_INT n, const MKL_INT k,
const double alpha, const double *a,
const MKL_INT lda, const double *b,
const MKL_INT ldb, const double beta,
double *c, const MKL_INT ldc);
```

`cblas_dgemm_batch` ルーチンは、`cblas_dgemm` ルーチンと似ていますが、`cblas_dgemm_batch` ルーチンは行列のグループを使用して「行列-行列操作」を行い一度に複数のグループを処理します。

`cblas_dgemm_batch` のインターフェイスを以下に示します。`cblas_dgemm` インターフェイスと似ていることに注意してください。ただし、行列の引数は行列へのポインターの配列として渡し、パラメーターはパラメーターの配列として渡します。

```
void cblas_dgemm_batch (const CBLAS_LAYOUT layout,
const CBLAS_TRANSPOSE* transa_array, const CBLAS_TRANSPOSE* transb_array,
const MKL_INT* m_array, const MKL_INT* n_array, const MKL_INT* k_array,
const double* alpha_array, const double **a_array,
const MKL_INT* lda_array, const double **b_array,
const MKL_INT* ldb_array, const double* beta_array,
double **c_array, const MKL_INT* ldc_array,
const MKL_INT group_count, const MKL_INT* group_size);
```

バッチ操作は次のように定義されています。

```
idx = 0
for i = 0 .. group_count - 1
   alpha_array[i] と beta_array[i] の alpha および beta
   for j = 0 .. group_size[i] - 1
      a, b, c 行列は、それぞれ a_array[idx], b_array[idx], and c_array[idx] にあります
      c := alpha*op(a)*op(b) + beta*c,
      idx = idx + 1
   end for
 end for
```

説明:

- `op(X)` は、`op(X) = X`、`op(X) = XT`、または `op(X) = XH` のいずれか
- `alpha` と `beta` は、`alpha_array` と `beta_array` のスカラー要素
- `a`、`b`、および `c` は、`m_array`、`n_array`、そして `k_arrayare` の要素である `m`、`n`、`k` の行列
- `op(a)` は `m-by-k matrix`
- `op(b)` は `k-by-n matrix`
- `C` は `m-by-n matrix`

- a、b、c は、それぞれ a_array、b_array、c_array が指すアドレスに格納されている行列を表します。a_array、b_array、および c_array のエントリー数は、total_batch_count に等しい group_size エントリーの合計です

異なる形状とパラメーターの乗算をグループにパッケージ化することでバッチ処理できます。各グループは、同じ形状 (同じ m、n、k) と同じパラメーターの行列乗算で構成されます。

バッチ API の基本前提は、バッチ内のすべての操作は (同一グループでも異なるグループでも) 互いに独立していることです。そのため、oneMKL はバッチ内の操作間の順序付けは保証せず、複数の操作を並列に実行します。

一般に、バッチサイズが大きくなるほど良い結果をもたらします。これにより、oneMKL は操作をより適切に並列化し、GPU 全体にワークを分散できます。

2 回の cblas_dgemm 呼び出しを、1 回の cblas_dgemm_batch 呼び出しに置き換える方法を説明します。次の例では、cblas_dgemm を 2 回呼び出しています。

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#include "mkl.h"
#include "mkl_omp_offload.h"

#define min(x,y) (((x) < (y)) ? (x) : (y))
#define epsilon 0.0000001f

bool compare(double x, double y)
{
    // x と y が等しければ true を返す
    return fabs(x - y) <= epsilon;
}

int main()
{
    double *A1, *B1, *C1, *C1_fl;
    double *A2, *B2, *C2, *C2_fl;
    int m, n, k, i, j, q;
    double alpha, beta;
    double sum;
    int fail;
    double t_start, t_end;

    m = 2000, k = 200, n = 1000;
    alpha = 1.0; beta = 0.0;

    printf (" Allocating memory for matrices aligned on 64-byte boundary for better \n"
            " performance \n\n");
    A1 = (double *)mkl_malloc (m*k*sizeof( double ), 64 );
    B1 = (double *)mkl_malloc (k*n*sizeof( double ), 64 );
    C1 = (double *)mkl_malloc (m*n*sizeof( double ), 64 );
    C1_fl = (double *)mkl_malloc (m*n*sizeof( double ), 64 );

    A2 = (double *)mkl_malloc (m*k*sizeof( double ), 64 );
    B2 = (double *)mkl_malloc (k*n*sizeof( double ), 64 );
    C2 = (double *)mkl_malloc (m*n*sizeof( double ), 64 );
    C2_fl = (double *)mkl_malloc (m*n*sizeof( double ), 64 );

    if (A1 == NULL || B1 == NULL || C1 == NULL || C1_fl == NULL ||
        A2 == NULL || B2 == NULL || C2 == NULL || C2_fl == NULL) {
      printf( "\n ERROR: Can't allocate memory for matrices. Aborting... \n\n");
      return 1;
```

```
46.     }
47.
48.     printf (" Intializing matrix data \n\n");
49.     for (i = 0; i < (m*k); i++) {
50.         A1[i] = A2[i] = (double)(i+1);
51.     }
52.
53.     for (i = 0; i < (k*n); i++) {
54.         B1[i] = B2[i] = (double)(-i-1);
55.     }
56.
57.     for (i = 0; i < (m*n); i++) {
58.         C1[i]    = C2[i]    = 0.0;
59.         C1_fl[i] = C2_fl[i] = 0.0;
60.     }
61.
62.     printf (" \nComputing matrix product using Intel MKL cblas_dgemm function \n");
63.
64.     t_start = omp_get_wtime();
65.
66.     #pragma omp target data                                  \
67.       map(to: A1[0:m*k], B1[0:k*n], A2[0:m*k], B2[0:k*n])  \
68.       map(tofrom: C1[0:m*n], C2[0:m*n])
69.     {
70.        #pragma omp dispatch
71.        cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
72.                    m, n, k, alpha, A1, k, B1, n, beta, C1, n);
73.
74.        #pragma omp dispatch
75.        cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
76.                    m, n, k, alpha, A2, k, B2, n, beta, C2, n);
77.     }
78.
79.     t_end = omp_get_wtime();
80.
81.     printf ("\n Top left corner of matrix C1: \n");
82.     for (i=0; i<min(m,6); i++) {
83.       for (j=0; j<min(n,6); j++) {
84.         printf ("%12.5G", C1[j+i*n]);
85.       }
86.       printf ("\n");
87.     }
88.
89.     printf ("\n Top left corner of matrix C2: \n");
90.     for (i=0; i<min(m,6); i++) {
91.       for (j=0; j<min(n,6); j++) {
92.         printf ("%12.5G", C2[j+i*n]);
93.       }
94.       printf ("\n");
95.     }
96.
97.     printf (" \nComputing matrix product using for-loops \n");
98.
99.     for (i = 0; i < m; i++) {
100.        for (j = 0; j < n; j++) {
101.            sum = 0.0;
102.            for (q = 0; q < k; q++)
103.                sum += A1[k*i+q] * B1[n*q+j];
104.            C1_fl[n*i+j] = sum;
105.        }
106.    }
107.
108.    for (i = 0; i < m; i++) {
109.        for (j = 0; j < n; j++) {
110.            sum = 0.0;
111.            for (q = 0; q < k; q++)
112.                sum += A2[k*i+q] * B2[n*q+j];
113.            C2_fl[n*i+j] = sum;
114.        }
```

```
115.    }
116.
117.    printf ("\n Top left corner of matrix C1: \n");
118.    for (i=0; i<min(m,6); i++) {
119.      for (j=0; j<min(n,6); j++) {
120.        printf ("%12.5G", C1_fl[j+i*n]);
121.      }
122.      printf ("\n");
123.    }
124.
125.    printf ("\n Top left corner of matrix C2: \n");
126.    for (i=0; i<min(m,6); i++) {
127.      for (j=0; j<min(n,6); j++) {
128.        printf ("%12.5G", C2_fl[j+i*n]);
129.      }
130.      printf ("\n");
131.    }
132.
133.    printf ("\n Computations completed. Verifying...\n\n");
134.
135.    fail = 0;
136.    for (i = 0; i < (m*n); i++) {
137.      if (! compare(C1[i], C1_fl[i]) || ! compare(C2[i], C2_fl[i])) {
138.          fail = 1;
139.          break;
140.      }
141.    }
142.
143.    if (fail) {
144.        printf (" **** FAIL **** \n");
145.    }
146.    else {
147.        printf(" time = %lf seconds\n", t_end - t_start);
148.        printf (" **** PASS **** \n");
149.    }
150.
151.    mkl_free(A1);
152.    mkl_free(B1);
153.    mkl_free(C1);
154.    mkl_free(C1_fl);
155.    mkl_free(A2);
156.    mkl_free(B2);
157.    mkl_free(C2);
158.    mkl_free(C2_fl);
159.
160.    return 0;
161.}
```

上記の `cblas_dgemm` の 2 つの呼び出しは、次の例に示すように 1 つの `cblas_dgemm_batch` に置き換え、一括してバッチ処理できます。同一パラメーター (`layout`、`transa`、`transb`、`m`、`n`、`k`、`alpha`、`lda`、`ldb`、`beta`、および `ldc`) で 2 回の行列乗算を行っているため、バッチはサイズ 2 の 1 つのグループで構成されていることに注意してください。ここで `total_batch_size` は 2 です。

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <math.h>
4.  #include <omp.h>
5.  #include "mkl.h"
6.  #include "mkl_omp_offload.h"
7.
8.  #define min(x,y) (((x) < (y)) ? (x) : (y))
9.  #define epsilon 0.0000001f
10.
11. bool compare(double x, double y)
12. {
13.     // x と y が等しければ true を返す
```

```
14.     return (fabs(x - y) <= epsilon);
15. }
16.
17. int main()
18. {
19.     double *A1, *B1, *C1, *C1_fl;
20.     double *A2, *B2, *C2, *C2_fl;
21.     int m, n, k, i, j, q;
22.     double alpha, beta;
23.     double sum;
24.     int fail;
25.     double t_start, t_end;
26.
27.     m = 2000, k = 200, n = 1000;
28.     alpha = 1.0; beta = 0.0;
29.
30.     printf (" Allocating memory for matrices aligned on 64-byte boundary for better
    \n"
31.             " performance \n\n");
32.     A1 = (double *)mkl_malloc (m*k*sizeof( double ), 64 );
33.     B1 = (double *)mkl_malloc (k*n*sizeof( double ), 64 );
34.     C1 = (double *)mkl_malloc (m*n*sizeof( double ), 64 );
35.     C1_fl = (double *)mkl_malloc (m*n*sizeof( double ), 64 );
36.
37.     A2 = (double *)mkl_malloc (m*k*sizeof( double ), 64 );
38.     B2 = (double *)mkl_malloc (k*n*sizeof( double ), 64 );
39.     C2 = (double *)mkl_malloc (m*n*sizeof( double ), 64 );
40.     C2_fl = (double *)mkl_malloc (m*n*sizeof( double ), 64 );
41.
42.     if (A1 == NULL || B1 == NULL || C1 == NULL || C1_fl == NULL ||
43.         A2 == NULL || B2 == NULL || C2 == NULL || C2_fl == NULL) {
44.       printf( "\n ERROR: Can't allocate memory for matrices.Aborting...\n\n");
45.       return 1;
46.     }
47.
48.     printf (" Intializing matrix data \n\n");
49.     for (i = 0; i < (m*k); i++) {
50.         A1[i] = A2[i] = (double)(i+1);
51.     }
52.
53.     for (i = 0; i < (k*n); i++) {
54.         B1[i] = B2[i] = (double)(-i-1);
55.     }
56.
57.     for (i = 0; i < (m*n); i++) {
58.         C1[i]    = C2[i]    = 0.0;
59.         C1_fl[i] = C2_fl[i] = 0.0;
60.     }
61.
62.     printf (" \nComputing matrix product using Intel MKL cblas_dgemm_batch function
    \n");
63.
64.     #define GRP_COUNT 1  // 1 グループ
65.
66.     MKL_INT group_count = GRP_COUNT;
67.     MKL_INT group_sizes[GRP_COUNT] = {2};  // 8 行列乗算
68.
69.     CBLAS_TRANSPOSE transa_array[GRP_COUNT] = {CblasNoTrans};
70.     CBLAS_TRANSPOSE transb_array[GRP_COUNT] = {CblasNoTrans};
71.
72.     MKL_INT m_array[GRP_COUNT] = {m};
73.     MKL_INT n_array[GRP_COUNT] = {n};
74.     MKL_INT k_array[GRP_COUNT] = {k};
75.
76.     MKL_INT lda_array[GRP_COUNT] = {k};
77.     MKL_INT ldb_array[GRP_COUNT] = {n};
78.     MKL_INT ldc_array[GRP_COUNT] = {n};
79.
```

```
80.     double alpha_array[GRP_COUNT] = {alpha};
81.     double beta_array[GRP_COUNT]  = {beta};
82.
83.     // 行列の乗算数 = 2
84.     double **a_array, **b_array, **c_array;
85.     a_array = (double **)mkl_calloc(2, sizeof( double* ), 64);
86.     b_array = (double **)mkl_calloc(2, sizeof( double* ), 64);
87.     c_array = (double **)mkl_calloc(2, sizeof( double* ), 64);
88.
89.     t_start = omp_get_wtime();
90.
91.     // cblas_dgemm_batch の呼び出し
92.     #pragma omp target enter data \
93.       map(to: A1[0:m*k], B1[0:k*n], C1[0:m*n]) \
94.       map(to: A2[0:m*k], B2[0:k*n], C2[0:m*n])
95.
96.     #pragma omp target data use_device_ptr(A1, B1, C1, A2, B2, C2)
97.     {
98.       a_array[0] = A1, a_array[1] = A2;
99.       b_array[0] = B1, b_array[1] = B2;
100.      c_array[0] = C1, c_array[1] = C2;
101.    }
102.
103.    #pragma omp target data                                \
104.      map(to:a_array[0:2], b_array[0:2], c_array[0:2])
105.    {
106.       #pragma omp dispatch
107.         cblas_dgemm_batch (
108.             CblasRowMajor,
109.             transa_array,
110.             transb_array,
111.             m_array,
112.             n_array,
113.             k_array,
114.             alpha_array,
115.             (const double **)a_array,
116.             lda_array,
117.             (const double **)b_array,
118.             ldb_array,
119.             beta_array,
120.             c_array,
121.             ldc_array,
122.             group_count,
123.             group_sizes);
124.    } // end target datamap
125.
126.    #pragma omp target exit data \
127.      map(from: C1[0:m*n], C2[0:m*n])
128.
129.    t_end = omp_get_wtime();
130.
131.    printf ("\n Top left corner of matrix C1: \n");
132.    for (i=0; i<min(m,6); i++) {
133.      for (j=0; j<min(n,6); j++) {
134.        printf ("%12.5G", C1[j+i*n]);
135.      }
136.      printf ("\n");
137.    }
138.
139.    printf ("\n Top left corner of matrix C2: \n");
140.    for (i=0; i<min(m,6); i++) {
141.      for (j=0; j<min(n,6); j++) {
142.        printf ("%12.5G", C2[j+i*n]);
143.      }
144.      printf ("\n");
145.    }
146.
147.    printf (" \nComputing matrix product using for-loops \n");
```

```
148.
149.    for (i = 0; i < m; i++) {
150.        for (j = 0; j < n; j++) {
151.            sum = 0.0;
152.            for (q = 0; q < k; q++)
153.                sum += A1[k*i+q] * B1[n*q+j];
154.            C1_fl[n*i+j] = sum;
155.        }
156.    }
157.
158.    for (i = 0; i < m; i++) {
159.        for (j = 0; j < n; j++) {
160.            sum = 0.0;
161.            for (q = 0; q < k; q++)
162.                sum += A2[k*i+q] * B2[n*q+j];
163.            C2_fl[n*i+j] = sum;
164.        }
165.    }
166.
167.    printf ("\n Top left corner of matrix C1: \n");
168.    for (i=0; i<min(m,6); i++) {
169.      for (j=0; j<min(n,6); j++) {
170.        printf ("%12.5G", C1_fl[j+i*n]);
171.      }
172.      printf ("\n");
173.    }
174.
175.    printf ("\n Top left corner of matrix C2: \n");
176.    for (i=0; i<min(m,6); i++) {
177.      for (j=0; j<min(n,6); j++) {
178.        printf ("%12.5G", C2_fl[j+i*n]);
179.      }
180.      printf ("\n");
181.    }
182.
183.    printf ("\n Computations completed.Verifying...\n\n");
184.
185.    fail = 0;
186.    for (i = 0; i < (m*n); i++) {
187.      if (! compare(C1[i], C1_fl[i]) || ! compare(C2[i], C2_fl[i])) {
188.          fail = 1;
189.          break;
190.      }
191.    }
192.
193.    if (fail) {
194.        printf (" **** FAIL **** \n");
195.    }
196.    else {
197.        printf(" time = %lf seconds\n", t_end - t_start);
198.        printf (" **** PASS **** \n");
199.    }
200.
201.    mkl_free(A1);
202.    mkl_free(B1);
203.    mkl_free(C1);
204.    mkl_free(C1_fl);
205.    mkl_free(A2);
206.    mkl_free(B2);
207.    mkl_free(C2);
208.    mkl_free(C2_fl);
209.
210.    return 0;
211.}
```

ここで使用する GPU (1 スタックのみ) で実行した 2 つの例のパフォーマンスは次のようになります。

```
dgemm_example_01_c.cpp (two calls to cblas_dgemm): 2.976183 seconds
dgemm_batch_example_01_c.cpp (one call to cblas_dgemm_batch): 1.881641 seconds
```

さらに複雑なバッチの例を以下に示します。この例では、3 つのグループで構成されるバッチ (`GROUP_COUNT=3`) があります。各グループのサイズは、1 から 10 の間でランダムに選択されます。いくつかのパラメーター (`layout`、`transA`、`transB`、`m`、`n`、および `k`) がランダムに選択されますが、各グループではすべての乗算で同じパラメーターが適用されます。`total_batch_size` はすべてのグループサイズの合計と等価です。

```
1.  #include <stdio.h>
2.  #include <omp.h>
3.  #include "mkl.h"
4.  #include "mkl_omp_offload.h"
5.  #include "common.h"
6.
7.  #define GROUP_COUNT 3
8.
9.  int dnum = 0;
10.
11. int main() {
12.
13.   CBLAS_LAYOUT layout = (rand_int(0,1) == 0) ? CblasColMajor : CblasRowMajor;
14.   CBLAS_TRANSPOSE *transA, *transB;
15.   MKL_INT *m, *n, *k, *lda, *ldb, *ldc;
16.   double *alpha, *beta;
17.   MKL_INT *group_size, *sizea_array, *sizeb_array, *sizec_array, total_batch_size = 0,
    sizea, sizeb, sizec;
18.   double **a_array, **b_array, **c_array, **c_ref_array;
19.   double **a_array_dev, **b_array_dev, **c_array_dev;
20.
21.   transA = (CBLAS_TRANSPOSE *)mkl_malloc(GROUP_COUNT * sizeof(CBLAS_TRANSPOSE), 64);
22.   transB = (CBLAS_TRANSPOSE *)mkl_malloc(GROUP_COUNT * sizeof(CBLAS_TRANSPOSE), 64);
23.
24.   m = (MKL_INT *)mkl_malloc(GROUP_COUNT * sizeof(MKL_INT), 64);
25.   n = (MKL_INT *)mkl_malloc(GROUP_COUNT * sizeof(MKL_INT), 64);
26.   k = (MKL_INT *)mkl_malloc(GROUP_COUNT * sizeof(MKL_INT), 64);
27.   lda = (MKL_INT *)mkl_malloc(GROUP_COUNT * sizeof(MKL_INT), 64);
28.   ldb = (MKL_INT *)mkl_malloc(GROUP_COUNT * sizeof(MKL_INT), 64);
29.   ldc = (MKL_INT *)mkl_malloc(GROUP_COUNT * sizeof(MKL_INT), 64);
30.   group_size = (MKL_INT *)mkl_malloc(GROUP_COUNT * sizeof(MKL_INT), 64);
31.   alpha = (double *)mkl_malloc(GROUP_COUNT * sizeof(double), 64);
32.   beta = (double *)mkl_malloc(GROUP_COUNT * sizeof(double), 64);
33.
34.   if ((m == NULL) || (n == NULL) || (k == NULL) || (lda == NULL) || (ldb == NULL) ||
35.  (ldc == NULL) || (group_size == NULL) || (alpha == NULL) || (beta == NULL)) {
36.       printf("Cannot allocate input arrays\n");
37.       return 1;
38.   }
39.
40.   MKL_INT i, j, p, idx;
41.
42.   for (i = 0; i < GROUP_COUNT; i++) {
43.       transA[i] = (rand_int(0,1) == 0) ? CblasNoTrans : CblasTrans;
44.       transB[i] = (rand_int(0,1) == 0) ? CblasNoTrans : CblasTrans;
45.       alpha[i] = rand_double_scalar();
46.       beta[i] = rand_double_scalar();
47.       m[i] = rand_int(1, 20);
48.       n[i] = rand_int(1, 20);
49.       k[i] = rand_int(1, 20);
50.       lda[i] = MAX(m[i], k[i]);
51.       ldb[i] = MAX(k[i], n[i]);
52.       ldc[i] = MAX(m[i], n[i]);
53.       group_size[i] = rand_int(1, 10);
54.       total_batch_size += group_size[i];
55. #ifdef MKL_ILP64
```

```
56.       printf("Group %lld: layout = %s, transA = %s, transB = %s, m = %lld, n = %lld,
    k = %lld, lda = %lld, ldb = %lld, ldc = %lld, alpha = %lf, beta = %lf, group_size
    = %lld\n",
57.            i, (layout == CblasColMajor) ? "Column Major" : "Row Major",
58.            (transA[i] == CblasNoTrans) ? "Non Transpose" : "Transpose",
59.            (transB[i] == CblasNoTrans) ? "Non Transpose" : "Transpose",
60.            m[i], n[i], k[i], lda[i], ldb[i], ldc[i], alpha[i], beta[i],
    group_size[i]);
61. #else
62.       printf("Group %d: layout = %s, transA = %s, transB = %s, m = %d, n = %d, k = %d,
    lda = %d, ldb = %d, ldc = %d, alpha = %lf, beta = %lf, group_size = %d\n",
63.            i, (layout == CblasColMajor) ? "Column Major" : "Row Major",
64.            (transA[i] == CblasNoTrans) ? "Non Transpose" : "Transpose",
65.            (transB[i] == CblasNoTrans) ? "Non Transpose" : "Transpose",
66.            m[i], n[i], k[i], lda[i], ldb[i], ldc[i], alpha[i],
    beta[i],group_size[i]);
67. #endif
68.   }
69.
70.   sizea_array = (MKL_INT *)mkl_malloc(sizeof(MKL_INT) * total_batch_size, 64);
71.   sizeb_array = (MKL_INT *)mkl_malloc(sizeof(MKL_INT) * total_batch_size, 64);
72.   sizec_array = (MKL_INT *)mkl_malloc(sizeof(MKL_INT) * total_batch_size, 64);
73.
74.   a_array = (double **)mkl_malloc(sizeof(double *) * total_batch_size, 64);
75.   b_array = (double **)mkl_malloc(sizeof(double *) * total_batch_size, 64);
76.   c_array = (double **)mkl_malloc(sizeof(double *) * total_batch_size, 64);
77.   a_array_dev = (double **)mkl_malloc(sizeof(double *) * total_batch_size, 64);
78.   b_array_dev = (double **)mkl_malloc(sizeof(double *) * total_batch_size, 64);
79.   c_array_dev = (double **)mkl_malloc(sizeof(double *) * total_batch_size, 64);
80.   c_ref_array = (double **)mkl_malloc(sizeof(double *) * total_batch_size, 64);
81.
82.   if ((sizea_array == NULL) || (sizeb_array == NULL) || (sizec_array == NULL) ||
83.  (a_array == NULL) || (b_array == NULL) || (c_array == NULL) ||
84. (a_array_dev == NULL) || (b_array_dev == NULL) ||
85.       (c_array_dev == NULL) || (c_ref_array == NULL)) {
86.       printf("Cannot allocate matrices and size arrays\n");
87.       return 1;
88.   }
89.
90.   idx = 0;
91.   for (i = 0; i < GROUP_COUNT; i++) {
92.       sizea = (((layout == CblasRowMajor) && (transA[i] == CblasTrans)) ||
93.                ((layout == CblasColMajor) && (transA[i] == CblasNoTrans))) ? lda[i] *
    k[i] : m[i] * lda[i];
94.       sizeb = (((layout == CblasRowMajor) && (transB[i] == CblasTrans)) ||
95.                ((layout == CblasColMajor) && (transB[i] == CblasNoTrans))) ? ldb[i] *
    n[i] : k[i] * ldb[i];
96.       sizec = (layout == CblasColMajor) ? ldc[i] * n[i] : ldc[i] * m[i];
97.       for (j = 0; j < group_size[i]; j++) {
98.           a_array[idx] = (double *)mkl_malloc(sizeof(double) * sizea, 64);
99.           a_array_dev[idx] = a_array[idx];
100.          sizea_array[idx] = sizea;
101.          if (a_array[idx] == NULL) {
102.              printf("cannot allocate a matrices\n");
103.              return 1;
104.          }
105.          b_array[idx] = (double *)mkl_malloc(sizeof(double) * sizeb, 64);
106.          b_array_dev[idx] = b_array[idx];
107.          sizeb_array[idx] = sizeb;
108.          if (b_array[idx] == NULL) {
109.              printf("cannot allocate b matrices\n");
110.              return 1;
111.          }
112.          c_array[idx] = (double *)mkl_malloc(sizeof(double) * sizec, 64);
113.          c_array_dev[idx] = c_array[idx];
114.          sizec_array[idx] = sizec;
115.          if (c_array[idx] == NULL) {
116.              printf("cannot allocate c matrices\n");
117.              return 1;
```

```
118.          }
119.          c_ref_array[idx] = (double *)mkl_malloc(sizeof(double) * sizec, 64);
120.          if (c_ref_array[idx] == NULL) {
121.              printf("cannot allocate c_ref matrices\n");
122.              return 1;
123.          }
124.          init_double_array(sizea, a_array[idx], 1);
125.          init_double_array(sizeb, b_array[idx], 1);
126.          init_double_array(sizec, c_array[idx], 1);
127.          for (p = 0; p < sizec_array[idx]; p++) c_ref_array[idx][p] =
    c_array[idx][p];
128.          idx++;
129.      }
130.  }
131.
132.  // ホストで gemm_batch を実行し、標準 oneMKL インターフェイスを使用ます
133.  cblas_dgemm_batch(layout, transA, transB, m, n, k, alpha,
134.(const double **) a_array, lda,(const double **) b_array,
135.ldb, beta, c_ref_array, ldc, GROUP_COUNT, group_size);
136.
137.  double *a, *b, *c;
138.  for (i = 0; i < total_batch_size; i++) {
139.      a = a_array[i];
140.      b = b_array[i];
141.      c = c_array[i];
142.#pragma omp target enter data
    map(to:a[0:sizea_array[i]],b[0:sizeb_array[i]],c[0:sizec_array[i]])
143.#pragma omp target data use_device_ptr(a,b,c)
144.      {
145.          a_array_dev[i] = a;
146.          b_array_dev[i] = b;
147.          c_array_dev[i] = c;
148.      }
149.  }
150.#pragma omp target data map(to:a_array_dev[0:total_batch_size], \
151.                          b_array_dev[0:total_batch_size], \
152.                          c_array_dev[0:total_batch_size]) device(dnum)
153.  {
154.
155.#pragma omp dispatch
156.      cblas_dgemm_batch(layout, transA, transB, m, n, k, alpha,
157.(const double **) a_array_dev, lda, (const double **) b_array_dev,
158.ldb, beta, c_array_dev, ldc, GROUP_COUNT, group_size);
159.
160.  }
161.
162.  for (i = 0; i < total_batch_size; i++) {
163.      a = a_array[i];
164.      b = b_array[i];
165.      c = c_array[i];
166.#pragma omp target exit data
    map(from:a[0:sizea_array[i]],b[0:sizeb_array[i]],c[0:sizec_array[i]])
167.  }
168.
169.  double computed, reference, diff;
170.  MKL_INT l;
171.  idx = 0;
172.  for (p = 0; p < GROUP_COUNT; p++) {
173.      for (l = 0; l < group_size[p]; l++) {
174.          for (i = 0; i < m[p]; i++) {
175.              for (j = 0; j < n[p]; j++) {
176.                  if (layout == CblasColMajor) {
177.                      computed = c_array[idx][i + j * ldc[p]];
178.                      reference = c_ref_array[idx][i + j * ldc[p]];
179.                  }
180.                  else {
181.                      computed = c_array[idx][j + i * ldc[p]];
182.                      reference = c_ref_array[idx][j + i * ldc[p]];
183.                  }
```

```
                    diff = computed - reference;
                    diff = (diff > 0) ? diff : -diff;
                    if (diff > 0.0001) {
#ifdef MKL_ILP64
                        printf("Error in matrix %lld (group = %lld, matrix index in group = %lld) at index [%lld][%lld], computed = %lf, reference = %lf, difference = %lf\n", idx, p, l, i, j, computed, reference, diff);
#else
                        printf("Error in matrix %d at index [%d][%d], computed = %lf, reference = %lf, difference = %lf\n", idx, i, j, computed, reference, diff);
#endif

                        free_double_matrices(a_array, total_batch_size);
                        free_double_matrices(b_array, total_batch_size);
                        free_double_matrices(c_array, total_batch_size);
                        free_double_matrices(c_ref_array, total_batch_size);
                        mkl_free(a_array);
                        mkl_free(b_array);
                        mkl_free(c_array);
                        mkl_free(c_ref_array);
                        mkl_free(a_array_dev);
                        mkl_free(b_array_dev);
                        mkl_free(c_array_dev);
                        mkl_free(sizea_array);
                        mkl_free(sizeb_array);
                        mkl_free(sizec_array);
                        mkl_free(transA); mkl_free(transB);
                        mkl_free(m); mkl_free(n); mkl_free(k);
                        mkl_free(lda); mkl_free(ldb); mkl_free(ldc); mkl_free(group_size);
                        mkl_free(alpha); mkl_free(beta);
                        return 1;
                    }
                }
            }
            idx++;
        }
    }

    printf("Validation PASSED\n");
    free_double_matrices(a_array, total_batch_size);
    free_double_matrices(b_array, total_batch_size);
    free_double_matrices(c_array, total_batch_size);
    free_double_matrices(c_ref_array, total_batch_size);
    mkl_free(a_array);
    mkl_free(b_array);
    mkl_free(c_array);
    mkl_free(c_ref_array);
    mkl_free(a_array_dev);
    mkl_free(b_array_dev);
    mkl_free(c_array_dev);
    mkl_free(sizea_array);
    mkl_free(sizeb_array);
    mkl_free(sizec_array);
    mkl_free(transA); mkl_free(transB);
    mkl_free(m); mkl_free(n); mkl_free(k);
    mkl_free(lda); mkl_free(ldb); mkl_free(ldc); mkl_free(group_size);
    mkl_free(alpha); mkl_free(beta);
    return 0;
}
```

## 独立した連続する GEMM 呼び出しの高速化

独立して実行できる連続した GEMM 呼び出しの実行を高速化する方法はいろいろあります。1 つは、上記のようにバッチ版の GEMM を呼び出して GEMM 呼び出しをバッチ処理する方法です。

もう 1 つは、GEMM 呼び出しを OpenMP* `parallel` 構造で囲み、並列領域を実行する各 OpenMP* スレッドが GEMM 呼び出しを 1 つずつディスパッチする方法です。この並列化の方法を次の例で説明します。

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <math.h>
4.  #include <omp.h>
5.  #include "mkl.h"
6.  #include "mkl_omp_offload.h"
7.
8.  #define min(x,y) (((x) < (y)) ? (x) : (y))
9.  #define epsilon 0.0000001f
10.
11. bool compare(double x, double y)
12. {
13.     // x と y が等しければ true を返す
14.     return fabs(x - y) <= epsilon;
15. }
16.
17. int main()
18. {
19.     double *A1, *B1, *C1, *C1_fl;
20.     double *A2, *B2, *C2, *C2_fl;
21.     int m, n, k, i, j, q;
22.     double alpha, beta;
23.     double sum;
24.     int fail;
25.     double t_start, t_end;
26.
27.     m = 2000, k = 200, n = 1000;
28.     alpha = 1.0; beta = 0.0;
29.
30.     printf (" Allocating memory for matrices aligned on 64-byte boundary for better
    \n"
31.             " performance \n\n");
32.     A1 = (double *)mkl_malloc (m*k*sizeof( double ), 64 );
33.     B1 = (double *)mkl_malloc (k*n*sizeof( double ), 64 );
34.     C1 = (double *)mkl_malloc (m*n*sizeof( double ), 64 );
35.     C1_fl = (double *)mkl_malloc (m*n*sizeof( double ), 64 );
36.
37.     A2 = (double *)mkl_malloc (m*k*sizeof( double ), 64 );
38.     B2 = (double *)mkl_malloc (k*n*sizeof( double ), 64 );
39.     C2 = (double *)mkl_malloc (m*n*sizeof( double ), 64 );
40.     C2_fl = (double *)mkl_malloc (m*n*sizeof( double ), 64 );
41.
42.     if (A1 == NULL || B1 == NULL || C1 == NULL || C1_fl == NULL ||
43.         A2 == NULL || B2 == NULL || C2 == NULL || C2_fl == NULL) {
44.       printf( "\n ERROR: Can't allocate memory for matrices.Aborting...\n\n");
45.       return 1;
46.     }
47.
48.     printf (" Intializing matrix data \n\n");
49.     for (i = 0; i < (m*k); i++) {
50.         A1[i] = A2[i] = (double)(i+1);
51.     }
52.
53.     for (i = 0; i < (k*n); i++) {
54.         B1[i] = B2[i] = (double)(-i-1);
55.     }
56.
57.     for (i = 0; i < (m*n); i++) {
58.         C1[i]    = C2[i]    = 0.0;
59.         C1_fl[i] = C2_fl[i] = 0.0;
60.     }
61.
62.     printf (" \nComputing matrix product using Intel MKL cblas_dgemm function \n");
63.
```

```
64.     t_start = omp_get_wtime();
65.
66.     #pragma omp target data                                  \
67.       map(to: A1[0:m*k], B1[0:k*n], A2[0:m*k], B2[0:k*n])  \
68.       map(tofrom: C1[0:m*n], C2[0:m*n])
69.     {
70.        #pragma omp parallel num_threads(2)
71.        {
72.           int id = omp_get_thread_num();
73.
74.           if (id == 0) {
75.              #pragma omp dispatch
76.              cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
77.                          m, n, k, alpha, A1, k, B1, n, beta, C1, n);
78.         }
79.         else if (id == 1) {
80.              #pragma omp dispatch
81.              cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
82.                          m, n, k, alpha, A2, k, B2, n, beta, C2, n);
83.           }
84.        }
85.     }
86.
87.     t_end = omp_get_wtime();
88.
89.     printf ("\n Top left corner of matrix C1: \n");
90.     for (i=0; i<min(m,6); i++) {
91.       for (j=0; j<min(n,6); j++) {
92.         printf ("%12.5G", C1[j+i*n]);
93.       }
94.       printf ("\n");
95.     }
96.
97.     printf ("\n Top left corner of matrix C2: \n");
98.     for (i=0; i<min(m,6); i++) {
99.       for (j=0; j<min(n,6); j++) {
100.        printf ("%12.5G", C2[j+i*n]);
101.      }
102.      printf ("\n");
103.    }
104.
105.    printf (" \nComputing matrix product using for-loops \n");
106.
107.    for (i = 0; i < m; i++) {
108.        for (j = 0; j < n; j++) {
109.            sum = 0.0;
110.            for (q = 0; q < k; q++)
111.                sum += A1[k*i+q] * B1[n*q+j];
112.            C1_fl[n*i+j] = sum;
113.        }
114.    }
115.
116.    for (i = 0; i < m; i++) {
117.        for (j = 0; j < n; j++) {
118.            sum = 0.0;
119.            for (q = 0; q < k; q++)
120.                sum += A2[k*i+q] * B2[n*q+j];
121.            C2_fl[n*i+j] = sum;
122.        }
123.    }
124.
125.    printf ("\n Top left corner of matrix C1: \n");
126.    for (i=0; i<min(m,6); i++) {
127.      for (j=0; j<min(n,6); j++) {
128.        printf ("%12.5G", C1_fl[j+i*n]);
129.      }
130.      printf ("\n");
131.    }
132.
```

```
133.    printf ("\n Top left corner of matrix C2: \n");
134.    for (i=0; i<min(m,6); i++) {
135.      for (j=0; j<min(n,6); j++) {
136.        printf ("%12.5G", C2_fl[j+i*n]);
137.      }
138.      printf ("\n");
139.    }
140.
141.    printf ("\n Computations completed.Verifying...\n\n");
142.
143.    fail = 0;
144.    for (i = 0; i < (m*n); i++) {
145.      if (! compare(C1[i], C1_fl[i]) || ! compare(C2[i], C2_fl[i])) {
146.          fail = 1;
147.          break;
148.      }
149.    }
150.
151.    if (fail) {
152.        printf (" **** FAIL **** \n");
153.    }
154.    else {
155.        printf(" time = %lf seconds\n", t_end - t_start);
156.        printf (" **** PASS **** \n");
157.    }
158.
159.    mkl_free(A1);
160.    mkl_free(B1);
161.    mkl_free(C1);
162.    mkl_free(C1_fl);
163.    mkl_free(A2);
164.    mkl_free(B2);
165.    mkl_free(C2);
166.    mkl_free(C2_fl);
167.
168.    return 0;
169.}
```

独立して連続した GEMM 呼び出しの実行を高速化するもう 1 つの方法は、ホストスレッドが次の GEMM 呼び出しをディスパッチする前に、ディスパッチされた GEMM 呼び出しの完了を待機しないよう、`dispatch` 構造で `nowait` 節を使用する方法です。最後の GEMM 呼び出しの後に OpenMP* `taskwait` ディレクティブを挿入して、ホストスレッドが処理を続ける前に、ディスパッチされたすべての oneMKL 呼び出しが完了することを保証します。この `nowait` を使用したアプローチを次に示します。

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <math.h>
4.  #include <omp.h>
5.  #include "mkl.h"
6.  #include "mkl_omp_offload.h"
7.
8.  #define min(x,y) (((x) < (y)) ? (x) : (y))
9.  #define epsilon 0.0000001f
10.
11. bool compare(double x, double y)
12. {
13.     // x と y が等しければ true を返す
14.     return fabs(x - y) <= epsilon;
15. }
16.
17. int main()
18. {
19.     double *A1, *B1, *C1, *C1_fl;
20.     double *A2, *B2, *C2, *C2_fl;
21.     int m, n, k, i, j, q;
22.     double alpha, beta;
```

```
23.     double sum;
24.     int fail;
25.     double t_start, t_end;
26.
27.     m = 2000, k = 200, n = 1000;
28.     alpha = 1.0; beta = 0.0;
29.
30.     printf (" Allocating memory for matrices aligned on 64-byte boundary for better
    \n"
31.             " performance \n\n");
32.     A1 = (double *)mkl_malloc (m*k*sizeof( double ), 64 );
33.     B1 = (double *)mkl_malloc (k*n*sizeof( double ), 64 );
34.     C1 = (double *)mkl_malloc (m*n*sizeof( double ), 64 );
35.     C1_fl = (double *)mkl_malloc (m*n*sizeof( double ), 64 );
36.
37.     A2 = (double *)mkl_malloc (m*k*sizeof( double ), 64 );
38.     B2 = (double *)mkl_malloc (k*n*sizeof( double ), 64 );
39.     C2 = (double *)mkl_malloc (m*n*sizeof( double ), 64 );
40.     C2_fl = (double *)mkl_malloc (m*n*sizeof( double ), 64 );
41.
42.     if (A1 == NULL || B1 == NULL || C1 == NULL || C1_fl == NULL ||
43.         A2 == NULL || B2 == NULL || C2 == NULL || C2_fl == NULL) {
44.       printf( "\n ERROR: Can't allocate memory for matrices.Aborting...\n\n");
45.       return 1;
46.     }
47.
48.     printf (" Intializing matrix data \n\n");
49.     for (i = 0; i < (m*k); i++) {
50.         A1[i] = A2[i] = (double)(i+1);
51.     }
52.
53.     for (i = 0; i < (k*n); i++) {
54.         B1[i] = B2[i] = (double)(-i-1);
55.     }
56.
57.     for (i = 0; i < (m*n); i++) {
58.         C1[i]    = C2[i]    = 0.0;
59.         C1_fl[i] = C2_fl[i] = 0.0;
60.     }
61.
62.     printf (" \nComputing matrix product using Intel MKL cblas_dgemm function \n");
63.
64.     t_start = omp_get_wtime();
65.
66.     #pragma omp target data                                  \
67.       map(to: A1[0:m*k], B1[0:k*n], A2[0:m*k], B2[0:k*n])  \
68.       map(tofrom: C1[0:m*n], C2[0:m*n])
69.     {
70.        #pragma omp dispatch nowait
71.        cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
72.                    m, n, k, alpha, A1, k, B1, n, beta, C1, n);
73.
74.        #pragma omp dispatch nowait
75.        cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
76.                    m, n, k, alpha, A2, k, B2, n, beta, C2, n);
77.
78.        #pragma omp taskwait
79.     }
80.
81.     t_end = omp_get_wtime();
82.
83.     printf ("\n Top left corner of matrix C1: \n");
84.     for (i=0; i<min(m,6); i++) {
85.       for (j=0; j<min(n,6); j++) {
86.         printf ("%12.5G", C1[j+i*n]);
87.       }
88.       printf ("\n");
89.     }
90.
```

```
91.     printf ("\n Top left corner of matrix C2: \n");
92.     for (i=0; i<min(m,6); i++) {
93.       for (j=0; j<min(n,6); j++) {
94.         printf ("%12.5G", C2[j+i*n]);
95.       }
96.       printf ("\n");
97.     }
98.
99.     printf (" \nComputing matrix product using for-loops \n");
100.
101.    for (i = 0; i < m; i++) {
102.        for (j = 0; j < n; j++) {
103.            sum = 0.0;
104.            for (q = 0; q < k; q++)
105.                sum += A1[k*i+q] * B1[n*q+j];
106.            C1_fl[n*i+j] = sum;
107.        }
108.    }
109.
110.    for (i = 0; i < m; i++) {
111.        for (j = 0; j < n; j++) {
112.            sum = 0.0;
113.            for (q = 0; q < k; q++)
114.                sum += A2[k*i+q] * B2[n*q+j];
115.            C2_fl[n*i+j] = sum;
116.        }
117.    }
118.
119.    printf ("\n Top left corner of matrix C1: \n");
120.    for (i=0; i<min(m,6); i++) {
121.      for (j=0; j<min(n,6); j++) {
122.        printf ("%12.5G", C1_fl[j+i*n]);
123.      }
124.      printf ("\n");
125.    }
126.
127.    printf ("\n Top left corner of matrix C2: \n");
128.    for (i=0; i<min(m,6); i++) {
129.      for (j=0; j<min(n,6); j++) {
130.        printf ("%12.5G", C2_fl[j+i*n]);
131.      }
132.      printf ("\n");
133.    }
134.
135.    printf ("\n Computations completed.Verifying...\n\n");
136.
137.    fail = 0;
138.    for (i = 0; i < (m*n); i++) {
139.      if (! compare(C1[i], C1_fl[i]) || ! compare(C2[i], C2_fl[i])) {
140.          fail = 1;
141.          break;
142.      }
143.    }
144.
145.    if (fail) {
146.        printf (" **** FAIL **** \n");
147.    }
148.    else {
149.        printf(" time = %lf seconds\n", t_end - t_start);
150.        printf (" **** PASS **** \n");
151.    }
152.
153.    mkl_free(A1);
154.    mkl_free(B1);
155.    mkl_free(C1);
156.    mkl_free(C1_fl);
157.    mkl_free(A2);
158.    mkl_free(B2);
159.    mkl_free(C2);
```

```
    mkl_free(C2_fl);

    return 0;
}
```

## GEMM 計算で使用される行列のパディング

GEMM 呼び出しは、行列 a、b、c の先頭の次元 lda、ldb、ldc をそれぞれパディングすることで高速化できます。

行列の先頭の次元は、メモリー内の行列レイアウトによって異なります。

- 行優先レイアウト (C/C++): 行列の主次元は、行列の列です。
- 列優先レイアウト (Fortran): 行列の主次元は、行列の行です。

DGEMM 呼び出しに渡す行列サイズには次の 2 つの留意事項があります。

**規則 1:** 最高のパフォーマンスを得るには、GEMM 呼び出しに渡す行列の最初の次元を 64 バイトの倍数 (フル・キャッシュライン・サイズ) にする必要があります。単精度データでは、先頭の次元が (64/sizeof(float)) の倍数の 16 であることを意味し、倍精度データでは、先頭の次元が (64/sizeof(double)) の倍数、つまり 8 であることを意味します。

可能であれば、すべての行列 (a、b、c) をパディングします。ただし、行列 c のパディングは、a と b ほど重要でありません。

**規則 2:** 最高のパフォーマンスを得るには、先頭の次元を 2 の乗数の倍数にしてはなりません (4096 など)。先頭の次元を上記の条件より少し大きくすると (例: 4096 から 4096 + 64 バイトなど)、パフォーマンスが向上する場合があります。

## パディングの例 (Fortran)

次の Fortran の例は、DGEMM 呼び出しに渡す行列をパディングして、パフォーマンスが向上する方法を示しています。

```
 include "mkl_omp_offload.f90"

       ! このサブルーチンはコマンドライン引数 m1、k1、n1 を読み取ります
        subroutine get_arguments (m1, k1, n1)
          implicit none
          integer           :: m1, k1, n1
          character(len=32) :: m1_char, k1_char, n1_char

       ! コマンドライン引数の数が正しいことを確認
       !
          if (command_argument_count() .ne. 3) then
            print *, "ERROR: Three command-line arguments expected; stopping."
            stop
          endif

       ! コマンドライン引数を取得
          call get_command_argument(1, m1_char)
          call get_command_argument(2, k1_char)
          call get_command_argument(3, n1_char)

       ! 引数を整数に変換
          read (m1_char,*) m1
          read (k1_char,*) k1
          read (n1_char,*) n1
```

```
25.       end subroutine get_arguments
26.
27.
28.      ! この関数は、>= n である 8 の倍数の最小値を返します
29.      ! 例:
30.      ! if n =  3, then get_mul8 =  8
31.      ! if n =  9, then get_mul8 = 16
32.      ! if n = 30, then get_mul8 = 32
33.      ! if n = 80, then get_mul8 = 8
34.       integer function get_mul8 (n)
35.         implicit none
36.         integer :: n
37.         integer :: mod
38.         if (mod(n,8) .eq. 0) then
39.             get_mul8 = n
40.         else
41.             get_mul8 = ((n/8) + 1) * 8
42.         endif
43.       end function get_mul8
44.
45.
46.      ! このサブルーチンで行列を初期化
47.       subroutine init_matrix (m, k, n, a, b, c)
48.         implicit none
49.         integer          :: m, k, n
50.         double precision :: a(m,k), b(k,n), c(m,n)
51.         integer          :: i, j
52.
53.         do i = 1, m
54.           do j = 1, k
55.             a(i,j) = (i-1) - (0.25 * k)
56.           end do
57.         end do
58.
59.         do i = 1, k
60.           do j = 1, n
61.             b(i,j) = -((i-1) + j)
62.           end do
63.         end do
64.
65.         do i = 1, m
66.           do j = 1, n
67.             c(i,j) = 0.2 + i - j
68.           end do
69.         end do
70.       end subroutine init_matrix
71.
72.
73.       program DGEMM_MAIN
74.
75. #if defined(MKL_ILP64)
76.         use onemkl_blas_omp_offload_ilp64
77. #else
78.         use onemkl_blas_omp_offload_lp64
79. #endif
80.         use omp_lib
81.         use iso_fortran_env
82.         implicit none
83.
84.         interface
85.           integer function get_mul8 (n)
86.             implicit none
87.             integer :: n
88.           end function get_mul8
89.         end interface
90.
91.         double precision :: alpha, beta
92.         integer :: m1, k1, n1, m2, k2, n2
```

```
93.          double precision, allocatable :: a1(:,:)
94.          double precision, allocatable :: b1(:,:)
95.          double precision, allocatable :: c1(:,:)
96.
97.          double precision, allocatable :: a2(:,:)
98.          double precision, allocatable :: b2(:,:)
99.          double precision, allocatable :: c2(:,:)
100.
101.         double precision :: start_t1, end_t1
102.         double precision :: start_t2, end_t2
103.
104.      ! コマンドライン引数 m1、k1、n1 の読み取り
105.
106.        call get_arguments (m1, k1, n1)
107.
108.      !
109.      ! alpha、beta、m2、k2、n2 を初期化
110.      !
111.
112.       alpha = 1.025
113.       beta  = 0.75
114.
115.       m2 = get_mul8(m1)
116.       k2 = get_mul8(k1)
117.       n2 = get_mul8(n1)
118.
119.      !
120.      ! 行列の割り当てと初期化。
121.      !
122.       allocate( a1(1:m1,1:k1) )
123.       allocate( b1(1:k1,1:n1) )
124.       allocate( c1(1:m1,1:n1) )
125.       allocate( a2(1:m2,1:k2) )
126.       allocate( b2(1:k2,1:n2) )
127.       allocate( c2(1:m2,1:n1) )
128.       call init_matrix (m1, k1, n1, a1, b1, c1)
129.       call init_matrix (m2, k2, n2, a2, b2, c2)
130.
131.
132.      !$omp target data map(to: a1, b1, a2, b2) map(tofrom: c1, c2)
133.
134.      ! デバイスをウォームアップ実行
135.         !$omp dispatch
136.         call DGEMM('N','N',m1,n1,k1,alpha,a1,m1,b1,k1,beta,c1,m1)
137.
138.      !
139.      ! デバイスで DGEMM を実行 (使用するパディングメトリックは a1、b1、c1)
140.      !
141.         start_t1 = omp_get_wtime()
142.
143.         !$omp dispatch
144.         call DGEMM('N','N',m1,n1,k1,alpha,a1,m1,b1,k1,beta,c1,m1)
145.
146.         end_t1 = omp_get_wtime()
147.
148.
149.      ! デバイスをウォームアップ実行
150.         !$omp dispatch
151.         call DGEMM('N','N',m2,n2,k2,alpha,a2,m2,b2,k2,beta,c2,m2)
152.
153.      !
154.      ! デバイスで DGEMM を実行 (使用するパディングメトリックは a2、b2、c2)
155.      !
156.         start_t2 = omp_get_wtime()
157.
158.         !$omp dispatch
```

```
159.         call DGEMM('N','N',m2,n2,k2,alpha,a2,m2,b2,k2,beta,c2,m2)
160.
161.         end_t2 = omp_get_wtime()
162.
163.         !$omp end target data
164.
165.         print 100, alpha, beta
166.         print *
167.         print 101, m1, n1, k1
168.         print 111, (end_t1 - start_t1)
169.         print *
170.         print 102, m2, n2, k2
171.         print 112, (end_t2 - start_t2)
172.
173. 100     format(7x, "ALPHA =", f10.4, "  BETA =",f10.4)
174. 101     format(7x, "M1 =", i5,"  N1 =", i5, "  K1 =",i5)
175. 111     format(7x, "Time (non-padded arrays)  =", f10.4, " sec")
176. 102     format(7x, "M2 =", i5,"  N2 =", i5, "  K2 =",i5)
177. 112     format(7x, "Time (padded arrays) =", f10.4, " sec")
178.
179.         end
```

上記の例では、配列境界 (`m1`、`k1`、`n1`) がコマンドライン引数として渡されています。行列 `a1(1:m1, 1:k1)`、`b1(1:k1, 1:n1)`、`c1(1:m1, 1:n1)` が割り当てられて初期化されます。

また、パディングされた行列 `a2(1:m2, 1:k2)`、`b2(1:k2, 1:n2)`、`c1(1:m2, 1:n2)` が割り当てられて初期化されます。`m2` は `>= m1` である 8 の最小倍数です。同様に、`k2` は `<= k1` である 8 の最小倍数で、`n2` は `>= n1` である 8 の最小倍数です。

プログラムは、`a1`、`b1`、`c1` の DGEMM 計算に要した時間と、`a2`、`b2`、`c2` の DGEMM 計算に要した時間を比較します。

このプログラムのコンパイルとリンク、および実行コマンドを以下に示します。

```
コンパイル:
$ ifx -fiopenmp -fopenmp-targets=spir64 -qmkl -fpp -free -c dgemm_pad_f_01.f

リンク:
$ ifx -fiopenmp -fopenmp-targets=spir64 -qmkl -lOpenCL dgemm_pad_f_01.o

実行:
$ expoet ZE_AFFINITY_MASK=0.0
$ export LIBOMPTARGET_DEBUG=0
$ ./a.out 12001 12001 12001
```

特定の GPU (1 スタックのみ) で実行した出力を示します。

```
ALPHA =   1.0250 BETA =   0.7500
M1 =12001 N1 =12001 K1 =12001
Time (non-padded arrays) =   0.2388 sec
M2 =12008 N2 =12008 K2 =12008 Time (padded arrays) =   0.1648 sec
```

上記では、配列 `a`、`b`、`c` をパディングすることで、DGEMM 呼び出しにかかる時間を、0.2388 秒から 0.1648 秒に短縮できました。

## パディングの例 (C/C++)

次の C/C++ の例は、DGEMM 呼び出しに渡す行列をパディングして、パフォーマンスが向上する方法を示しています。

```
#include "mkl.h"
#include "mkl_omp_offload.h"
#include <float.h>
#include <math.h>
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

#if defined(PRECISION)
#if PRECISION == 1
#define FLOAT double
#else
#define FLOAT float
#endif
#else
#define PRECISION 1
#define FLOAT double
#endif

#define index(i, j, ld) (((j) * (ld)) + (i))

#define RAND() ((FLOAT)rand() / (FLOAT)RAND_MAX * 2.0 - 1.0)

#define MALLOC(x) mkl_malloc((x), 64);
#define FREE mkl_free

#define MALLOC_CHECK(p)
  if (p == NULL) {                                                              \
    fprintf(stderr, "%s:%d: memory allocation error\n", __FILE__, __LINE__);
    return EXIT_FAILURE;
  }

#ifndef max
#define max(a, b) (((a) > (b)) ? (a) : (b))
#endif
#ifndef min
#define min(a, b) (((a) < (b)) ? (a) : (b))
#endif

void printMat(FLOAT *P, int m, int n, int ld) {
  int i, j;
  for (i = 0; i < m; i++) {
    printf("\n");
    for (j = 0; j < n; j++)
      printf("%f ", P[index(i, j, ld)]);
  }
  printf("\n");
}

void gemm_naive(int m, int n, int k, FLOAT alpha, const FLOAT *A, int lda,
                const FLOAT *B, int ldb, FLOAT beta, FLOAT *C, int ldc) {
  int i, j, kk;
  FLOAT temp;
  for (j = 0; j < n; j++) {
    for (i = 0; i < m; i++) {
      temp = 0.0;
      for (kk = 0; kk < k; kk++) {
        temp += alpha * A[index(i, kk, lda)] * B[index(kk, j, ldb)];
      }
      C[index(i, j, ldc)] = temp + beta * C[index(i, j, ldc)];
    }
  }
}

FLOAT infinity_norm_error(int m, int n, int ld, const FLOAT *ans,
                          const FLOAT *ref) {
  int i, j, ind;
  FLOAT norm = 0.0, temp;
```

```
  for (i = 0; i < m; i++) {
    temp = 0.0;
    for (j = 0; j < n; j++) {
      ind = index(i, j, ld);
      temp += fabs(ref[ind] - ans[ind]);
    }
    norm = max(norm, temp);
  }
  return norm;
}

double mysecond() {
  struct timeval tp;
  struct timezone tzp;
  int i;
  i = gettimeofday(&tp, &tzp);
  if (i != 0) {
    fprintf(stderr, "%s:%d: timing error %d\n", __FILE__, __LINE__, i);
    return EXIT_FAILURE;
  }
  return ((double)tp.tv_sec + (double)tp.tv_usec * 1.e-6);
}

#if defined(USE_MKL)
int dnum = 0;
#endif

#if PRECISION == 1
#define LD_ALIGN 256
#define LD_BIAS 8
#else
#define LD_ALIGN 512
#define LD_BIAS 16
#endif

#define HPL_PTR(ptr_, al_) ((((size_t)(ptr_) + (al_)-1) / (al_)) * (al_))

#if defined(PAD_LD)
static inline int getld(int x) {
  int ld;
  ld = HPL_PTR(x, LD_ALIGN); // Rule 1
  if (ld - LD_BIAS >= x)
    ld -= LD_BIAS;
  else
    ld += LD_BIAS; // Rule 2
  return ld;
}
#else
static inline int getld(int x) { return x; }
#endif

int main(int argc, char **argv) {
  int i, j;

  if ((argc < 4) || (argc > 4 && argc < 8)) {
    printf("Performs a DGEMM test C = alpha*A*B + beta*C\n");
    printf("A matrix is MxK and B matrix is KxN\n");
    printf("All matrices are stored in column-major format\n");
    printf("Run as ./dgemm_cublas <M> <K> <N> [<alpha> <beta> <iterations> "
           "<verify>]\n");
    printf("Required inputs are:\n");
    printf("      M: number of rows of matrix A\n");
    printf("      K: number of cols of matrix A\n");
    printf("      N: number of cols of matrix B\n");
    printf("Optional inputs are (all must be provided if providing any):\n");
    printf("      alpha: scalar multiplier (default: 1.0)\n");
    printf("      beta:  scalar multiplier (default: 0.0)\n");
    printf("      iterations: number of blocking DGEMM calls to perform "
           "(default: 10)\n");
```

```
139.    printf("      verify: set to 1 to check solution against CPU reference, "
140.           "not recommended for large M|K|N (default: 0)\n");
141.    return EXIT_FAILURE;
142.  }
143.
144.  FLOAT alpha, beta;
145.  int niter, verify;
146.  int HA = atoi(argv[1]);
147.  int WA = atoi(argv[2]);
148.  int WB = atoi(argv[3]);
149.
150.  if ((HA == 0) || (WA == 0) || (WB == 0))
151.    exit(1);
152.
153.  if (argc > 4) {
154.
155.#if PRECISION == 1
156.    sscanf(argv[4], "%lf", &alpha);
157.    sscanf(argv[5], "%lf", &beta);
158.#else
159.    sscanf(argv[4], "%f", &alpha);
160.    sscanf(argv[5], "%f", &beta);
161.#endif
162.    niter = atoi(argv[6]);
163.    verify = atoi(argv[7]);
164.  } else {
165.    alpha = 1.0;
166.    beta = 0.0;
167.    niter = 10;
168.    verify = 0;
169.  }
170.
171.#if PRECISION == 1
172.  printf("DGEMM performance test\n");
173.#else
174.  printf("SGEMM performance test\n");
175.#endif
176.
177.  int HB = WA;
178.  int WC = WB;
179.  int HC = HA;
180.
181.  int ldA = getld(HA);
182.  int ldB = getld(HB);
183.  int ldC = getld(HC);
184.
185.  printf("M = %d, K = %d, N = %d, ldA = %d, ldB = %d, ldC = %d, alpha = %f, "
186.         "beta = %f, iterations = %d, verify? = %d\n",
187.         HA, WA, WB, ldA, ldB, ldC, alpha, beta, niter, verify);
188.
189.  double start_t, end_t, tot_t = 0.0, best_t = DBL_MAX;
190.
191.  /* ホスト行列を割り当て */
192.  FLOAT *A = (FLOAT *)MALLOC(ldA * WA * sizeof(FLOAT));
193.  MALLOC_CHECK(A);
194.  FLOAT *B = (FLOAT *)MALLOC(ldB * WB * sizeof(FLOAT));
195.  MALLOC_CHECK(B);
196.  FLOAT *C = (FLOAT *)MALLOC(ldC * WC * sizeof(FLOAT));
197.  MALLOC_CHECK(C);
198.  FLOAT *Cref = NULL;
199.  if (verify) {
200.    Cref = (FLOAT *)MALLOC(ldC * WC * sizeof(FLOAT));
201.    MALLOC_CHECK(Cref);
202.  }
203.
204.  printf("\n---------------------\n");
205.  printf("Array A: %d x %d\n", ldA, WA);
206.  printf("Array B: %d x %d\n", ldB, WB);
207.  printf("Array C: %d x %d\n", ldC, WC);
```

```
  printf("---------------------\n");

  /*INITIALIZE WITH PSEUDO-RANDOM DATA*/
  srand(2864);
  for (j = 0; j < WA; j++)
    for (i = 0; i < HA; i++)
      A[index(i, j, ldA)] = RAND();

  for (j = 0; j < WB; j++)
    for (i = 0; i < HB; i++)
      B[index(i, j, ldB)] = RAND();

  if (beta != 0.0) {
    for (j = 0; j < WC; j++)
      for (i = 0; i < HC; i++)
        C[index(i, j, ldC)] = RAND();
  } else {
    for (j = 0; j < WC; j++)
      for (i = 0; i < HC; i++)
        C[index(i, j, ldC)] = 0.0;
  }

  if (verify) {
    for (j = 0; j < WC; j++)
      for (i = 0; i < HC; i++)
        Cref[index(i, j, ldC)] = C[index(i, j, ldC)];
  }

#if defined(USE_MKL)
  size_t sizea = (size_t)ldA * WA;
  size_t sizeb = (size_t)ldB * WB;
  size_t sizec = (size_t)ldC * WC;

#pragma omp target data map(to: A [0:sizea], B [0:sizeb]) map(tofrom: C [0:sizec])
  {

    // ウォームアップ実行
    #pragma omp dispatch
#if PRECISION == 1
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, HA, WB, WA, alpha, A,
                ldA, B, ldB, beta, C, ldC);
#else
    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, HA, WB, WA, alpha, A,
                ldA, B, ldB, beta, C, ldC);
#endif

    // dispatch 構造を使用して、gpu で gemm を実行します
    for (i = 0; i < niter; i++) {
      start_t = mysecond();

      #pragma omp dispatch
#if PRECISION == 1
      cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, HA, WB, WA, alpha,
                  A, ldA, B, ldB, beta, C, ldC);
#else
      cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, HA, WB, WA, alpha,
                  A, ldA, B, ldB, beta, C, ldC);
#endif

      end_t = mysecond();
      tot_t += end_t - start_t;
      best_t = min(best_t, end_t - start_t);
    } // end for
  }
#endif // end #pragma omp target data

  double tflop_count;
  tflop_count = (double)2.0 * HA * WB * WA;
```

```
276.  if (beta != 0.0)
277.    tflop_count += (double)HA * WB;
278.  tflop_count *= 1.E-12;
279.
280.  printf("Total runtime for %d iterations: %f seconds.\n", niter, tot_t);
281.  printf("Mean TFLOP/s: %f\n", (double)niter * tflop_count / tot_t);
282.  printf("Best TFLOP/s: %f\n", (double)tflop_count / best_t);
283.
284.  if (verify) {
285.    // ホストでリファレンスのソリューションを計算します
286.    // (ウォームアップの時間を考慮して niter に 1 を追加)
287.    for (i = 0; i < (niter + 1); i++) {
288.      gemm_naive(HA, WB, WA, alpha, A, ldA, B, ldB, beta, Cref, ldC);
289.    }
290.
291.    printf("Error Matrix Infinity Norm = %f\n",
292.           infinity_norm_error(HA, WB, ldC, C, Cref));
293.  }
294.
295.  FREE(A);
296.  FREE(B);
297.  FREE(C);
298.  if (verify)
299.    FREE(Cref);
300.
301.  return EXIT_SUCCESS;
302.}
```

プログラムには、行列 a、b、c の次元を指定するコマンドライン引数と、いくつかの追加オプション・パラメーターを受け取ります。PAD_LD マクロが定義されると、上記の Tul1 と規則 2 に従って行列の先頭次元がパディングされます。次に、プログラムは cblas_dgemm または cblas_sgemm を niter 回呼び出して、合計時間を出力します。

**コンパイルと実行 (LD_PAD マクロ定義なし)**

コンパイル:

```
$ icpx -fiopenmp -fopenmp-targets=spir64 -qmkl -DUSE_MKL -DPRECISION=1 -c
dgemm_pad_c_01.cpp

$ icpx -fiopenmp -fopenmp-targets=spir64 -qmkl -lOpenCL dgemm_pad_c_01.o -o
dgemm_nopad.exe
```

実行:

```
$ export ZE_AFFINITY_MASK=0.0
$ export LIBOMPTARGET_DEBUG=0
$ ./dgemm_nopad.exe 4001 4001 4001 1.0 0.0 100 0
```

パフォーマンス (パディングなし) は、特定の GPU (1 スタックのみ) で次のようになりました。

```
DGEMM performance test
M = 15000, K = 15000, N = 15000, ldA = 15000, ldB = 15000, ldC = 15000, alpha =
1.000000, beta = 0.000000, iterations = 100, verify? = 0

----------------------
Array A: 15000 x 15000
Array B: 15000 x 15000
Array C: 15000 x 15000
----------------------
Total runtime for 100 iterations: 138.368065 seconds.
Mean TFLOP/s: 4.878293
Best TFLOP/s: 5.657242
```

### コンパイルと実行 (LD_PAD マクロ定義あり)

```
コンパイル:
$ icpx -fiopenmp -fopenmp-targets=spir64 -qmkl -DUSE_MKL -DPRECISION=1 -DPAD_LD -c
dgemm_pad_c_01.cpp

リンク:
$ icpx -fiopenmp -fopenmp-targets=spir64 -qmkl -lOpenCL dgemm_pad_c_01.o -o
dgemm_pad.exe

実行:
$ export ZE_AFFINITY_MASK=0.0
$ export LIBOMPTARGET_DEBUG=0
$ ./dgemm_pad.exe 4001 4001 4001 1.0 0.0 100 0
```

行列 a、b、c の主要次元を 15000 から 15096 に増やしてパディングしたパフォーマンスは、特定の GPU (1 スタックのみ) で次のようになりました。

```
DGEMM performance test
M = 15000, K = 15000, N = 15000, ldA = 15096, ldB = 15096, ldC = 15096, alpha =
1.000000, beta = 0.000000, iterations = 100, verify? = 0

----------------------
Array A: 15096 x 15000
Array B: 15096 x 15000
Array C: 15096 x 15000
----------------------
Total runtime for 100 iterations: 145.219459 seconds.
Mean TFLOP/s: 4.648137
Best TFLOP/s: 5.525268
```

### 参考資料

1. 『インテル® oneAPI マス・カーネル・ライブラリー・デベロッパー・リファレンス - C』(英語)
2. 『インテル® oneAPI マス・カーネル・ライブラリー・デベロッパー・リファレンス - Fortran』(英語)
3. 『インテル® oneAPI マス・カーネル・ライブラリー・デベロッパー・リファレンス – Fortran』-「新しい行列引数タイプ」(英語)
4. バッチ GEMM 操作の紹介 (英語)
5. インテル® oneAPI マス・カーネル・ライブラリーのリンクライン・アドバイザー (英語)

## OpenMP* アプリケーションのパフォーマンスを解析するツール

OpenMP* プログラムのパフォーマンスを解析し、ボトルネックを特定するのに役立つさまざまなツールやメカニズムが提供されています。

- インテル® VTune™ プロファイラー
  インテル® VTune™ プロファイラーを使用して、OpenMP* アプリケーションのパフォーマンスを解析できます。アプリケーションの中で最も時間を費やす (ホットな) 関数を特定し、アプリケーションが CPU 依存であるか、GPU 依存であるか、またコードを GPU にどれだけ効率良くオフロードできるか、シーケンシャルとスレッド化のパフォーマンスを高めるため最適化すべきコード領域などを特定するのを支援します。インテル® VTune™ プロファイラーの詳細については、『インテル® VTune™ プロファイラーのユーザーガイド』をご覧ください。

- レベルゼロと OpenCL* トレースツール
  `onetrace` は、SYCL* および OpenMP* GPU オフロードに対応したレベルゼロ・バックエンド向けのホストおよびデバイスのトレースツールです。このツールに関する詳細は、このドキュメントにある「GPU 向けのインテル® プロファイル・ツール・インターフェイス」を参照してください。

  `onetrace` を `-h -d` オプションで使用すると、トレースの最後にホスト側とデバイス側のサマリーを、「API Timing Results」と「Device Timing Results」という項目に出力します。

  トレースには、明示的なデータ転送のみが出力されることに注意してください。USM (統合共有メモリー) に割り当てられたデータ転送がトレースに含まれないことがあります。

**注:**

- `onetrace` は、oneMKL カーネルがオフロードされたことを確認するのに便利です。`OMP_TARGET_OFFLOAD=MANDATORY` 環境変数は oneMKL に影響しないため、oneMKL カーネルがオフロードされたことを確認するには利用できません。oneMKL カーネル (および他のカーネル) がオフロードされたことを確認する方法の 1 つは、`onetrace` が生成するトレースの「`Device Timing Results`」にリストされるカーネルを確認することです。

**`SYCL_PI_TRACE=2` 環境変数:** SYCL* のランタイム・プラグイン・インターフェイス (PI) は、SYCL* ランタイムのデバイスに依存しない部分と、デバイス上での実行を制御するデバイス固有のランタイムレイヤー間とのインターフェイス・レイヤーです。`SYCL_PI_TRACE=2` に設定すると、すべての PI 呼び出しと引数および戻り値のトレースを取得できます。詳細については、「DPC++ ランタイム環境変数」を参照してください。

**`LIBOMPTARGET_DEBUG=1` 環境変数:** `LIBOMPTARGET_DEBUG` 環境変数は、`libomptarget.so` からのデバッグ情報の取得を制御します。

デバッグ出力は、このドキュメントの「多くの GPU リソースを使用する」および「データ転送とメモリー割り当てを最小化する」の項で示すように、ループ反復の ND-range のパーティション化、ホストとデバイス間のデータ転送、メモリー使用などに関する有用な情報を提供します。

`LIBOMPTARGET_DEBUG` 環境変数の詳細については、LLVM/OpenMP ランタイム (英語) を参照してください。

**`LIBOMPTARGET_PLUGIN_PROFILE` 環境変数:** `LIBOMPTARGET_PROFILE` 環境変数は、`libomptarget.so` が時間プロファイル出力を生成するのを制御します。詳細については、LLVM/OpenMP ランタイム (英語) を参照してください。

**デバイスのコンパイラーで生成されたアセンブリーのダンプ:** ジャストインタイム (JIT) コンパイルを実行する前に (または事前コンパイル (AOT) ではプログラムを実行する前に)、次の 2 つの環境変数を設定することでコンパイラーが生成するアセンブリーをダンプできます。

```
export IGC_ShaderDumpEnable=1
export IGC_DumpToCustomDir=my_dump_dir
```

`LLVM IR`、アセンブリー、および `GenISA` ファイルは、`my_dump_dir` (または指定する名前) のサブディレクトリーにダンプされます。このサブディレクトリーに、各カーネルの `*.asm` ファイルがあります。ファイル名は、カーネルが起動されるソースの行番号を示します。ファイルのヘッダーには、SIMD 幅やコンパイルオプションなどの情報が含まれています。

また、`my_dump_dir` には `HardwareCaps.txt` というファイルがあり、ベクトルエンジン数、スレッド数、スライス数など GPU に関連する情報が含まれます。

インテル® グラフィックス・コンパイラーの詳細と、コンパイルを制御するフラグ (環境変数) の一覧は、インテル® グラフィックス・コンパイラーの Linux* 向け OpenCL* 設定フラグ (英語) を参照してください。

デバッグとプロファイルの詳細については、このドキュメントの「パフォーマンスのプロファイルと解析」の項を参照してください。

## OpenMP* オフロードの最良の事例

ここでは、GPU にオフロードするアプリケーションのパフォーマンスを向上させる最良の事例を紹介します。事例を以下のカテゴリーに分類して、以降のセクションで説明します。

- 多くの GPU リソースを使用する
- データ転送とメモリー割り当てを最小化する
- OpenMP* 構造を有効活用する
- メモリー割り当て
- Fortran の例
- 節: `is_device_ptr`、`use_device_ptr`、`has_device_addr`、`use_device_addr`
- プリフェッチ

**注:**

OpenMP* のパフォーマンスを収集するため次の構成を使用しました。

- 2 スタックのインテル® GPU
- 単一の GPU タイルのみ (暗黙的または明示的なスケーリングなし)
- インテル® コンパイラー、ランタイム、および GPU ドライバー
- レベルゼロプラグイン
- 起動時のオーバーヘッドを計測しないようにダミーの `target` 構造を挿入
- ジャストインタイム (JIT) コンパイルモードを使用

### 多くの GPU リソースを使用する

並列に実行できる work-item 数を多くして、より多くの GPU リソースを利用する (GPU を処理で一杯にする) ことで、オフロードされたコードのパフォーマンスを向上できます。

**注:**

- ループ反復の ND-Range への分割は、コンパイラーとランタイムのヒューリスティックによって決定されますが、GPU ドライバーとハードウェア構成にも依存します。そのため、時間経過とともに変化する可能性があります。しかし、`LIBOMPTARGET_DEBUG=1` の出力からパーティショニングを割り出す方法は変わりません。

### `collapse` 節

入れ子になったループの並列性を高める方法として、`collapse` 節を使用して入れ子になった 2 階層以上のループを 1 つのループとして融合する方法があります。1 つのループに融合することで、並列に実行できる反復回数が増加し、GPU でより多くの work-item を処理できます。

次の例では、4 つの入れ子になったループを GPU にオフロードしています。`parallel for` ディレクティブは、一番外側のループ (30 行目) を並列実行することを指示しています。ループの反復数は `BLOCKS` であり 8 が設定されています。

```
1.  #include <math.h>
2.  #include <omp.h>
3.  #define P 16
4.  #define BLOCKS 8
5.  #define SIZE (BLOCKS * P * P * P)
6.  #define MAX 100
7.  #define scaled_rand() ((rand() % MAX) / (1.0 * MAX))
8.  #define IDX2(i, j) (i * P + j)
9.  #define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)
10. int main(void) {
11.   double w[SIZE];            /* 出力 */
12.   double u[SIZE], dx[P * P]; /* 入力 */
13.   int b, i, j, k, l;         /* ループカウンター */
14.   double start, end;         /* タイマー */
15.   omp_set_default_device(0);
16.   /* 起動時のオーバーヘッドを計測しないよう、ダミーの target 領域を追加しています */
17.   #pragma omp target
18.   { ; }
19.   /* 乱数値で入力を初期化します */
20.   srand(0);
21.   for (int i = 0; i < SIZE; i++)
22.     u[i] = scaled_rand();
23.   for (int i = 0; i < P * P; i++)
24.     dx[i] = scaled_rand();
25.   /* デバイスへデータをマップ */
26.   #pragma omp target enter data map(to: u[0:SIZE], dx[0:P * P])
27.   start = omp_get_wtime();
28.   /* collapse 節なしでカーネルをオフロードします */
29.   #pragma omp target teams distribute parallel for private(b, i, j, k, l)
30.   for (b = 0; b < BLOCKS; b++) {
31.     for (i = 0; i < P; i++) {
32.       for (j = 0; j < P; j++) {
33.         for (k = 0; k < P; k++) {
34.           double ur = 0.;
35.           double us = 0.;
36.           double ut = 0.;
37.           for (l = 0; l < P; l++) {
38.             ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
39.             us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
40.             ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
41.           }
42.           w[IDX4(b, i, j, k)] = ur * us * ut;
43.         }
44.       }
45.     }
46.   }
47.   end = omp_get_wtime();
48.   #pragma omp target exit data map(from: w[0:SIZE])
49.   /* 結果を出力します */
50.   printf("no-collapse-clause: w[0]=%lf time=%lf\n", w[0], end - start);
51.   return 0;
52. }
```

**コンパイルコマンド**

```
$ icx -fiopenmp -fopenmp-targets=spir64 test_no_collapse.cpp
```

**実行コマンド**

```
$ export OMP_TARGET_OFFLOAD=MANDATORY
$ export ZE_AFFINITY_MASK=0.0
$ export LIBOMPTARGET_DEBUG=1
./a.out
```

libomptarget.so のデバッグ情報 (環境変数 LIBOMPTARGET_DEBUG=1 の時に実行時に出力される) には、ループ反復の ND-range パーティション化と collapse 節を使用した際の並列性の向上が示されています。出力の Lb と Ub は、パーティション化された各次元の並列ループの上限と下限を示します。

collapse 節がないと、LIBOMPTARGET_DEBUG=1 による出力は、50 行目 target 領域に対しの次のような情報を示します。

```
Libomptarget --> Launching target execution __omp_offloading_3d_9b5f515d__Z4main_l45
with pointer 0x0000000000143d5d8 (index=1).
Target LEVEL0 RTL --> Executing a kernel 0x0000000000143d5d8...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 32
Target LEVEL0 RTL --> Preferred group size is multiple of 64
Target LEVEL0 RTL --> Level 0: Lb = 0, Ub = 7, Stride = 1
Target LEVEL0 RTL --> Group sizes = {1, 1, 1}
Target LEVEL0 RTL --> Group counts = {8, 1, 1}
```

collapse 節がない場合、最も外側のループの上限は (BLOCKS) = 8 であるため、並列ループの反復回数は 8 になります。この場合、1 つの work-group が 8 つの work-item を持ちます (work-group の合計は 1 x 1 x 1 = 1 で、各 work-group のサイズは 8 x 1 x 1 = 8 work-item)。カーネルは SIMD32 でベクトル化されており、32 個の work-item が 1 つの sub-group にまとめられています。work-group ごとに work-item が 1 つしかないため、各 work-group には 1 つの SIMD レーンのみがアクティブな sub-group も 1 つしかないことになります。

parallel for ディレクティブに collapse 節を追加することで、work-item の数を増やして GPU の並列性を高めることができます。最初に、次の例に示すように collapse(2) 節を追加してみます。

```
/* collapse 節を指定してカーネルをオフロードします */
  #pragma omp target teams distribute parallel for collapse(2) private(b, i, j, k, l)
  for (b = 0; b < BLOCKS; b++) {
    for (i = 0; i < P; i++) {
      for (j = 0; j < P; j++) {
        for (k = 0; k < P; k++) {
          double ur = 0.;
          double us = 0.;
          double ut = 0.;

          for (l = 0; l < P; l++) {
            ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
            us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
            ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
          }

          w[IDX4(b, i, j, k)] = ur * us * ut;
        }
      }
    }
  }
```

`LIBOMPTARGET_DEBUG=1` による出力は、`collapse(2)` が指定された場合のパーティション化の情報を次のように示します。

```
Libomptarget --> Launching target execution __omp_offloading_3d_9b5f515f__Z4main_l45
with pointer 0x00000000017f45d8 (index=1).
Target LEVEL0 RTL --> Executing a kernel 0x00000000017f45d8...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 32
Target LEVEL0 RTL --> Preferred group size is multiple of 64
Target LEVEL0 RTL --> Level 0: Lb = 0, Ub = 15, Stride = 1
Target LEVEL0 RTL --> Level 1: Lb = 0, Ub = 7, Stride = 1
Target LEVEL0 RTL --> Group sizes = {1, 1, 1}
Target LEVEL0 RTL --> Group counts = {16, 8, 1}
```

`collapse(2)` の場合、並列ループの反復回数は `BLOCKS` x `P` = 8 x 16 = 128 となります。この場合、最終的に work-group は 128 になり、各 work-group には 1 個の work-item があります (work-group の合計は 16 x 8 x 1 = 128 で、各 work-group のサイズは 1 x 1 x 1 = 1 work-item です)。カーネルは SIMD32 でベクトル化されており、32 個の work-item が 1 つの sub-group にまとめられています。つまり work-group ごとに work-item が 1 つしかないため、各 work-group には 1 つの SIMD レーンのみがアクティブが sub-group が 1 つしかないことになります。

一方、`collapse(3)` 節にすると、`LIBOMPTARGET_DEBUG=1` の出力は次のようなパーティション化の情報を示します。

```
Libomptarget --> Launching target execution __omp_offloading_3d_9b5f5160__Z4main_l45
with pointer 0x0000000001728d08 (index=1).
Target LEVEL0 RTL --> Executing a kernel 0x0000000001728d08...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 32
Target LEVEL0 RTL --> Preferred group size is multiple of 64
Target LEVEL0 RTL --> Level 0: Lb = 0, Ub = 15, Stride = 1
Target LEVEL0 RTL --> Level 1: Lb = 0, Ub = 15, Stride = 1
Target LEVEL0 RTL --> Level 2: Lb = 0, Ub = 7, Stride = 1
Target LEVEL0 RTL --> Group sizes = {8, 1, 1}
Target LEVEL0 RTL --> Group counts = {2, 16, 8}
```

`collapse(3)` の場合、並列ループの反復数は `BLOCKS` x `P` x `P` = 8 x 16 x 16 = 2048 となります。この場合、最終的に work-group は 256 になり、各 work-group には 8 個の work-item があります (work-group の合計は 2 x 16 x 8 = 256 で、各 work-group のサイズは 8 x 1 x 1 = 8 work-item です)。カーネルは SIMD32 でベクトル化されており、32 個の work-item が 1 つの sub-group にまとめられています。work-group ごとに 8 つの work-item しかないため、8 つの SIMD レーンがアクティブな sub-group は 1 つしかないことになります。

`collapse(3)` 節の代わりに `collapse(4)` 節を使用すると、`LIBOMPTARGET_DEBUG=1` の出力は次のようなパーティション化情報を示します。

```
Target LEVEL0 RTL --> Executing a kernel 0x0000000001aab5d8...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 32
Target LEVEL0 RTL --> Preferred group size is multiple of 64
Target LEVEL0 RTL --> Level 0: Lb = 0, Ub = 32767, Stride = 1
Target LEVEL0 RTL --> Group sizes = {64, 1, 1}
Target LEVEL0 RTL --> Group counts = {512, 1, 1}
```

`collapse(4)` の場合、並列ループの反復回数は `BLOCKS` x `P` x `P` x `P` = 8 x 16 x 16 x 16= 32768 となります。この場合、最終的に work-group は 512 になり、各 work-group には 64 個の work-item があります (work-group の合計は 512 x 1 x 8 = 512 で、各 work-group のサイズは 64 x 1 x 1 = 64 work-item です)。カーネルは SIMD32 でベクトル化されており、32 個の work-item が 1 つの sub-group にまとめられています。つまり、各 work-group は 2 つの sub-group を持つことになります。

`collapse` 節を使用すると、入れ子になったループの実行時間が大幅に短縮されます。ここで使用する GPU (1 スタックのみ) で実行した各バージョンのパフォーマンスは次のようになります。

```
no collapse version :0.002430 seconds
collapse(2) version :0.000839 seconds
collapse(3) version : 0.000321 seconds
collapse(4) version : 0.000325 seconds
```

上記の結果から、`collapse(3)` 節または `collapse(4)` 節を追加すると、パフォーマンスがおよそ 7.5 倍向上したことが分かります (0.000321 秒 vs 0.002430 秒)。

**注:**

- GPU 上では `collapse` 節で実際のループを融合できないかもしれませんが、この節はコンパイラーとランタイムに入れ子になったループの並列性の度合いを伝え、ND-range のパーティション化に利用されることに留意してください。
- ベクトルロードとストアの利点を活用するため、入れ子になったループの最も内側のループは、ベクトル化できるように `collapse` で融合しないことを推奨します。最も内側のループがユニットストライドで、反復数が SIMD 幅と同じになるよう十分に大きければ、最良のパフォーマンスを得ることができます。

## データ転送とメモリー割り当てを最小化する

GPU に計算をオフロードする場合、ホストとデバイス間のデータ転送を最小にして、デバイス上のメモリー割り当てを減らすことが重要です。これには、次に示すようにいくつかの方法があります。

### `target enter data` と `target exit data` ディレクティブを使用する

変数が複数の `target` 構造で使用される場合、`target enter data` と `target exit data` ディレクティブでコード領域を囲むことで、ホストとデバイス間のデータ転送を最小限にすることができます。

ホストからデバイスにデータを転送する最初の `target` 構造の前に `target enter data` ディレクティブを配置し、デバイスからホストにデータを転送する最後の `target` 構造の後に `target exit data` ディレクティブを配置します。

次の例では、2 つの `target` 構造 (30 行目と 50 行目) があり、それぞれの `target` 構造は配列 `dx` と `u` を読み取り、配列 `w` に書き込むことを想定しています。

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <time.h>
4.  #include <math.h>
5.  #include <omp.h>
6.  #define P 16
7.  #define BLOCKS 8
8.  #define SIZE (BLOCKS * P * P * P)
9.  #define MAX 100
10. #define scaled_rand() ((rand() % MAX) / (1.0 * MAX))
11. #define IDX2(i, j) (i * P + j)
12. #define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)
13. int main(void) {
14.   double w[SIZE];             /* 出力 */
15.   double u[SIZE], dx[P * P]; /* 入力 */
16.   int b, i, j, k, l;          /* ループカウンター */
17.   double start, end;          /* タイマー */
```

```
18.   omp_set_default_device(0);
19.   /* 起動時のオーバーヘッドを計測しないよう、ダミーの target 領域を追加しています */
20.   #pragma omp target
21.   { ; }
22.   /* 乱数値で入力を初期化します */
23.   srand(0);
24.   for (int i = 0; i < SIZE; i++)
25.     u[i] = scaled_rand();
26.   for (int i = 0; i < P * P; i++)
27.     dx[i] = scaled_rand();
28.   start = omp_get_wtime();
29.   /* カーネル #1 をオフロード */
30.   #pragma omp target teams distribute parallel for collapse(4) \
31.   map(to: u[0:SIZE], dx[0:P * P]) map(from: w[0:SIZE]) private(b, i, j, k, l)
32.   for (b = 0; b < BLOCKS; b++) {
33.     for (i = 0; i < P; i++) {
34.       for (j = 0; j < P; j++) {
35.         for (k = 0; k < P; k++) {
36.           double ur = 0.;
37.           double us = 0.;
38.           double ut = 0.;
39.           for (l = 0; l < P; l++) {
40.             ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
41.             us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
42.             ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
43.           }
44.           w[IDX4(b, i, j, k)] = ur * us * ut;
45.         }
46.       }
47.     }
48.   }
49.   /* カーネル #2 をオフロード */
50.   #pragma omp target teams distribute parallel for collapse(4) \
51.   map(to: u[0:SIZE], dx[0:P * P]) map(tofrom: w[0:SIZE]) private(b, i, j, k, l)
52.   for (b = 0; b < BLOCKS; b++) {
53.     for (i = 0; i < P; i++) {
54.       for (j = 0; j < P; j++) {
55.         for (k = 0; k < P; k++) {
56.           double ur = b + i + j - k;
57.           double us = b + i + j - k;
58.           double ut = b + i + j - k;
59.           for (l = 0; l < P; l++) {
60.             ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
61.             us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
62.             ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
63.           }
64.           w[IDX4(b, i, j, k)] += ur * us * ut;
65.         }
66.       }
67.     }
68.   }
69.   end = omp_get_wtime();
70.   /* 結果を出力します */
71.   printf("target region: w[0]=%lf time=%lf\n", w[0], end - start);
72.   return 0;
73. }
```

**コンパイルコマンド**

```
$ icx-cl -fiopenmp -fopenmp-targets=spir64 test_no_target_enter_exit_data.cpp
```

**実行コマンド**

```
$ export OMP_TARGET_OFFLOAD=MANDATORY
$ export ZE_AFFINITY_MASK=0.0
$ export LIBOMPTARGET_DEBUG=1
./a.out
```

最初の `target` 構造 (30 行目) に到達するとランタイムは次のように処理します。

- 配列 `dx` と `u` は `map` 節で `to` マップタイプとして指定されているため、デバイス上に `dx` と `u` のストレージが割り当てられ、ホスト上の `dx` と `u` の値が対応するデバイスの配列にコピーされます。
- 配列 `w` は `map` 節で `from` マップタイプとして指定されているため、デバイス上には初期化されない配列 `w` が割り当てられます。

最初の `target` 構造の最後で次のことが行われます。

- 配列 `w` は map 節で `from` マップタイプとして指定されているため、デバイス上の配列 `w` の値はホスト上の配列 `w` にコピーされます。

2 番目の `target` 構造 (50 行目) に到達するとランタイムは次のように処理します。

- 配列 `dx` と `u` は `map` 節で `to` マップタイプとして指定されているため、デバイス上に `dx` と `u` のストレージが割り当てられ、ホスト上の `dx` と `u` の値が対応するデバイスの配列にコピーされます。

2 番目の `target` 構造の最後で次のことが行われます。

- 配列 `w` は `map` 節で `from` マップタイプとして指定されているため、デバイス上の配列 `w` の値はホスト上の配列 `w` にコピーされます。

次の `LIBOMPTARGET_DEBUG=1` の出力は、両方の `target` 構造 (30 行目と 50 行目) で同じデータがパーティション化されていることを示しています。

```
Libomptarget --> Launching target execution __omp_offloading_3d_15ece5c8__Z4main_l42
with pointer 0x00000000024cb5d8 (index=1).
Target LEVEL0 RTL --> Executing a kernel 0x00000000024cb5d8...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 32
Target LEVEL0 RTL --> Preferred group size is multiple of 64
Target LEVEL0 RTL --> Level 0: Lb = 0, Ub = 32767, Stride = 1
Target LEVEL0 RTL --> Group sizes = {64, 1, 1}
Target LEVEL0 RTL --> Group counts = {512, 1, 1}
Target LEVEL0 RTL --> Executing a kernel 0x0000000002b9c5e0...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 32
Target LEVEL0 RTL --> Preferred group size is multiple of 64
Target LEVEL0 RTL --> Level 0: Lb = 0, Ub = 32767, Stride = 1
Target LEVEL0 RTL --> Group sizes = {64, 1, 1}
Target LEVEL0 RTL --> Group counts = {512, 1, 1}
Target LEVEL0 RTL --> Kernel Pointer argument 0 (value: 0xff00ffffffee0000) was set
successfully for device 0.
```

(両方の target 領域で) 転送されたデータ量は、LIBOMPTARGET_DEBUG=1 の出力を grep "Libomptarget --> Moving" を実行して確認できます。

```
$ grep "Libomptarget --> Moving" test_no_target_enter_exit_data.debug
Libomptarget --> Moving 2048 bytes (hst:0x00007fff60f05030) ->
(tgt:0xff00fffffffe0000)
Libomptarget --> Moving 262144 bytes (hst:0x00007fff60ec5030) ->
(tgt:0xff00ffffffee0000)
Libomptarget --> Moving 262144 bytes (tgt:0xff00ffffff20000) ->
(hst:0x00007fff60e85030)
Libomptarget --> Moving 2048 bytes (hst:0x00007fff60f05030) ->
(tgt:0xff00fffffffe0000)
Libomptarget --> Moving 262144 bytes (hst:0x00007fff60ec5030) ->
(tgt:0xff00ffffffee0000)
Libomptarget --> Moving 262144 bytes (hst:0x00007fff60e85030) ->
(tgt:0xff00ffffff20000)
Libomptarget --> Moving 262144 bytes (tgt:0xff00ffffff20000) ->
(hst:0x00007fff60e85030)
```

次の修正例に示すように、target enter data と target exit data ディレクティブを使用することで、ホストからデバイス、またはデバイスからホストへのデータコピーを減らすことができます。

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <time.h>
4.  #include <math.h>
5.  #include <omp.h>
6.  #define P 16
7.  #define BLOCKS 8
8.  #define SIZE (BLOCKS * P * P * P)
9.  #define MAX 100
10. #define scaled_rand() ((rand() % MAX) / (1.0 * MAX))
11. #define IDX2(i, j) (i * P + j)
12. #define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)
13. int main(void) {
14.   double w[SIZE];            /* 出力 */
15.   double u[SIZE], dx[P * P]; /* 入力 */
16.   int b, i, j, k, l;         /* ループカウンター */
17.   double start, end;         /* タイマー */
18.   omp_set_default_device(0);
19.   /* 起動時のオーバーヘッドを計測しないよう、ダミーの target 領域を追加しています */
20.   #pragma omp target
21.   { ; }
22.   /* 乱数値で入力を初期化します */
23.   srand(0);
24.   for (int i = 0; i < SIZE; i++)
25.     u[i] = scaled_rand();
26.   for (int i = 0; i < P * P; i++)
27.     dx[i] = scaled_rand();
28.   start = omp_get_wtime();
29.   /* データをデバイスにマップします。w はデフォルトの map(tofrom: w[0:SIZE])
30.   にはせず、map(alloc: w[0:SIZE]) にします */
31.   #pragma omp target enter data map(to: u[0:SIZE], dx[0:P * P]) \
32.   map(alloc: w[0:SIZE])
33.   /* カーネル #1 をオフロード */
34.   #pragma omp target teams distribute parallel for collapse(4) private(b, i, j, k, l)
35.   for (b = 0; b < BLOCKS; b++) {
36.     for (i = 0; i < P; i++) {
37.       for (j = 0; j < P; j++) {
38.         for (k = 0; k < P; k++) {
39.           double ur = 0.;
40.           double us = 0.;
```

```
41.           double ut = 0.;
42.           for (l = 0; l < P; l++) {
43.             ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
44.             us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
45.             ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
46.           }
47.           w[IDX4(b, i, j, k)] = ur * us * ut;
48.         }
49.       }
50.     }
51.   }
52.   /* カーネル #2 をオフロード */
53.   #pragma omp target teams distribute parallel for collapse(4) private(b, i, j, k, l)
54.   for (b = 0; b < BLOCKS; b++) {
55.     for (i = 0; i < P; i++) {
56.       for (j = 0; j < P; j++) {
57.         for (k = 0; k < P; k++) {
58.           double ur = b + i + j - k;
59.           double us = b + i + j - k;
60.           double ut = b + i + j - k;
61.           for (l = 0; l < P; l++) {
62.             ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
63.             us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
64.             ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
65.           }
66.           w[IDX4(b, i, j, k)] += ur * us * ut;
67.         }
68.       }
69.     }
70.   }
71.   #pragma omp target exit data map(from: w[0:SIZE])
72.   end = omp_get_wtime();
73.   /* 結果を出力します */
74.   printf("target region: w[0]=%lf time=%lf\n", w[0], end - start);
75.   return 0;
76. }
```

この例では、`target enter data` ディレクティブ (31 行目) に到達すると次のことが行われます。

- 配列 `dx` と `u` は `map` 節で to マップタイプとして指定されているため、デバイス上に `dx` と `u` のストレージが割り当てられ、ホスト上の `dx` と `u` の値が対応するデバイスの配列にコピーされます。
- 配列 `w` は `map` 節で `alloc` マップタイプとして指定されているため、デバイス上には初期化されない配列 `w` が割り当てられます。

最初の `target` 構造 (34 行目) に到達するとランタイムは次のように処理します。

- ランタイムは、配列 `dx`、`u`、および `w` がデバイスに存在するか確認します。存在するとデータ転送は行われません。

最初の `target` 構造の最後で次のことが行われます。

- ランタイムは、配列 `dx`、`u`、および `w` のストレージがデバイス上に保持されると想定し、デバイスからホストへのコピーバックは行いません。

2 番目の `target` 構造 (53 行目) に到達するとランタイムは次のように処理します。

- ここでもホストからデバイスへのデータ転送は行われません。

2 番目の `target` 構造の最後で次のことが行われます。

- ランタイムは、配列 `dx`、u、および `w` のストレージがデバイス上に保持されると想定し、デバイスからホストへのコピーバックは行いません。

`target exit data` ディレクティブ (71 行目) に到達すると次のことが行われます。

- 配列 `w` は `map` 節で `from` マップタイプとして指定されているため、デバイス上の配列 `w` の値はホスト上の配列 `w` にコピーされます。

`target enter data` と `target exit data` ディレクティブを使用することで、特定の GPU (1 スタックのみ) の実行時間が短縮されました。

```
No target enter/exit data version : 0.001204 seconds target enter/exit data
version   : 0.000934 seconds
```

`LIBOMPTARGET_DEBUG=1` の出力は、データのパーティション化が両方の例 (`target enter data` と `target exit data` あり/なし) で同じであることを示しています。

```
Libomptarget --> Looking up mapping(HstPtrBegin=0x00007ffd899939c0, Size=2048)...
Libomptarget --> Mapping exists with HstPtrBegin=0x00007ffd899939c0,
TgtPtrBegin=0xff00ffffffee0000, Size=2048, DynRefCount=2 (update suppressed),
HoldRefCount=0
Libomptarget --> Obtained target argument (Begin: 0xff00ffffffee0000, Offset: 0) from
host pointer 0x00007ffd899939c0
Libomptarget --> Looking up mapping(HstPtrBegin=0x00007ffd899539c0, Size=262144)...
Libomptarget --> Mapping exists with HstPtrBegin=0x00007ffd899539c0,
TgtPtrBegin=0xff00ffffffef0000, Size=262144, DynRefCount=2 (update suppressed),
HoldRefCount=0
Libomptarget --> Obtained target argument (Begin: 0xff00ffffffef0000, Offset: 0) from
host pointer 0x00007ffd899539c0
Libomptarget --> Looking up mapping(HstPtrBegin=0x00007ffd899139c0, Size=262144)...
```

```
Libomptarget --> Launching target execution __omp_offloading_3d_fadb4d__Z4main_l47
with pointer 0x0000000002b9c5d8 (index=1).
Target LEVEL0 RTL --> Executing a kernel 0x0000000002b9c5d8...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 32
Target LEVEL0 RTL --> Preferred group size is multiple of 64
Target LEVEL0 RTL --> Level 0: Lb = 0, Ub = 32767, Stride = 1
Target LEVEL0 RTL --> Group sizes = {64, 1, 1}
Target LEVEL0 RTL --> Group counts = {512, 1, 1}
```

`target enter data` と `target exit data` によるパフォーマンスの向上は、データ転送の削減によるものであり、ここでは次の 3 つのデータ転送が行われています。

```
$ grep "Libomptarget --> Moving" test_target_enter_exit_data.debug
Libomptarget --> Moving 262144 bytes (hst:0x00007ffd899539c0) ->
(tgt:0xff00ffffffef0000)
Libomptarget --> Moving 2048 bytes (hst:0x00007ffd899939c0) ->
(tgt:0xff00ffffffee0000)
Libomptarget --> Moving 262144 bytes (tgt:0xff00fffffff30000) ->
(hst:0x00007ffd899139c0)
```

## マップタイプの選択

パフォーマンスを向上するには、マップされる変数のマップタイプを、その変数の `target` 構造での利用状況と一致させることが重要です。

次の例では、配列 `u` と `dx` は `target` 構造では読み取りのみであり、配列 `w` は `target` 構造で書き込まれています。しかし、これらの変数のマップタイプはすべて `tofrom` に指定されており、非効率です。

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <time.h>
4.
5.  #include <math.h>
6.  #include <omp.h>
7.
8.  #define P 16
9.  #define BLOCKS 8
10. #define SIZE (BLOCKS * P * P * P)
11.
12. #define MAX 100
13. #define scaled_rand() ((rand() % MAX) / (1.0 * MAX))
14.
15. #define IDX2(i, j) (i * P + j)
16. #define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)
17.
18. int main(void) {
19.   double w[SIZE];             /* 出力 */
20.   double u[SIZE], dx[P * P]; /* 入力 */
21.   int b, i, j, k, l;          /* ループカウンター */
22.   double start, end;          /* タイマー */
23.
24.   omp_set_default_device(0);
25.
26.   /* 起動時のオーバーヘッドを計測しないよう、ダミーの target 領域を追加しています */
27.   #pragma omp target
28.   { ; }
29.
30.   /* 乱数値で入力を初期化します */
31.   srand(0);
32.   for (int i = 0; i < SIZE; i++)
33.     u[i] = scaled_rand();
34.
35.   for (int i = 0; i < P * P; i++)
36.     dx[i] = scaled_rand();
37.
38.   start = omp_get_wtime();
39.
40.   #pragma omp target teams distribute parallel for \
41.     private(b, i, j, k, l) \
42.     map(tofrom: u[0:SIZE], dx[0:P * P]) \
43.     map(tofrom: w [0:SIZE])
44.   for (int n = 0; n < SIZE; n++) {
45.     k = n - (n / P) * P;
46.     j = (n - k) / P;
47.     i = (n - (j * P + k)) / (P * P);
48.     b = n / (P * P * P);
49.
50.     double ur = 0.;
51.     double us = 0.;
52.     double ut = 0.;
53.
54.     for (l = 0; l < P; l++) {
55.       ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
```

```
56.       us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
57.       ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
58.     }
59.
60.     w[IDX4(b, i, j, k)] = ur * us * ut;
61.   }
62.
63.   end = omp_get_wtime();
64.
65.   printf("offload: w[0]=%lf time=%lf\n", w[0], end - start);
66.
67.   return 0;
68. }
```

**コンパイルコマンド**

```
$ icpx -fiopenmp -fopenmp-targets=spir64 test_map_tofrom.cpp
```

**実行コマンド**

```
$ export OMP_TARGET_OFFLOAD=MANDATORY
$ export ZE_AFFINITY_MASK=0.0
$ export LIBOMPTARGET_DEBUG=1
$ ./a.out
```

パフォーマンスを向上するには、次に示すように `u` と `dx` のマップタイプを `to` に `w` のマップタイプは `from` にする必要があります。

```
#pragma omp target teams distribute parallel for \
    private(b, i, j, k, l) \
    map(to: u[0:SIZE], dx[0:P * P]) \
    map(from: w [0:SIZE])
  for (int n = 0; n < SIZE; n++) {
    k = n - (n / P) * P;
    j = (n - k) / P;
    i = (n - (j * P + k)) / (P * P);
    b = n / (P * P * P);

    double ur = 0.;
    double us = 0.;
    double ut = 0.;

    for (l = 0; l < P; l++) {
      ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
      us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
      ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
    }

    w[IDX4(b, i, j, k)] = ur * us * ut;
  }
```

より明確なマップタイプ (`tofrom` ではなく `to` または `from`) を使用することで、特定の GPU (1 スタックのみ) での実行時間を短縮できました。

```
tofrom map-types version     : 0.001141 seconds
to or from map-types version : 0.000908 seconds
```

`LIBOMPTARGET_DEBUG=1` の出力は、`u`、`dx`、および `w` に `tofrom` マップタイプを指定すると、ホストとデバイス間で不要なデータ転送があることを示しています。`tofrom` マップタイプでは、`u`、`dx`、`w` の値がホストからデバイスに、またデバイスからホストに合計 6 回コピーされます。

```
$ grep "Libomptarget --> Moving" test_map_tofrom.debug
Libomptarget --> Moving 2048 bytes (hst:0x00007fff1f6ad540) ->
(tgt:0xff00fffffffe0000)
Libomptarget --> Moving 262144 bytes (hst:0x00007fff1f66d540) ->
(tgt:0xff00ffffffee0000)
Libomptarget --> Moving 262144 bytes (hst:0x00007fff1f62d540) ->
(tgt:0xff00fffffff20000)
Libomptarget --> Moving 262144 bytes (tgt:0xff00fffffff20000) ->
(hst:0x00007fff1f62d540)
Libomptarget --> Moving 262144 bytes (tgt:0xff00ffffffee0000) ->
(hst:0x00007fff1f66d540)
Libomptarget --> Moving 2048 bytes (tgt:0xff00fffffffe0000) ->
(hst:0x00007fff1f6ad540)
```

より明確なマップタイプ (`to` または `from`) を指定することで、`u` と `dx` の値をホストからデバイスにコピーする 2 回の転送と、`w` の値をデバイスからホストにコピーする 1 回の転送の合計 3 回のデータ転送に抑えることができます。

```
$ grep "Libomptarget --> Moving" test_map_to_or_from.debug
Libomptarget --> Moving 2048 bytes (hst:0x00007fffc2258fd0) ->
(tgt:0xff00fffffffe0000)
Libomptarget --> Moving 262144 bytes (hst:0x00007fffc2218fd0) ->
(tgt:0xff00ffffffee0000)
Libomptarget --> Moving 262144 bytes (tgt:0xff00fffffff20000) ->
(hst:0x00007fffc21d8fd0)
```

## 読み取り専用のスカラー変数をマップしない

コンパイラーは、`target` 構造の読み取り専用スカラー変数がマップされず、`target` 構造の `firstprivate` 節に指定されるか、もしくはどの節にも指定されていない場合、最も効率の良いコードを生成できます。スカラー変数が `target` 構造のいずれの節にも指定されない場合、デフォルトは `firstprivate` になることに注意してください。

読み取り専用のスカラー変数を `map(to:)` 節に指定すると、デバイス上で不要なメモリー割り当てやホストからデバイスへのデータ転送が発生します。一方、`target` 構造で読み取り専用のスカラー変数を `firstprivate` として指定すると、カーネル起動時に引数として渡され、変数のメモリー割り当てやコピーは必要ありません。

次の例では、入れ子になったループを GPU にオフロードしています。`target` 構造の `map(to:)` 節には、読み取り専用の 3 つのスカラー変数 `s1`、`s2`、`s3` が指定されています。

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <time.h>
4.
5.  #include <math.h>
6.  #include <omp.h>
7.
8.  #define P 16
9.  #define BLOCKS 8
10. #define SIZE (BLOCKS * P * P * P)
11.
12. #define MAX 100
13. #define scaled_rand() ((rand() % MAX) / (1.0 * MAX))
14.
15. #define IDX2(i, j) (i * P + j)
16. #define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)
17.
18. int main(void) {
19.   double w[SIZE];            /* 出力 */
20.   double u[SIZE], dx[P * P]; /* 入力 */
21.   double s1, s2, s3;         /* スカラー */
22.   int b, i, j, k, l;         /* ループカウンター */
```

```
23.   double start, end;         /* タイマー */
24.
25.   omp_set_default_device(0);
26.
27.   /* 起動時のオーバーヘッドを計測しないよう、ダミーの target 領域を追加しています */
28.   #pragma omp target
29.   { ; }
30.
31.   /* 乱数値で入力を初期化します */
32.   srand(0);
33.   for (int i = 0; i < SIZE; i++)
34.     u[i] = scaled_rand();
35.
36.   for (int i = 0; i < P * P; i++)
37.     dx[i] = scaled_rand();
38.
39.   /* スカラーの初期化 */
40.   s1 = u[SIZE / 2];
41.   s2 = scaled_rand();
42.   s3 = 0.145;
43.
44.   /* デバイスへデータをマップ */
45.   #pragma omp target enter data map(to: u[0:SIZE], dx[0:P * P])
46.
47.   start = omp_get_wtime();
48.
49.   /* collapse 節を指定してカーネルをオフロードします */
50.   #pragma omp target teams distribute parallel for collapse(4) \
51.     map(to: s1, s2, s3) private(b, i, j, k, l)
52.   for (b = 0; b < BLOCKS; b++) {
53.     for (i = 0; i < P; i++) {
54.       for (j = 0; j < P; j++) {
55.         for (k = 0; k < P; k++) {
56.           double ur = 0.;
57.           double us = 0.;
58.           double ut = 0.;
59.
60.           for (l = 0; l < P; l++) {
61.             ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)] + s1;
62.             us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)] - s2;
63.             ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)] * s3;
64.           }
65.
66.           w[IDX4(b, i, j, k)] = ur * us * ut;
67.         }
68.       }
69.     }
70.   }
71.
72.   end = omp_get_wtime();
73.
74.   #pragma omp target exit data map(from: w[0:SIZE])
75.
76.   /* 結果を出力します */
77.   printf("collapse-clause: w[0]=%lf time=%lf\n", w[0], end - start);
78.
79.   return 0;
80. }
```

**コンパイルコマンド**

```
$ icpx -fiopenmp -fopenmp-targets=spir64 test_scalars_map.cpp
```

**実行コマンド**

```
$ export OMP_TARGET_OFFLOAD=MANDATORY
$ export ZE_AFFINITY_MASK=0.0
$ export LIBOMPTARGET_DEBUG=1
$ ./a.out
```

次の例に示すように、target 構造の firstprivate 節で s1、s2、および s3 を指定するか、どの節にも指定しないほうが効率的です。

```
/* collapse 節を指定してカーネルをオフロードします */
  #pragma omp target teams distribute parallel for collapse(4) \
    firstprivate(s1, s2, s3) private(b, i, j, k, l)
  for (b = 0; b < BLOCKS; b++) {
    for (i = 0; i < P; i++) {
      for (j = 0; j < P; j++) {
        for (k = 0; k < P; k++) {
          double ur = 0.;
          double us = 0.;
          double ut = 0.;

          for (l = 0; l < P; l++) {
            ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)] + s1;
            us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)] - s2;
            ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)] * s3;
          }

          w[IDX4(b, i, j, k)] = ur * us * ut;
        }
      }
    }
  }
```

map(to:s1, s2, s3) の代わりに firstprivate(s1, s2, s3) を使用すると、特定の GPU (1 スタックのみ) での実行時間が短縮できました。

```
map(to:s1,s2,s3) バージョン : 0.001324 seconds
firstprivate(s1,s2,s3) バージョン : 0.000730 seconds
```

LIBOMPTARGET_DEBUG=1 の出力では、データのパーティション化の両方の例 map(to:s1, s2, s3) と firstprivate(to:s1, s2, s3) で同じであることが分かります。

```
Libomptarget --> Launching target execution __omp_offloading_3d_9b49d7d8__Z4main_l51
with pointer 0x0000000002b295d8 (index=1).
Target LEVEL0 RTL --> Executing a kernel 0x0000000002b295d8...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 32
Target LEVEL0 RTL --> Preferred group size is multiple of 64
Target LEVEL0 RTL --> Level 0: Lb = 0, Ub = 32767, Stride = 1
Target LEVEL0 RTL --> Group sizes = {64, 1, 1}
Target LEVEL0 RTL --> Group counts = {512, 1, 1}
```

```
Libomptarget --> Launching target execution __omp_offloading_3d_9b49d7dd__Z4main_l51
with pointer 0x0000000001f475d8 (index=1).
Target LEVEL0 RTL --> Executing a kernel 0x0000000001f475d8...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 32
Target LEVEL0 RTL --> Preferred group size is multiple of 64
Target LEVEL0 RTL --> Level 0: Lb = 0, Ub = 32767, Stride = 1
Target LEVEL0 RTL --> Group sizes = {64, 1, 1}
Target LEVEL0 RTL --> Group counts = {512, 1, 1}
```

map(to:s1, s2, s3) 節を指定すると、デバイスメモリー割り当てとメモリー転送が発生します。

次の LIBOMPTARGET_DEBUG=1 の出力は、map(to:s1, s2, s3) 節を使用した際のデバイス上のメモリー割り当てに関連する情報を示しています。

```
Target LEVEL0 RTL --> Memory usage for device memory, device 0x000000000278e470
Target LEVEL0 RTL --> -- Allocator:       Native,         Pool
Target LEVEL0 RTL --> -- Requested:      1179648,       526360
Target LEVEL0 RTL --> -- Allocated:      1179648,       526528
Target LEVEL0 RTL --> -- Freed    :      1179648,       262336
Target LEVEL0 RTL --> -- InUse    :            0,       264192
Target LEVEL0 RTL --> -- PeakUse  :      1179648,       526528
Target LEVEL0 RTL --> -- NumAllocs:            3,            6
```

割り当てられたメモリーは 1,179,648 バイトであり、(Pool からの) 割り当て数は 3 つの配列 (dx、u、w) と 3 つのスカラー (s1、s2、s3) の 6 つであることに注意してください。

一方、firstprivate(s1, s2, s3) 節を指定すると LIBOMPTARGET_DEBUG=1 の出力は、デバイスメモリーの割り当てが減少していることを示しています。以下に示すように、割り当てメモリーサイズは、1,179,648 から 1,114,112 バイトに減少し (64 K バイト)、Pool からの割り当て数は 6 から 3 に減少しています。

```
Target LEVEL0 RTL --> Memory usage for device memory, device 0x0000000001bab440
Target LEVEL0 RTL --> -- Allocator:       Native,         Pool
Target LEVEL0 RTL --> -- Requested:      1114112,       526336
Target LEVEL0 RTL --> -- Allocated:      1114112,       526336
Target LEVEL0 RTL --> -- Freed    :      1114112,       262144
Target LEVEL0 RTL --> -- InUse    :            0,       264192
Target LEVEL0 RTL --> -- PeakUse  :      1114112,       526336
Target LEVEL0 RTL --> -- NumAllocs:            2,            3
```

また、map(to:) 節を指定すると、メモリー割り当てが増加するだけでなく、ホストからデバイスへのデータ転送がさらに多くなります。これは、LIBOMPTARGET_DEBUG=1 の出力に、grep "Libomptarget --> Moving" コマンドを実行すると分かります。

```
$ grep "Libomptarget --> Moving" test_scalars_map.debug
Libomptarget --> Moving 262144 bytes (hst:0x00007ffdf5526760) ->
(tgt:0xff00ffffffef0000)
Libomptarget --> Moving 2048 bytes (hst:0x00007ffdf5566760) ->
(tgt:0xff00ffffffee0000)
Libomptarget --> Moving 8 bytes (hst:0x00007ffdf55670a0) -> (tgt:0xff00ffffffed0000)
Libomptarget --> Moving 8 bytes (hst:0x00007ffdf55670a8) -> (tgt:0xff00ffffffed0040)
Libomptarget --> Moving 8 bytes (hst:0x00007ffdf55670b0) -> (tgt:0xff00ffffffed0080)
Libomptarget --> Moving 262144 bytes (hst:0x00007ffdf54e6760) ->
(tgt:0xff00ffffffff30000)
Libomptarget --> Moving 262144 bytes (tgt:0xff00ffffffff30000) ->
(hst:0x00007ffdf54e6760)
```

また、firstprivate(to:s1, s2, s3) 節を指定すると LIBOMPTARGET_DEBUG=1 の出力は次のようになります。

```
$ grep "Libomptarget --> Moving" test_scalars_fp.debug
Libomptarget --> Moving 262144 bytes (hst:0x00007ffda809c4a0) ->
(tgt:0xff00ffffffef0000)
Libomptarget --> Moving 2048 bytes (hst:0x00007ffda80dc4a0) ->
(tgt:0xff00ffffffee0000)
Libomptarget --> Moving 262144 bytes (hst:0x00007ffda805c4a0) ->
(tgt:0xff00ffffffff30000)
Libomptarget --> Moving 262144 bytes (tgt:0xff00ffffffff30000) ->
(hst:0x00007ffda805c4a0)
```

map(to:s1, s2, s3) の例では、さらの 3 つのデータが転送 (それぞれ 8 バイト) されていることに注意してください。この転送では、s1、s2、および s3 の値がホストからデバイスにコピーされます。

## ループ境界をマップしないことで適切な ND-Range のパーティション化が可能

前述したように、`target` 構造内の読み取り専用のスカラー変数がマップされない、または `target` 構造の `firstprivate` 節に指定されるか、どの節にも指定されていない場合、コンパイラーは効率良いコードを生成できます。

これは、スカラー変数が `target` 構造の並列ループの境界にある場合、特に顕著です。ループ境界 (下限、上限、またはステップ) のいずれかがマップされている場合、デバイス上で不要なメモリー割り当てや、ホストからデバイスへのデータコピーが発生します。これはループのパーティション化でも影響を受け、不適切な ND-range のパーティション化が発生してパフォーマンスを損ねる可能性があります。

次の例を考えてみます。ここでは `parallel for` ループを GPU にオフロードしています。`for` ループの上限はスカラー変数 `upper` で `target` 構造でマップされています (48 行目)。

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <time.h>
4.
5.  #include <math.h>
6.  #include <omp.h>
7.
8.  #define P 16
9.  #define BLOCKS 8
10. #define SIZE (BLOCKS * P * P * P)
11.
12. #define MAX 100
13. #define scaled_rand() ((rand() % MAX) / (1.0 * MAX))
14.
15. #define IDX2(i, j) (i * P + j)
16. #define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)
17.
18. int main(void) {
19.   double w[SIZE];             /* 出力 */
20.   double u[SIZE], dx[P * P]; /* 入力 */
21.   int b, i, j, k, l;          /* ループカウンター */
22.   int upper;
23.   double start, end;          /* タイマー */
24.
25.   omp_set_default_device(0);
26.
27.   /* 起動時のオーバーヘッドを計測しないよう、ダミーの target 領域を追加しています */
28.   #pragma omp target
29.   { ; }
30.
31.   /* 乱数値で入力を初期化します */
32.   srand(0);
33.   for (int i = 0; i < SIZE; i++)
34.     u[i] = scaled_rand();
35.
36.   for (int i = 0; i < P * P; i++)
37.     dx[i] = scaled_rand();
38.
39.   upper = (int)dx[0] + SIZE;
40.
41.   /* デバイスへデータをマップ */
42.   #pragma omp target enter data map(to: u[0:SIZE], dx[0:P * P])
43.
44.   start = omp_get_wtime();
45.
46.   /* カーネルをオフロード */
```

```
47.   #pragma omp target teams distribute parallel for private(b, i, j, k, l) \
48.     map(to: upper)
49.   for (int n = 0; n < upper; n++) {
50.     double ur = 0.;
51.     double us = 0.;
52.     double ut = 0.;
53.
54.     k = n - (n / P) * P;
55.     j = (n - k) / P;
56.     i = (n - (j * P + k)) / (P * P);
57.     b = n / (P * P * P);
58.
59.     for (l = 0; l < P; l++) {
60.       ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
61.       us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
62.       ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
63.     }
64.
65.     w[IDX4(b, i, j, k)] = ur * us * ut;
66.   }
67.
68.   end = omp_get_wtime();
69.
70.   /* デバイスからデータをマップします */
71.   #pragma omp target exit data map(from: w[0:SIZE])
72.
73.   printf("offload: w[0]=%lf time=%lf\n", w[0], end - start);
74.
75.   return 0;
76. }
```

**コンパイルコマンド**

```
$ icpx -fiopenmp -fopenmp-targets=spir64 test_loop_bounds_map.cpp
```

**実行コマンド**

```
$ export OMP_TARGET_OFFLOAD=MANDATORY
$ export ZE_AFFINITY_MASK=0.0
$ export LIBOMPTARGET_DEBUG=1
$ ./a.out
```

`upper` はマップされているため、ホスト上の変数 `upper` の値はデバイス上の値と異なる可能性があります。そのため、実行時に `target` 領域をオフロードすると、ホストでオフロードしたループの反復回数を知ることができません。その場合、ランタイム (`libomptarget.so`) は、デバイスとカーネルのプロパティーから、GPU 全体を使用するように ND-Range のパーティション化を行います。

コンパイラーが生成するオフロードループのコードには、オフロードループの内側に最内ループ (work-item ごと) が含まれます。グローバルサイズが実際のループの反復回数よりも小さい場合、各 work-item は元のループの反復回数を処理します。逆に、グローバルサイズが実際のループの反復回数よりも大きい場合、いくつかの work-item はワークを処理しません。コンパイラーが生成するループ内の `if` 条件は、これをチェックし残りのループ本体をスキップします。

上記の例 (`upper` がマップ) の `LIBOMPTARGET_DEBUG=1` の出力は、次のように ND-range のパーティション化を示します。

```
Libomptarget --> Launching target execution __omp_offloading_3d_1ff4bf1c__Z4main_l48
with pointer 0x00000000021175d8 (index=1).
Target LEVEL0 RTL --> Executing a kernel 0x00000000021175d8...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 32
```

```
Target LEVEL0 RTL --> Preferred group size is multiple of 64
Target LEVEL0 RTL --> Group sizes = {1024, 1, 1}
Target LEVEL0 RTL --> Group counts = {512, 1, 1}
```

上記のパーティション化による work-item の総数は、512 x 1024 = 524,288 であり、実際のループの反復回数よりも大きく (32,767 回) なっています。いくつかの work-item は何のワークも処理しません。

ホスト側でオフロードされたループの反復回数が分かっていれば、さらに適切な ND-Range のパーティション化が可能です。これにより、コンパイラーとランタイムは、ループの反復回数に合わせて RD-Range をパーティション化できます。

適切なパーティション化を行うには、次の例に示すように `target` 構造の `map(to:upper)` の代わりに `firstprivate(upper)` を使用します。この方法では、コンパイラーはホスト上の `upper` 変数の値がデバイス上の `upper` 変数と同じであることを認識できます。

```
#pragma omp target teams distribute parallel for private(b, i, j, k, l) \
    firstprivate(upper)
  for (int n = 0; n < upper; n++) {
    double ur = 0.;
    double us = 0.;
    double ut = 0.;

    k = n - (n / P) * P;
    j = (n - k) / P;
    i = (n - (j * P + k)) / (P * P);
    b = n / (P * P * P);

    for (l = 0; l < P; l++) {
      ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
      us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
      ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
    }

    w[IDX4(b, i, j, k)] = ur * us * ut;
  }
```

修正した例 (`upper` が `firstprivate`) の `LIBOMPTARGET_DEBUG=1` の出力は、次のように ND-range のパーティション化を示します。

```
Libomptarget --> Launching target execution __omp_offloading_3d_1fed0edf__Z4main_l48
with pointer 0x00000000029b3d08 (index=1).
Target LEVEL0 RTL --> Executing a kernel 0x00000000029b3d08...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 32
Target LEVEL0 RTL --> Preferred group size is multiple of 64
Target LEVEL0 RTL --> Level 0: Lb = 0, Ub = 32767, Stride = 1
Target LEVEL0 RTL --> Group sizes = {64, 1, 1}
Target LEVEL0 RTL --> Group counts = {512, 1, 1}
```

上記のパーティション化による work-item の総数は、512 x 64 = 32,767 であり、実際のループの反復回数に正確に一致します。

`map(to:upper)` の代わりに `firstprivate(upper)` を使用すると、特定の GPU (1 スタックのみ) の実行時間は次のように短縮されました。

```
map(to:upper) version      : 0.000415 seconds
firstprivate(upper) version :0.000307 seconds
```

## デバイス上にメモリーを割り当て

上記に示すように `map` 節は、元のホスト変数をデバイス上の対応する変数にどのようにマップするかを決定します。ただし、`map(to:)` 節は、すべてのケースにおいてデバイス上の変数にメモリーを割り当てる最良の方法であるわけではありません。

次の例では、変数 `ur`、`us`、および `ut` はデバイス上の計算でテンポラリー配列として使用されています。配列は、`map(to:)` 節を使用してデバイスにマップされています (45 行 - 47 行)。

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <time.h>
4.
5.  #include <math.h>
6.  #include <omp.h>
7.
8.  #define P 16
9.  #define BLOCKS 8
10. #define SIZE (BLOCKS * P * P * P)
11.
12. #define MAX 100
13. #define scaled_rand() ((rand() % MAX) / (1.0 * MAX))
14.
15. #define IDX2(i, j) (i * P + j)
16. #define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)
17.
18. int main(void) {
19.   double w[SIZE];             /* 出力 */
20.   double u[SIZE], dx[P * P]; /* 入力 */
21.   double ur[SIZE], us[SIZE], ut[SIZE]; /* ワーク配列 */
22.   int b, i, j, k, l;          /* ループカウンター */
23.   double start, end;          /* タイマー */
24.
25.   omp_set_default_device(0);
26.
27.   /* 起動時のオーバーヘッドを計測しないよう、ダミーの target 領域を追加しています。*/
28.   #pragma omp target
29.   { ; }
30.
31.   /* 乱数値で入力を初期化します */
32.   srand(0);
33.   for (int i = 0; i < SIZE; i++)
34.     u[i] = scaled_rand();
35.
36.   for (int i = 0; i < P * P; i++)
37.     dx[i] = scaled_rand();
38.
39.   start = omp_get_wtime();
40.
41.   /* カーネルをオフロードします */
42.   #pragma omp target teams distribute parallel for simd simdlen(16) collapse(4) \
43.     map(to:u[0:SIZE],dx[0:P*P]) \
44.     map(from:w[0:SIZE])         \
45.     map(to:ur[0:SIZE])          \
46.     map(to:us[0:SIZE])          \
47.     map(to:ut[0:SIZE])          \
48.     private(b,i,j,k,l)
49.   for (b = 0; b < BLOCKS; b++) {
50.     for (i = 0; i < P; i++) {
51.       for (j = 0; j < P; j++) {
52.         for (k = 0; k < P; k++) {
53.           w[IDX4(b, i, j, k)] = 0.;
```

```
54.           ur[IDX4(b, i, j, k)] = 0.;
55.           us[IDX4(b, i, j, k)] = 0.;
56.           ut[IDX4(b, i, j, k)] = 0.;
57.
58.           for (l = 0; l < P; l++) {
59.             ur[IDX4(b, i, j, k)] += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
60.             us[IDX4(b, i, j, k)] += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
61.             ut[IDX4(b, i, j, k)] += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
62.           }
63.
64.           w[IDX4(b, i, j, k)] = ur[IDX4(b, i, j, k)] * us[IDX4(b, i, j, k)] *
65.               ut[IDX4(b, i, j, k)];
66.         }
67.       }
68.     }
69.   }
70.
71.   end = omp_get_wtime();
72.
73.   /* 結果を出力します */
74.   printf("collapse-clause: w[0]=%lf time=%lf\n", w[0], end - start);
75.
76.   return 0;
77. }
```

**コンパイルコマンド**

```
$ icpx -fiopenmp -fopenmp-targets=spir64 test_map_to.cpp
```

**実行コマンド**

```
$ export OMP_TARGET_OFFLOAD=MANDATORY
$ export ZE_AFFINITY_MASK=0.0
$ export LIBOMPTARGET_DEBUG=1
$ ./a.out
```

ホストとデバイス間で転送されたデータ量は、`LIBOMPTARGET_DEBUG=1` の出力を `grep "Libomptarget --> Moving"` を実行して確認できます。出力は、配列 `ur`、`us` および `ut` の `map(to:)` 節が、ホストからデバイスへ各配列の 262,144 バイトのの転送が発生していることを示しています。

```
$ grep "Libomptarget --> Moving" test_map_to.debug
Libomptarget --> Moving 262144 bytes (hst:0x00007fffca630880) ->
(tgt:0xff00ffffffff30000)
Libomptarget --> Moving 262144 bytes (hst:0x00007fffca670880) ->
(tgt:0xff00ffffffff70000)
Libomptarget --> Moving 262144 bytes (hst:0x00007fffca6b0880) ->
(tgt:0xff00ffffffffb0000)
Libomptarget --> Moving 2048 bytes (hst:0x00007fffca770880) ->
(tgt:0xff00ffffffffee0000)
Libomptarget --> Moving 262144 bytes (hst:0x00007fffca730880) ->
(tgt:0xff00ffffffffde0000)
Libomptarget --> Moving 262144 bytes (tgt:0xff00ffffffffef0000) ->
(hst:0x00007fffca6f0880)
```

配列 `ur`、`us` および `ut` は、デバイス上で一時的な作業用の配列として使用されるため、これらのデータを転送する意味はありません。そのため、配列の宣言を `declare target` と `end declare target` ディレクティブの間に置くとよいでしょう。これは、配列がデバイスのデータ環境にマップされることを意図しますが、ホストとデバイス間で配列の一貫性を維持するため `target update` ディレクティブが指定されない限り、配列のデータ転送は発生しません。次に、この方法の修正例を示します。

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
```

```
3.  #include <time.h>
4.
5.  #include <math.h>
6.  #include <omp.h>
7.
8.  #define P 16
9.  #define BLOCKS 8
10. #define SIZE (BLOCKS * P * P * P)
11.
12. #define MAX 100
13. #define scaled_rand() ((rand() % MAX) / (1.0 * MAX))
14.
15. #define IDX2(i, j) (i * P + j)
16. #define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)
17.
18. #pragma omp declare target
19. double ur[SIZE], us[SIZE], ut[SIZE]; /* ワーク配列 */
20. #pragma omp end declare target
21.
22. int main(void) {
23.   double w[SIZE];            /* 出力 */
24.   double u[SIZE], dx[P * P]; /* 入力 */
25.   int b, i, j, k, l;         /* ループカウンター */
26.   double start, end;         /* タイマー */
27.
28.   omp_set_default_device(0);
29.
30.   /* 起動時のオーバーヘッドを計測しないよう、ダミーの target 領域を追加しています。*/
31.   #pragma omp target
32.   { ; }
33.
34.   /* 乱数値で入力を初期化します */
35.   srand(0);
36.   for (int i = 0; i < SIZE; i++)
37.     u[i] = scaled_rand();
38.
39.   for (int i = 0; i < P * P; i++)
40.     dx[i] = scaled_rand();
41.
42.   start = omp_get_wtime();
43.
44.   /* カーネルをオフロードします */
45.   #pragma omp target teams distribute parallel for simd simdlen(16) collapse(4) \
46.     map(to:u[0:SIZE],dx[0:P*P]) \
47.     map(from:w[0:SIZE])         \
48.     private(b,i,j,k,l)
49.   for (b = 0; b < BLOCKS; b++) {
50.     for (i = 0; i < P; i++) {
51.       for (j = 0; j < P; j++) {
52.         for (k = 0; k < P; k++) {
53.           w[IDX4(b, i, j, k)] = 0.;
54.           ur[IDX4(b, i, j, k)] = 0.;
55.           us[IDX4(b, i, j, k)] = 0.;
56.           ut[IDX4(b, i, j, k)] = 0.;
57.
58.           for (l = 0; l < P; l++) {
59.             ur[IDX4(b, i, j, k)] += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
60.             us[IDX4(b, i, j, k)] += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
61.             ut[IDX4(b, i, j, k)] += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
62.           }
63.
64.           w[IDX4(b, i, j, k)] = ur[IDX4(b, i, j, k)] * us[IDX4(b, i, j, k)] *
65.                                 ut[IDX4(b, i, j, k)];
66.         }
67.       }
```

```
68.     }
69.   }
70.
71.   end = omp_get_wtime();
72.
73.   /* 結果を出力します */
74.   printf("collapse-clause: w[0]=%lf time=%lf\n", w[0], end - start);
75.
76.   return 0;
77. }
```

上記の修正例では、デバイス上の配列 `ur`、`us` および `ut` のメモリー割り当ては行われますが、データ転送は行われません。これは、`LIBOMPTARGET_DEBUG=1` の出力に、`grep "Libomptarget --> Moving"` コマンドを実行すると分かります。これにより、各配列に対しホストからデバイスへの転送 (262,144 バイト) が見られなくなりました。

```
$ grep "Libomptarget --> Moving" test_map_to.debug
Libomptarget --> Moving 2048 bytes (hst:0x00007ffc546bfec0) ->
(tgt:0xff00ffffffee0000)
Libomptarget --> Moving 262144 bytes (hst:0x00007ffc5467fec0) ->
(tgt:0xff00ffffffff30000)
Libomptarget --> Moving 262144 bytes (tgt:0xff00ffffffef0000) ->
(hst:0x00007ffc5463fec0)
```

ホストとデバイス間でデータ転送を行わずにメモリーを確保する別の例として、次のように `map(to:)` 節の代わりに `map(alloc:)` 節を使用します (45 行から 47 行目)。

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <time.h>
4.
5.  #include <math.h>
6.  #include <omp.h>
7.
8.  #define P 16
9.  #define BLOCKS 8
10. #define SIZE (BLOCKS * P * P * P)
11.
12. #define MAX 100
13. #define scaled_rand() ((rand() % MAX) / (1.0 * MAX))
14.
15. #define IDX2(i, j) (i * P + j)
16. #define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)
17.
18. int main(void) {
19.   double w[SIZE];             /* 出力 */
20.   double u[SIZE], dx[P * P]; /* 入力 */
21.   double ur[SIZE], us[SIZE], ut[SIZE]; /* ワーク配列 */
22.   int b, i, j, k, l;          /* ループカウンター */
23.   double start, end;          /* タイマー */
24.
25.   omp_set_default_device(0);
26.
27.   /* 起動時のオーバーヘッドを計測しないよう、ダミーの target 領域を追加しています。*/
28.   #pragma omp target
29.   { ; }
30.
31.   /* 乱数値で入力を初期化します */
32.   srand(0);
33.   for (int i = 0; i < SIZE; i++)
34.     u[i] = scaled_rand();
35.
36.   for (int i = 0; i < P * P; i++)
```

```
37.     dx[i] = scaled_rand();
38.
39.   start = omp_get_wtime();
40.
41.   /* カーネルをオフロードします */
42.   #pragma omp target teams distribute parallel for simd simdlen(16) collapse(4) \
43.     map(to:u[0:SIZE],dx[0:P*P]) \
44.     map(from:w[0:SIZE])         \
45.     map(alloc:ur[0:SIZE])       \
46.     map(alloc:us[0:SIZE])       \
47.     map(alloc:ut[0:SIZE])       \
48.     private(b,i,j,k,l)
49.   for (b = 0; b < BLOCKS; b++) {
50.     for (i = 0; i < P; i++) {
51.       for (j = 0; j < P; j++) {
52.         for (k = 0; k < P; k++) {
53.           w[IDX4(b, i, j, k)] = 0.;
54.           ur[IDX4(b, i, j, k)] = 0.;
55.           us[IDX4(b, i, j, k)] = 0.;
56.           ut[IDX4(b, i, j, k)] = 0.;
57.
58.           for (l = 0; l < P; l++) {
59.             ur[IDX4(b, i, j, k)] += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
60.             us[IDX4(b, i, j, k)] += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
61.             ut[IDX4(b, i, j, k)] += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
62.           }
63.
64.           w[IDX4(b, i, j, k)] = ur[IDX4(b, i, j, k)] * us[IDX4(b, i, j, k)] *
65.                ut[IDX4(b, i, j, k)];
66.         }
67.       }
68.     }
69.   }
70.
71.   end = omp_get_wtime();
72.
73.   /* 結果を出力します */
74.   printf("collapse-clause: w[0]=%lf time=%lf\n", w[0], end - start);
75.
76.   return 0;
77. }
```

上記の例では、配列 `ur`、`us` および `ut` は、`map(alloc:)` 節によりデバイス上に `ur`、`us` および `ut` が割り当てられますが、データ転送は行われません。`declare target` と `end declare target` を使用した場合と同じです。

```
$ grep "Libomptarget --> Moving" test_map_alloc.debug
Libomptarget --> Moving 2048 bytes (hst:0x00007ffd46f256c0) ->
(tgt:0xff00ffffffee0000)
Libomptarget --> Moving 262144 bytes (hst:0x00007ffd46ee56c0) ->
(tgt:0xff00ffffffde0000)
Libomptarget --> Moving 262144 bytes (tgt:0xff00ffffffef0000) ->
(hst:0x00007ffd46ea56c0)
```

ここで使用する GPU (1 スタックのみ) で実行した各バージョンのパフォーマンスは次のようになります。

```
map(to: ) version                           : 0.001430 seconds
declare target / end declare target version : 0.000874 seconds
map(alloc: ) version                        : 0.000991 seconds
```

## OpenMP* 構造を有効活用する

### nowait を使用して同期を軽減

適切な場合は、target 構造で nowait 節を使用して同期を排除します。

デフォルトでは、target 領域の終わりに暗黙のバリアがあり、target 構造に到達したホストスレッドは、target 領域が完了するまで待機します。

target 構造に nowait 節を追加すると暗黙のバリアがなくなり、target 構造に到達したホストスレッドは、target 領域が完了を待たずに処理を続行します。これにより、ホストスレッドは target 領域の完了を待機しないため、target 領域はデバイス上で非同期に実行できます。

次の例では、2 つの行列 v1 と v2 の積を parallel 領域で計算するケースを考えてみます (45 行目)。計算の半分は、ホスト上で parallel 領域を実行するスレッドのチームで行われます。残りの半分の計算は、デバイス上で実行されます。チームのプライマリー・スレッドは、デバイス上で計算を行うため target 領域を起動します。

デフォルトでは、チームのプライマリー・スレッドは、target 領域の完了を待機してから、ホストでの計算 (for ループのワークシェア) を続行します。

```
1.  /*
2.   * This test is taken from OpenMP API 5.0.1 Examples (June 2020)
3.   * https://www.openmp.org/wp-content/uploads/openmp-examples-5-0-1.pdf
4.   * (4.13.2 nowait Clause on target Construct)
5.   */
6.
7.
8.  #include <stdio.h>
9.  #include <stdlib.h>
10. #include <time.h>
11. #include <omp.h>
12.
13. #define N 100000 // N は偶数
14.
15. void init(int n, float *v1, float *v2) {
16.   int i;
17.
18.   for(i=0; i<n; i++){
19.     v1[i] = i * 0.25;
20.     v2[i] = i - 1.25;
21.   }
22. }
23.
24. int main() {
25.   int i, n=N;
26.   float v1[N],v2[N],vxv[N];
27.   double start,end; // timers
28.
29.   init(n, v1, v2);
30.
31.   /* 開始時間を測定しないよう、ダミーの parallel と target 領域を
32.      追加します。*/
33.   #pragma omp parallel
34.   {
35.      #pragma omp master
36.      #pragma omp target
37.        {;}
38.   }
39.
```

```
40.   start=omp_get_wtime();
41.
42.   #pragma omp parallel
43.   {
44.      #pragma omp master
45.      #pragma omp target teams distribute parallel for        \
46.        map(to: v1[0:n/2])                                    \
47.        map(to: v2[0:n/2])                                    \
48.        map(from: vxv[0:n/2])
49.      for(i=0; i<n/2; i++){
50.        vxv[i] = v1[i]*v2[i];
51.      }
52.      /* プライマリー・スレッドは、このポイントの先に進む前に
53.         target 領域の完了を待機します。*/
54.
55.      #pragma omp for
56.      for(i=n/2; i<n; i++) {
57.        vxv[i] = v1[i]*v2[i];
58.      }
59.      /* ワークシェアリング for の終わりには暗黙の同期があります */
60.   }
61.
62.   end=omp_get_wtime();
63.
64.   printf("vxv[0]=%f, vxv[n-1]=%f, time=%lf\n", vxv[0], vxv[n-1], end-start);
65.   return 0;
66. }
```

**コンパイルコマンド**

```
$ icpx -fiopenmp -fopenmp-targets=spir64 test_target_no_nowait.cpp
```

**実行コマンド**

```
$ export OMP_TARGET_OFFLOAD=MANDATORY
$ export ZE_AFFINITY_MASK=0.0
$ export LIBOMPTARGET_DEBUG=1
$ ./a.out
```

`nowait` 節を `target` 構造に追加すると、プライマリー・スレッドは `target` 領域の完了を待つことなく、`for` ループのワークシェリングを開始できるため、パフォーマンスが向上する可能性があります。`for` ループのワークシェリングの最後に暗黙のバリアがあるため、`target` 領域との同期が保証されます。

```
1.  #include <stdlib.h>
2.  #include <time.h>
3.  #include <omp.h>
4.
5.  #define N 100000 // N は偶数
6.
7.  void init(int n, float *v1, float *v2) {
8.    int i;
9.
10.   for(i=0; i<n; i++){
11.     v1[i] = i * 0.25;
12.     v2[i] = i - 1.25;
13.   }
14. }
15.
16. int main() {
17.   int i, n=N;
18.   float v1[N],v2[N],vxv[N];
19.   double start,end; // タイマー
20.
```

```
21.   init(n, v1,v2);
22.
23.   /* 起動のオーバーヘッドを計測しないように、ダミーの parallel および target (nowait) 領域を挿入
24.      */
25.   #pragma omp parallel
26.   {
27.      #pragma omp master
28.      #pragma omp target nowait
29.        {;}
30.   }
31.
32.   start=omp_get_wtime();
33.
34.   #pragma omp parallel
35.   {
36.      #pragma omp master
37.      #pragma omp target teams distribute parallel for nowait \
38.        map(to: v1[0:n/2])                                    \
39.        map(to: v2[0:n/2])                                    \
40.        map(from: vxv[0:n/2])
41.      for(i=0; i<n/2; i++){
42.        vxv[i] = v1[i]*v2[i];
43.      }
44.
45.      #pragma omp for
46.      for(i=n/2; i<n; i++) {
47.        vxv[i] = v1[i]*v2[i];
48.      }
49.      /* ワークシェアリング for の終わりには暗黙の同期があります。Target 領域は、この時点までに
50.         完了することが保証されます */
51.   }
52.
53.   end=omp_get_wtime();
54.
55.   printf("vxv[1]=%f, vxv[n-1]=%f, time=%lf\n", vxv[1], vxv[n-1], end-start);
56.   return 0;
57. }
```

ラボマシンで実行した 2 つのバージョンのパフォーマンスは次のようになりました。

```
no nowait version        : 0.008220 seconds
nowait on target version : 0.002110 seconds
```

**Fortran**

上記と同じ `nowait` の例を Fortran で記述すると次のようになります。

```
1.        !
2.        ! This test is from OpenMP API 5.0.1 Examples (June 2020)
3.        ! https://www.openmp.org/wp-content/uploads/openmp-examples-5-0-1.pdf
4.        !(4.13.2 nowait Clause on target Construct)
5.        !
6.
7.        subroutine init(n, v1, v2)
8.        integer :: i, n
9.        real :: v1(n), v2(n)
10.
11.       do i = 1, n
12.          v1(i) = i * 0.25
13.          v2(i) = i - 1.25
14.       end do
15.       end subroutine init
16.
17.       program test_target_nowait
18.       use omp_lib
```

```
19.       use iso_fortran_env
20.       implicit none
21.
22.       integer, parameter :: NUM=100000 ! NUM は偶数でなければなりません
23.       real :: v1(NUM), v2(NUM), vxv(NUM)
24.       integer :: n, i
25.       real(kind=REAL64) :: start, end
26.
27.       n = NUM
28.       call init(n, v1, v2)
29.
30.       ! 起動のオーバーヘッドを計測しないように、ダミーの parallel および target (nowait) 領域を挿入
31.       !
32.       !$omp parallel
33.         !$omp master
34.           !$omp target nowait
35.           !$omp end target
36.         !$omp end master
37.       !$omp end parallel
38.
39.       start=omp_get_wtime()
40.
41.       !$omp parallel
42.
43.         !$omp master
44.           !$omp target teams distribute parallel do nowait &
45.           !$omp& map(to: v1(1:n/2)) &
46.           !$omp& map(to: v2(1:n/2)) &
47.           !$omp& map(from: vxv(1:n/2))
48.           do i = 1, n/2
49.              vxv(i) = v1(i)*v2(i)
50.           end do
51.         !$omp end master
52.
53.         !$omp do
54.         do i = n/2+1, n
55.            vxv(i) = v1(i)*v2(i)
56.         end do
57.
58.       !$omp end parallel
59.
60.       end=omp_get_wtime()
61.
62.       write(*,110) "vxv(1)=", vxv(1), ", vxv(n-1)=", vxv(n-1), ", time=", end-start
63. 110   format (A, F10.6, A, F17.6, A, F10.6)
64.
65.       end program test_target_nowait
```

## メモリー割り当て

ここでは、いくつかのメモリー割り当て方法と、サポートされる割り当てタイプについて説明します。ホスト上のポインターは、デバイス上のポインターと同じサイズです。

**ホスト割り当て**は、ホストが所有し、システムメモリーから割り当てることを意図しています。ホスト割り当ては、ホストとサポートされるすべてのデバイスでアクセスできます。そのため、ホスト割り当てのポインターは、ホストとサポートされるデバイスで同じポインターを使用できます。ホスト割り当てでは、システムメモリーとデバイスのローカルメモリー間での移行は想定されていません。デバイス上でホスト割り当てのポインターがアクセスされると、データは通常、デバイスとホストを接続する PCIe* などのバスを介して転送されます。

**デバイス割り当て**は、デバイスが所有し、デバイス・ローカル・メモリー外に割り当てることを意図しています。割り当てられたストレージは、該当するデバイスからリード/ライトできますが、ホストや他のサポートされるデバイスからは直接アクセスできません。

**共有割り当て**は、ホストとサポートされるすべてのデバイスでアクセスできます。そのため、ホスト割り当てのように、ホストとサポートされるすべてのデバイスは、共有割り当てメモリーを指す同じポインターを使用できます。ただし、共有割り当ては、特定のデバイスが所有するものではなく、ホストと複数のデバイス間での移行を意図するものです。移行後のデバイスでのアクセスは、レイテンシーが大きいバスを介したシステムメモリーのアクセスではなく、高速なデバイス・ローカル・メモリーであることを意味します。

**共有システム割り当て**は、共有割り当てのサブクラスであり、メモリーはシステム・アロケーター (`malloc` や `new` など) ではなく、割り当て API (OpenMP* のメモリー割り当て API) を使用します。共有システム割り当ては関連するデバイスを持たず、本質的にクロスデバイスです。ほかの共有割り当てと同様に、共有システム割り当ては、ホストとサポートされるデバイス間の移行を意図しているため、共有システム割り当てへの同じポインターがホストとすべてのデバイス間で使用される可能性があります。

**注:**

- 現在、インテル® データセンター GPU マックス・シリーズのシステムでは共有システム割り当てはサポートされていません。ただし、割り当て API によって割り当てられる、共有割り当てはサポートされます。

次の表は、各タイプのメモリー割り当て特性をまとめたものです。

| 割り当てタイプ | 初期の場所 | ホストでアクセス可能 | デバイスでアクセス可能 |
|---|---|---|---|
| ホスト | ホスト | はい | はい |
| デバイス | デバイス | いいえ | はい |
| 共有 | ホスト、デバイス、または未指定 | はい | はい |
| 共有システム | ホスト | はい | はい |

ホスト割り当ては広範囲なアクセス (ホストおよびサポートされるすべてのデバイスからの直接アクセス) を提供しますが、一般的にデータは PCIe* などのバスを介して転送されるためアクセスのコストは高くなります。

共有割り当ても幅広いアクセスを提供しますが、アクセスしたデバイスにデータが移行されるため、アクセスごとのコストはホスト割り当てよりも低くなる可能性があります。

デバイス割り当てにはアクセス制限 (ホストや他のサポートされるデバイスから直接アクセスできない) がありますが、デバイスのローカルメモリーにアクセスできるため、高いパフォーマンスを発揮します。

## メモリー割り当ての OpenMP* ランタイムルーチン

インテル® コンパイラーは、メモリー割り当てを行う多くの OpenMP* ランタイムルーチンをサポートしています。それらのルーチンを次の表に示します。

| OpenMP* メモリー割り当てルーチン | インテル拡張 | 割り当てタイプ |
|---|---|---|
| `omp_target_alloc` | いいえ | デバイス |
| `omp_target_alloc_device` | はい | デバイス |
| `omp_target_alloc_host` | はい | ホスト |
| `omp_target_alloc_shared` | はい | 共有 |

`omp_target_alloc_device`、`omp_target_alloc_host` および `omp_target_alloc_shared` の 3 つのルーチンは、インテルによる OpenMP* 仕様の拡張であることに注意してください。

次の例では、上記の OpenMP* メモリー割り当てルーチンを使用しています。`map` 節を使用した場合と比較します。

メモリー割り当ての詳細については、以下を参照してください。

- 『Data Parallel C++』 James Reinders 共著 (英語)
- SYCL 2020 仕様 リビジョン 6 日本語版
- oneAPI ゼロレベル仕様 (英語)
- このドキュメントの SYCL* セクション

## map 節を使用する

最初の例では、`map` 節を使用してデバイス上にメモリーを割り当てて、ホストとデバイス間でデータをコピーしています。

次の例では、配列 `A`、`B`、`C` は、C/C++ の標準ライブラリー・ルーチン `malloc` によってシステムメモリーに割り当てられています。

52 行目の `target` 構造はデバイス上で配列 C の値を計算するカーネルです。この `target` 構造では、計算前にホストからデバイスへ、そして計算後にデバイスからホストへ `C` の値を転送する必要があるため、`map(tofrom: C[0:length)` 節が指定されています。配列 `A` と `B` の値は、ホストからデバイスに転送する必要があり、デバイスではこれらの値を読み取るだけなので `map(to: A[0:length], B[0:length])` が指定されています。そして、`map` 節は、配列のストレージをデバイス上に割り当て、データをホストからデバイスへ、そしてデバイスからホストへコピーすることを指示しています。

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <stdint.h>
4.  #include <math.h>
5.  #include <omp.h>
6.
7.  #define iterations 100
8.  #define length     64*1024*1024
9.
10. int main(void)
11. {
12.   size_t bytes = length*sizeof(double);
13.   double * __restrict A;
14.   double * __restrict B;
15.   double * __restrict C;
16.   double scalar = 3.0;
17.   double nstream_time = 0.0;
18.
19.   // malloc() を使用してホスト上に配列を割り当てます
20.
21.   A = (double *) malloc(bytes);
22.   if (A == NULL){
23.      printf(" ERROR: Cannot allocate space for A using plain malloc().\n");
24.      exit(1);
25.   }
26.
27.   B = (double *) malloc(bytes);
28.   if (B == NULL){
29.      printf(" ERROR: Cannot allocate space for B using plain malloc().\n");
30.      exit(1);
31.   }
32.
33.   C = (double *) malloc(bytes);
34.   if (C == NULL){
35.      printf(" ERROR: Cannot allocate space for C using plain malloc().\n");
```

```
36.       exit(1);
37.    }
38.
39.    // 配列を初期化
40.
41.    #pragma omp parallel for
42.    for (size_t i=0; i<length; i++) {
43.        A[i] = 2.0;
44.        B[i] = 2.0;
45.        C[i] = 0.0;
46.    }
47.
48.    // 計算を実行
49.
50.    nstream_time = omp_get_wtime();
51.    for (int iter = 0; iter<iterations; iter++) {
52.        #pragma omp target teams distribute parallel for \
53.          map(to: A[0:length], B[0:length]) \
54.          map(tofrom: C[0:length])
55.        for (size_t i=0; i<length; i++) {
56.            C[i] += A[i] + scalar * B[i];
57.        }
58.    }
59.    nstream_time = omp_get_wtime() - nstream_time;
60.
61.    // 結果を評価して出力
62.
63.    double ar = 2.0;
64.    double br = 2.0;
65.    double cr = 0.0;
66.    for (int iter = 0; iter<iterations; iter++) {
67.        for (int i=0; i<length; i++) {
68.            cr += ar + scalar * br;
69.        }
70.    }
71.
72.    double asum = 0.0;
73.    #pragma omp parallel for reduction(+:asum)
74.    for (size_t i=0; i<length; i++) {
75.        asum += fabs(C[i]);
76.    }
77.
78.    free(A);
79.    free(B);
80.    free(C);
81.
82.    double epsilon=1.e-8;
83.    if (fabs(cr - asum)/asum > epsilon) {
84.        printf("Failed Validation on output array\n"
85.               "       Expected checksum: %lf\n"
86.               "       Observed checksum: %lf\n"
87.               "ERROR: solution did not validate\n", cr, asum);
88.        return 1;
89.    } else {
90.        printf("Solution validates\n");
91.        double avgtime = nstream_time/iterations;
92.        printf("Checksum = %lf; Avg time (s): %lf\n", asum, avgtime);
93.    }
94.
95.    return 0;
96. }
```

**コンパイルコマンド**

```
$ icpx -fiopenmp -fopenmp-targets=spir64 test_target_map.cpp
```

**実行コマンド**

```
$ export OMP_TARGET_OFFLOAD=MANDATORY
$ export ZE_AFFINITY_MASK=0.0
$ export LIBOMPTARGET_DEBUG=1
$ ./a.out
```

反復ループ内の `target` 構造の `map` 節は、それぞれの `target` 領域の入り口でデータ (`A`、`B`、`C` の値) をホストからデバイスに転送し、各 `target` 領域の出口でデータ (`C` の値) をデバイスからホストに転送します。これらのデータ転送には重大なパフォーマンス上のオーバーヘッドが発生します。`map` 節を使用する最善のアプローチは、反復ループ全体を `map` 節が関連する `target data` 構造に含むことです。これにより、転送は反復ループの入り口で一度、そして反復ループの出口で一度行われます。`target data` と `map` 節を使用した修正例を以下に示します。

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <stdint.h>
4.  #include <math.h>
5.  #include <omp.h>
6.
7.  #define iterations 100
8.  #define length     64*1024*1024
9.
10. int main(void)
11. {
12.   size_t bytes = length*sizeof(double);
13.   double * __restrict A;
14.   double * __restrict B;
15.   double * __restrict C;
16.   double scalar = 3.0;
17.   double nstream_time = 0.0;
18.
19.   // malloc() を使用してホスト上に配列を割り当てます
20.
21.   A = (double *) malloc(bytes);
22.   if (A == NULL){
23.      printf(" ERROR: Cannot allocate space for A using plain malloc().\n");
24.      exit(1);
25.   }
26.
27.   B = (double *) malloc(bytes);
28.   if (B == NULL){
29.      printf(" ERROR: Cannot allocate space for B using plain malloc().\n");
30.      exit(1);
31.   }
32.
33.   C = (double *) malloc(bytes);
34.   if (C == NULL){
35.      printf(" ERROR: Cannot allocate space for C using plain malloc().\n");
36.      exit(1);
37.   }
38.
39.   // 配列を初期化
40.
41.   #pragma omp parallel for
42.   for (size_t i=0; i<length; i++) {
43.       A[i] = 2.0;
44.       B[i] = 2.0;
45.       C[i] = 0.0;
46.   }
47.
48.   // 計算を実行
49.
50.   nstream_time = omp_get_wtime();
51.   #pragma omp target data  map(to: A[0:length], B[0:length]) \
```

```
52.     map(tofrom: C[0:length])
53.   {
54.      for (int iter = 0; iter<iterations; iter++) {
55.          #pragma omp target teams distribute parallel for
56.          for (size_t i=0; i<length; i++) {
57.              C[i] += A[i] + scalar * B[i];
58.          }
59.      }
60.   }
61.   nstream_time = omp_get_wtime() - nstream_time;
62.
63.   // 結果を評価して出力
64.
65.   double ar = 2.0;
66.   double br = 2.0;
67.   double cr = 0.0;
68.   for (int iter = 0; iter<iterations; iter++) {
69.       for (int i=0; i<length; i++) {
70.           cr += ar + scalar * br;
71.       }
72.   }
73.
74.   double asum = 0.0;
75.   #pragma omp parallel for reduction(+:asum)
76.   for (size_t i=0; i<length; i++) {
77.       asum += fabs(C[i]);
78.   }
79.
80.   free(A);
81.   free(B);
82.   free(C);
83.
84.   double epsilon=1.e-8;
85.   if (fabs(cr - asum)/asum > epsilon) {
86.       printf("Failed Validation on output array\n"
87.              "       Expected checksum: %lf\n"
88.              "       Observed checksum: %lf\n"
89.              "ERROR: solution did not validate\n", cr, asum);
90.       return 1;
91.   } else {
92.       printf("Solution validates\n");
93.       double avgtime = nstream_time/iterations;
94.       printf("Checksum = %lf; Avg time (s): %lf\n", asum, avgtime);
95.   }
96.
97.   return 0;
98. }
```

## omp_target_alloc

次に、上記の例を修正して `map` 節の代わりにデバイス割り当てを使用します。配列 `A`、`B`、および `C` のストレージは、OpenMP* ランタイムルーチン `omp_target_alloc` を呼び出してデバイス上に直接割り当てられます。このルーチンは、デバイス上に割り当てるバイト数と、ストレージを割り当てるデバイス番号の 2 つの引数を受け取り、デバイス上に割り当てられたストレージのデバイスアドレスを参照するデバイスポインターを返します。`omp_target_alloc` が NULL を返す場合、割り当てが成功しなかったことを意味します。

`target` 構造に割り当てられたメモリーにアクセスするには、`omp_target_alloc` を呼び出して取得したデバイスポインターを `target` 構造の `is_device_ptr` 節に指定します。これにより、カーネルはデバイス上に存在するデータを操作するため、カーネル実行の前後でデータ転送が発生しないようになります。

プログラムの最後でランタイムルーチン `omp_target_free` を呼び出して、`A`、`B`、`C` のストレージ割り当てを解放します。

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <math.h>
#include <omp.h>

#define iterations 100
#define length     64*1024*1024

int main(void)
{
  int device_id = omp_get_default_device();
  size_t bytes = length*sizeof(double);
  double * __restrict A;
  double * __restrict B;
  double * __restrict C;
  double scalar = 3.0;
  double nstream_time = 0.0;

  // デバイスメモリーに配列を割り当て

  A = (double *) omp_target_alloc(bytes, device_id);
  if (A == NULL){
     printf(" ERROR: Cannot allocate space for A using omp_target_alloc().\n");
     exit(1);
  }

  B = (double *) omp_target_alloc(bytes, device_id);
  if (B == NULL){
     printf(" ERROR: Cannot allocate space for B using omp_target_alloc().\n");
     exit(1);
  }

  C = (double *) omp_target_alloc(bytes, device_id);
  if (C == NULL){
     printf(" ERROR: Cannot allocate space for C using omp_target_alloc().\n");
     exit(1);
  }

  // 配列を初期化

  #pragma omp target teams distribute parallel for \
    is_device_ptr(A,B,C)
  for (size_t i=0; i<length; i++) {
      A[i] = 2.0;
      B[i] = 2.0;
      C[i] = 0.0;
  }

  // 計算を iterations 回繰り返します

  nstream_time = omp_get_wtime();
  for (int iter = 0; iter<iterations; iter++) {
      #pragma omp target teams distribute parallel for \
         is_device_ptr(A,B,C)
      for (size_t i=0; i<length; i++) {
          C[i] += A[i] + scalar * B[i];
      }
  }
  nstream_time = omp_get_wtime() - nstream_time;

  // 結果を評価して出力

  double ar = 2.0;
  double br = 2.0;
  double cr = 0.0;
  for (int iter = 0; iter<iterations; iter++) {
```

```
68.       for (int i=0; i<length; i++) {
69.           cr += ar + scalar * br;
70.       }
71.   }
72.
73.   double asum = 0.0;
74.   #pragma omp target teams distribute parallel for reduction(+:asum) \
75.     map(tofrom: asum) is_device_ptr(C)
76.   for (size_t i=0; i<length; i++) {
77.       asum += fabs(C[i]);
78.   }
79.
80.   omp_target_free(A, device_id);
81.   omp_target_free(B, device_id);
82.   omp_target_free(C, device_id);
83.
84.   double epsilon=1.e-8;
85.   if (fabs(cr - asum)/asum > epsilon) {
86.       printf("Failed Validation on output array\n"
87.              "       Expected checksum: %lf\n"
88.              "       Observed checksum: %lf\n"
89.              "ERROR: solution did not validate\n", cr, asum);
90.       return 1;
91.   } else {
92.       printf("Solution validates\n");
93.       double avgtime = nstream_time/iterations;
94.       printf("Checksum = %lf; Avg time (s): %lf\n", asum, avgtime);
95.   }
96.
97.   return 0;
98. }
```

**注:**

- `omp_target_alloc` を呼び出す場合、指定するデバイス番号はホストデバイス以外のサポートされるデバイスでなければなりません。これがストレージを割り当てるデバイスとなります。
- 配列 `A`、`B`、`C` はホストからアクセスできないため、配列の初期化、カーネル実行、および `C` の要素の合計はすべて OpenMP* `target` 領域内で行う必要があります。
- デバイス割り当ては `omp_target_alloc` 呼び出しで指定されたデバイスからのみアクセスできますが、`omp_target_memcpy` を呼び出してホストまたは他のデバイスに割り当てられたメモリーにコピーできます。

## omp_target_alloc_device

インテル拡張機能である `omp_target_alloc_device` は、`omp_target_alloc` に類似しており、デバイス上に割り当てるバイト数と、ストレージを割り当てるデバイス番号の 2 つの引数を受け取ります。デバイス上に割り当てられたストレージのデバイスアドレスを参照するデバイスポインターを返します。`omp_target_alloc_device` が NULL を返す場合、割り当てが成功しなかったことを意味します。

上記の `omp_target_alloc` の例は、次のように `omp_target_alloc` 呼び出しを `omp_target_alloc` 呼び出しに置き換えるだけで書き換えできます。

プログラムの最後でランタイムルーチン omp_target_free を呼び出して、A、B、C のストレージ割り当てを解放します。

```
// デバイスメモリーに配列を割り当て

  A = (double *) omp_target_alloc_device(bytes, device_id);
  if (A == NULL){
     printf(" ERROR: Cannot allocate space for A using omp_target_alloc_device().\n");
     exit(1);
  }

  B = (double *) omp_target_alloc_device(bytes, device_id);
  if (B == NULL){
     printf(" ERROR: Cannot allocate space for B using omp_target_alloc_device().\n");
     exit(1);
  }

  C = (double *) omp_target_alloc_device(bytes, device_id);
  if (C == NULL){
     printf(" ERROR: Cannot allocate space for C using omp_target_alloc_device().\n");
     exit(1);
  }
```

**注:**

- omp_target_alloc に適用される注意事項は、omp_target_alloc_device にも同様に適用されます。

## omp_target_alloc_host

上記の例は、A、B、C をホストに割り当てるように書き換えることもできます。これにより、ホストとサポートされるすべてのデバイスからメモリーにアクセスできるようになります。

以下の修正例では、omp_target_alloc_host ランタイムルーチン (インテル拡張機能) を呼び出して、配列 A、B、および C にストレージを割り当てています。このルーチンは、割り当てバイト数とデバイス番号を引数として受け取ります。デバイス番号はホストデバイス以外のサポートされるデバイスでなければなりません。このルーチンは、ホストメモリーの格納場所へのポインターを返します。omp_target_alloc_host が NULL を返す場合、割り当てが成功しなかったことを意味します。

プログラムの最初に unified_address ディレクティブが指定されていることに注意してください。これには、OpenMP* API ルーチンとディレクティブを介してアクセス可能なすべてのデバイスが、統合アドレス空間を使用できる実装を保証する必要があります。このアドレス空間では、ポインターは常にすべてのデバイスで同じメモリー位置を参照します。ターゲット領域内で使用するためデバイスポインターからデバイスアドレスを取得する is_device_ptr 節は必要ありません。インテル® コンパイラーを使用するとデフォルトで統合アドレス空間が保障されているため、unified_address ディレクティブは必要ありませんが、移植性のためディレクティブを残しておくと良いでしょう。

omp_target_alloc_host 呼び出しが返すポインターを使用して、ホストとサポートされるすべてのデバイスからストレージにアクセスできます。統合アドレス空間が使用されているので、デバイスからメモリーにアクセスするため、target 構造に map 節や is_device_ptr 節は必要ありません。

プログラムの最後でランタイムルーチン omp_target_free を呼び出して、A、B、C のストレージ割り当てを解放します。

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <math.h>
#include <omp.h>

#pragma omp requires unified_address

#define iterations 100
#define length     64*1024*1024

int main(void)
{
  int device_id = omp_get_default_device();
  size_t bytes = length*sizeof(double);
  double * __restrict A;
  double * __restrict B;
  double * __restrict C;
  double scalar = 3.0;
  double nstream_time = 0.0;

  // ホストメモリーに配列を割り当て

  A = (double *) omp_target_alloc_host(bytes, device_id);
  if (A == NULL){
     printf(" ERROR: Cannot allocate space for A using omp_target_alloc_host().\n");
     exit(1);
  }

  B = (double *) omp_target_alloc_host(bytes, device_id);
  if (B == NULL){
     printf(" ERROR: Cannot allocate space for B using omp_target_alloc_host().\n");
     exit(1);
  }

  C = (double *) omp_target_alloc_host(bytes, device_id);
  if (C == NULL){
     printf(" ERROR: Cannot allocate space for C using omp_target_alloc_host().\n");
     exit(1);
  }

  // 配列を初期化

  #pragma omp parallel for
  for (size_t i=0; i<length; i++) {
      A[i] = 2.0;
      B[i] = 2.0;
      C[i] = 0.0;
  }

  // 計算を実行

  nstream_time = omp_get_wtime();
  for (int iter = 0; iter<iterations; iter++) {
      #pragma omp target teams distribute parallel for
      for (size_t i=0; i<length; i++) {
          C[i] += A[i] + scalar * B[i];
      }
  }
  nstream_time = omp_get_wtime() - nstream_time;

  // 結果を評価して出力

  double ar = 2.0;
  double br = 2.0;
  double cr = 0.0;
  for (int iter = 0; iter<iterations; iter++) {
```

```
68.       for (int i=0; i<length; i++) {
69.           cr += ar + scalar * br;
70.       }
71.   }
72.
73.   double asum = 0.0;
74.   #pragma omp parallel for reduction(+:asum)
75.   for (size_t i=0; i<length; i++) {
76.       asum += fabs(C[i]);
77.   }
78.
79.   omp_target_free(A, device_id);
80.   omp_target_free(B, device_id);
81.   omp_target_free(C, device_id);
82.
83.   double epsilon=1.e-8;
84.   if (fabs(cr - asum)/asum > epsilon) {
85.       printf("Failed Validation on output array\n"
86.              "       Expected checksum: %lf\n"
87.              "       Observed checksum: %lf\n"
88.              "ERROR: solution did not validate\n", cr, asum);
89.       return 1;
90.   } else {
91.       printf("Solution validates\n");
92.       double avgtime = nstream_time/iterations;
93.       printf("Checksum = %lf; Avg time (s): %lf\n", asum, avgtime);
94.   }
95.
96.   return 0;
97. }
```

**注:**

- `omp_target_alloc_host` を呼び出す場合、指定するデバイス番号はホストデバイス以外のサポートされるデバイスでなければなりません。
- 配列 `A`、`B` および `C` はホストとデバイスからアクセスできるため、配列の初期化と `C` の要素の合計は、ホスト (`target` 構造の外で) またはデバイス (`target` 構造の中) のどちらでも実行できます。
- インテル® データセンター GPU マックス・シリーズは、ホスト割り当て (`omp_target_alloc_host` で割り当てられたメモリー) のアトミック操作 (または、一部のリダクションなどアトミック操作を行うアルゴリズム) をサポートしていません。代わりに、`omp_target_alloc_device` で割り当てたメモリーでアトミック操作を行います。

## omp_target_alloc_shared

上記の例では、ホスト割り当ての代わりに共有割り当てを使用するように変更されています。`omp_target_alloc_shared` ランタイムルーチンは、配列 `A`、`B`、および `C` にストレージを割り当てています。このルーチンは、割り当てバイト数とデバイス番号を引数として受け取ります。デバイス番号はホストデバイス以外のサポートされるデバイスでなければなりません。このルーチンは、共有メモリーの格納場所へのポインターを返します。`omp_target_alloc_shared` が NULL を返すと、割り当てが成功しなかったことを意味します。

移植性のため、プログラムの最初に `unified_address` ディレクティブが指定されていることに注意してください。

`omp_target_alloc_shared` 呼び出しが返すポインターを使用して、ホストとサポートされるすべてのデバイスからストレージにアクセスできます。統合アドレス空間が使用されているので、デバイスからメモリーをアクセスするため、`target` 構造に `map` 節や `is_device_ptr` 節は必要ありません。

プログラムの最後でランタイムルーチン `omp_target_free` を呼び出して、`A`、`B`、`C` のストレージ割り当てを解放します。

```
// 共有メモリーに配列を割り当て

  A = (double *) omp_target_alloc_shared(bytes, device_id);
  if (A == NULL){
     printf(" ERROR: Cannot allocate space for A using omp_target_alloc_shared().\n");
     exit(1);
  }

  B = (double *) omp_target_alloc_shared(bytes, device_id);
  if (B == NULL){
     printf(" ERROR: Cannot allocate space for B using omp_target_alloc_shared().\n");
     exit(1);
  }

  C = (double *) omp_target_alloc_shared(bytes, device_id);
  if (C == NULL){
     printf(" ERROR: Cannot allocate space for C using omp_target_alloc_shared().\n");
     exit(1);
  }
```

**注:**

- `omp_target_alloc_shared` を呼び出す場合、指定するデバイス番号はホストデバイス以外のサポートされるデバイスでなければなりません。
- 配列はホストとデバイスからアクセスできるため、初期化と検証はホストまたはデバイス (`target` 構造内) のどちらでも実行できます。
- `omp_target_alloc_shared` で割り当てたメモリーへのホストとデバイスからの同時アクセスはサポートされていません。

### omp_target_memcpy

次の例は、ランタイムルーチン `omp_target_memcpy` を使用して、ホストからデバイスへ、またはデバイスからホストへメモリーをコピーする方法を示しています。最初の配列 `h_A`、`h_B` および `h_C` は、通常の `malloc` でシステムメモリーに割り当てられてから初期化されています。対応する配列 `d_A`、`d_B` および `d_C` は、`omp_target_alloc` によってデバイスに割り当てられています。

98 行目で、`target` 構造を開始する前に、`h_A`、`h_B`、そして `h_C` の値が `omp_target_memcpy` を呼び出して `d_A`、`d_B`、`d_C` にコピーされます。`target` 領域の後で、`omp_target_memcpy` を呼び出してデバイスで計算された新しい `d_C` の値が `h_C` にコピーされます。

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <stdint.h>
4.  #include <math.h>
5.  #include <omp.h>
6.
7.  #define iterations 100
8.  #define length     64*1024*1024
9.
10. int main(void)
11. {
12.   int device_id = omp_get_default_device();
13.   int host_id = omp_get_initial_device();
14.   size_t bytes = length*sizeof(double);
15.   double * __restrict h_A;
16.   double * __restrict h_B;
17.   double * __restrict h_C;
18.   double * __restrict d_A;
19.   double * __restrict d_B;
20.   double * __restrict d_C;
21.   double scalar = 3.0;
```

```
22.   double nstream_time = 0.0;
23.
24.   // 通常の malloc() を使用して、配列 h_A、h_B、および h_C をホストに割り当てます
25.
26.   h_A = (double *) malloc(bytes);
27.   if (h_A == NULL){
28.     printf(" ERROR: Cannot allocate space for h_A using plain malloc().\n");
29.     exit(1);
30.   }
31.
32.   h_B = (double *) malloc(bytes);
33.   if (h_B == NULL){
34.     printf(" ERROR: Cannot allocate space for h_B using plain malloc().\n");
35.     exit(1);
36.   }
37.
38.   h_C = (double *) malloc(bytes);
39.   if (h_C == NULL){
40.     printf(" ERROR: Cannot allocate space for h_C using plain malloc().\n");
41.     exit(1);
42.   }
43.
44.   // omp_target_alloc() を使用して、配列 d_A、d_B、および d_C をデバイスに割り当てます
45.
46.   d_A = (double *) omp_target_alloc(bytes, device_id);
47.   if (d_A == NULL){
48.     printf(" ERROR: Cannot allocate space for d_A using omp_target_alloc().\n");
49.     exit(1);
50.   }
51.
52.   d_B = (double *) omp_target_alloc(bytes, device_id);
53.   if (d_B == NULL){
54.     printf(" ERROR: Cannot allocate space for d_B using omp_target_alloc().\n");
55.     exit(1);
56.   }
57.
58.   d_C = (double *) omp_target_alloc(bytes, device_id);
59.   if (d_C == NULL){
60.     printf(" ERROR: Cannot allocate space for d_C using omp_target_alloc().\n");
61.     exit(1);
62.   }
63.
64.   // ホスト上で配列を初期化します
65.
66.   #pragma omp parallel for
67.   for (size_t i=0; i<length; i++) {
68.     h_A[i] = 2.0;
69.     h_B[i] = 2.0;
70.     h_C[i] = 0.0;
71.   }
72.
73.   // omp_target_memcpy() を呼び出して、ホストからデバイスへ値をコピーします
74.
75.   int rc = 0;
76.   rc = omp_target_memcpy(d_A, h_A, bytes, 0, 0, device_id, host_id);
77.   if (rc) {
78.     printf("ERROR: omp_target_memcpy(A) returned %d\n", rc);
79.     exit(1);
80.   }
81.
82.   rc = omp_target_memcpy(d_B, h_B, bytes, 0, 0, device_id, host_id);
83.   if (rc) {
84.     printf("ERROR: omp_target_memcpy(B) returned %d\n", rc);
85.     exit(1);
86.   }
87.
88.   rc = omp_target_memcpy(d_C, h_C, bytes, 0, 0, device_id, host_id);
```

```
  if (rc) {
    printf("ERROR: omp_target_memcpy(C) returned %d\n", rc);
    exit(1);
  }

  // 計算を実行

  nstream_time = omp_get_wtime();
  for (int iter = 0; iter<iterations; iter++) {
    #pragma omp target teams distribute parallel for \
      is_device_ptr(d_A,d_B,d_C)
    for (size_t i=0; i<length; i++) {
      d_C[i] += d_A[i] + scalar * d_B[i];
    }
  }
  nstream_time = omp_get_wtime() - nstream_time;

  // omp_target_memcpy() を呼び出して、デバイスからホストへ値をコピーします

  rc = omp_target_memcpy(h_C, d_C, bytes, 0, 0, host_id, device_id);
  if (rc) {
    printf("ERROR: omp_target_memcpy(A) returned %d\n", rc);
    exit(1);
  }

  // 結果を評価して出力

  double ar = 2.0;
  double br = 2.0;
  double cr = 0.0;
  for (int iter = 0; iter<iterations; iter++) {
    for (int i=0; i<length; i++) {
      cr += ar + scalar * br;
    }
  }

  double asum = 0.0;
  #pragma omp parallel for reduction(+:asum)
  for (size_t i=0; i<length; i++) {
    asum += fabs(h_C[i]);
  }

  free(h_A);
  free(h_B);
  free(h_C);
  omp_target_free(d_A, device_id);
  omp_target_free(d_B, device_id);
  omp_target_free(d_C, device_id);

  double epsilon=1.e-8;
  if (fabs(cr - asum)/asum > epsilon) {
    printf("Failed Validation on output array\n"
    "       Expected checksum: %lf\n"
    "       Observed checksum: %lf\n"
    "ERROR: solution did not validate\n", cr, asum);
  return 1;
  } else {
    printf("Solution validates\n");
  double avgtime = nstream_time/iterations;
  printf("Checksum = %lf; Avg time (s): %lf\n", asum, avgtime);
  }

  return 0;
}
```

## パフォーマンス上の考察

上記の例 (`map` 節、`omp_target_alloc`、`omp_target_alloc_device`、`omp_target_alloc_host`、`omp_target_alloc_shared`、`omp_target_memcpy` を使用) では、メインのカーネルは配列 `C` の値を計算する `target` 構造です。正確なタイミングを取得するため、この `target` 構造はループ内にありオフロードを反復します (反復数は 100)。反復ループにかかった合計時間を 100 で割って、平均カーネル時間を計算します。

```
// 計算を iterations 回繰り返します

nstream_time = omp_get_wtime();
for (int iter = 0; iter<iterations; iter++) {
    #pragma omp target teams distribute parallel for \
       is_device_ptr(A,B,C)
    for (size_t i=0; i<length; i++) {
        C[i] += A[i] + scalar * B[i];
    }
}
nstream_time = omp_get_wtime() - nstream_time;
```

`LIBOMPTARGET_DEBUG=1` の出力は、上記のすべての例が同じ ND_range のパーティション化であることを示しています。

```
Target LEVEL0 RTL --> Allocated a device memory 0xff00000020200000
Libomptarget --> omp_target_alloc returns device ptr 0xff00000020200000
Libomptarget --> Call to omp_target_alloc for device 0 requesting 536870912 bytes
Libomptarget --> Call to omp_get_num_devices returning 1
Libomptarget --> Call to omp_get_initial_device returning 1
Libomptarget --> Checking whether device 0 is ready.
Libomptarget --> Is the device 0 (local ID 0) initialized? 1
Libomptarget --> Device 0 is ready to use.
```

次の表は、使用した特定の GPU (1 スタックのみ) で実行した際の、それぞれのカーネルが要した平均時間です。

| バージョン | 時間 (秒) |
|---|---|
| `map` | 0.183604 |
| `map + target data` | 0.012757 |
| `omp_target_alloc` | 0.002501 |
| `omp_target_alloc_device` | 0.002499 |
| `omp_target_alloc_host` | 0.074412 |
| `omp_target_alloc_shared` | 0.012491 |
| `omp_target_memcpy` | 0.011072 |

上記のパフォーマンス値を見ると、`map` バージョンが最も低速である (0.183604 秒) ことが分かります。これは、各カーネルの起動、および最初と最後のデータ転送が影響しています。メインのカーネルは 100 回起動されています。各カーネル起動の最初でデバイス上に配列 `A`、`B`、`C` のストレージが確保され、これらの配列の値がホストからデバイスにコピーされます。カーネルの最後で、配列 `C` の値はデバイスからホストにコピーされます。反復ループ全体を `map` 節を持つ `target data` 構造内に置くと、反復ループの最初のカーネル起動に 1 回、そしてループの最後のカーネルの後に 1 回転送が行われるため、実行時間は 0.012757 秒に短縮されます。

`omp_target_alloc` と `omp_target_alloc_device` のバージョンが最良のパフォーマンスを示しています (0.002501 秒と 0.002499 秒)。これらのバージョンでは、`A`、`B` および `C` のストレージはデバイスメモリーに直接割り当てられ、デバイス上のアクセスはデバイスのローカルメモリーになります。これは、デバイス側でスクラッチの配列を使用するアプリケーションには有用なモデルです。これらの配列は、ホストでアクセスする必要がありません。このような場合、例に示すように、デバイスに配列を割り当てて、データを転送しないことを推奨します。

omp_target_alloc_shared バージョンの良好なパフォーマンスを示していますが、最良の結果にはおよびません (0.012491 秒)。このバージョンでは、A、B および C のストレージは共有メモリーに割り当てられます。そのため、ホストとデバイス間でデータを移行することができます。移行によるオーバーヘッドはありますが、移行後はデバイス上のアクセスは高速なデバイスのローカルメモリーから行われます。このバージョンでは、配列の初期化はホストで行われます。最初のカーネル起動時に配列がデバイスに移行され、カーネルはデバイス上でローカルに配列にアクセスします。最後に、ホストがリダクション計算を行う前に、C 配列全体がホストに移行されます。

omp_target_alloc_host バージョン (0.074412 秒) は、omp_target_alloc_shared バージョンよりもおよそ 6 倍の時間を要しています。これは、ホストメモリーに割り当てられたデータが、ホストからデバイスに移行しないためです。データは通常、デバイスとホストを接続する PCIe* などのバスを介して転送されます。これは、ローカルデバイスのメモリーアクセスよりも低速です。デバイス上でアクセスされるデータが少量であれば、omp_target_alloc_host を使用してホストメモリーにデータを割り当てるべきです。ただし、配列がデバイス上で頻繁にアクセスされる場合、配列はデバイスメモリーに保持する必要があります。データをホストメモリーに保持して PCIe* 経由でアクセスすると、パフォーマンスが低下します。

最後に、データの転送に関する注意点を述べます。map バージョンの転送データ量は、LIBOMPTARGET_DEBUG=1 の出力に grep "Libomptarget --> Moving" を実行して確認できます。メインのカーネルを呼び出すごとに、次のようなデータ転送が行われることに注意してください。

```
$ grep "Libomptarget --> Moving" test_target_map.debug
Libomptarget --> Moving 536870912 bytes (hst:0x00007f1a5fc8b010) ->
(tgt:0xff00000000200000)
Libomptarget --> Moving 536870912 bytes (hst:0x00007f1a9fc8d010) ->
(tgt:0xff00000020200000)
Libomptarget --> Moving 536870912 bytes (hst:0x00007f1a7fc8c010) ->
(tgt:0xff00000040200000)
Libomptarget --> Moving 536870912 bytes (tgt:0xff00000000200000) ->
(hst:0x00007f1a5fc8b010)
```

一方、omp_target_alloc_... バージョンのデータ転送は、ランタイムの下層で処理されます。LIBOMPTARGET_DEBUG=1 の出力に grep "Libomptarget --> Moving" コマンドを実行してもデータ転送は表示されません。

**Fortran の例**

Fortran では target data と map 節を次のように使用します。

```
1.  program main
2.      use iso_fortran_env
3.      use omp_lib
4.      implicit none
5.
6.      integer, parameter :: iterations=100
7.      integer, parameter :: length=64*1024*1024
8.      real(kind=REAL64), parameter ::  epsilon=1.D-8
9.      real(kind=REAL64), allocatable ::  A(:)
10.     real(kind=REAL64), allocatable ::  B(:)
11.     real(kind=REAL64), allocatable ::  C(:)
12.     real(kind=REAL64) :: scalar=3.0
13.     real(kind=REAL64) :: ar, br, cr, asum
14.     real(kind=REAL64) :: nstream_time, avgtime
15.     integer :: err, i, iter
16.
17.     !
18.     ! ホスト上でプレーンな割り当てにより配列を割り当て
19.
20.     allocate( A(length), stat=err )
21.     if (err .ne. 0) then
```

```
22.       print *, "Allocation of A returned ", err
23.       stop 1
24.     endif
25.
26.     allocate( B(length), stat=err )
27.     if (err .ne. 0) then
28.       print *, "Allocation of B returned ", err
29.       stop 1
30.     endif
31.
32.     allocate( C(length), stat=err )
33.     if (err .ne. 0) then
34.       print *, "Allocation of C returned ", err
35.       stop 1
36.     endif
37.
38.     !
39.     ! 配列の初期化
40.
41.     !$omp parallel do
42.     do i = 1, length
43.        A(i) = 2.0
44.        B(i) = 2.0
45.        C(i) = 0.0
46.     end do
47.
48.     !
49.     ! 計算を実行
50.
51.     nstream_time = omp_get_wtime()
52.     !$omp target data  map(to: A, B) map(tofrom: C)
53.
54.     do iter = 1, iterations
55.        !$omp target teams distribute parallel do
56.        do i = 1, length
57.           C(i) = C(i) + A(i) + scalar * B(i)
58.        end do
59.     end do
60.
61.     !$omp end target data
62.     nstream_time = omp_get_wtime() - nstream_time
63.
64.     !
65.     ! 結果を評価して出力
66.
67.     ar = 2.0
68.     br = 2.0
69.     cr = 0.0
70.     do iter = 1, iterations
71.        do i = 1, length
72.           cr = cr + ar + scalar * br
73.        end do
74.     end do
75.
76.     asum = 0.0
77.     !$omp parallel do reduction(+:asum)
78.     do i = 1, length
79.        asum = asum + abs(C(i))
80.     end do
81.
82.     if (abs(cr - asum)/asum > epsilon) then
83.        write(*,110) "Failed Validation on output array: Expected =", cr, ", Observed
    =", asum
84.     else
85.        avgtime = nstream_time/iterations
86.        write(*,120) "Solution validates: Checksum =", asum, ", Avg time (s) =",
    avgtime
87.     endif
```

```
88.
89. 110 format (A, F20.6, A, F20.6)
90. 120 format (A, F20.6, A, F10.6)
91.
92.     deallocate(A)
93.     deallocate(B)
94.     deallocate(C)
95.
96.     end program main
```

以下に、omp_target_alloc_device を使用した Fortran の例を示します。この例では、アロケーター omp_target_device_mem_alloc を使用して、デバイス上に配列 A、B そして C を割り当てています。target data ディレクティブに use_device_addr(A, B, C) 節を指定して、配列がデバイスアドレスを持ち、アドレスが target 領域で使用されることを明示します。

```
1.  use iso_fortran_env
2.      use omp_lib
3.      implicit none
4.
5.      integer, parameter :: iterations=100
6.      integer, parameter :: length=64*1024*1024
7.      real(kind=REAL64), parameter ::  epsilon=1.D-8
8.      real(kind=REAL64), allocatable ::  A(:)
9.      real(kind=REAL64), allocatable ::  B(:)
10.     real(kind=REAL64), allocatable ::  C(:)
11.     real(kind=REAL64) :: scalar=3.0
12.     real(kind=REAL64) :: ar, br, cr, asum
13.     real(kind=REAL64) :: nstream_time, avgtime
14.     integer :: i, iter
15.
16.     !
17.     ! デバイスメモリーに配列を割り当て
18.
19.     !$omp allocators allocate(allocator(omp_target_device_mem_alloc): A)
20.     allocate(A(length))
21.
22.     !$omp allocators allocate(allocator(omp_target_device_mem_alloc): B)
23.     allocate(B(length))
24.
25.     !$omp allocators allocate(allocator(omp_target_device_mem_alloc): C)
26.     allocate(C(length))
27.
28.     !
29.     ! target data の始まり
30.
31.     !$omp target data use_device_addr(A, B, C)
32.
33.     !
34.     ! 配列の初期化
35.
36.     !$omp target teams distribute parallel do
37.     do i = 1, length
38.        A(i) = 2.0
39.        B(i) = 2.0
40.        C(i) = 0.0
41.     end do
42.
43.     !
44.     ! 計算を実行
45.
46.     nstream_time = omp_get_wtime()
47.     do iter = 1, iterations
48.        !$omp target teams distribute parallel do
49.        do i = 1, length
50.           C(i) = C(i) + A(i) + scalar * B(i)
```

```
51.        end do
52.     end do
53.     nstream_time = omp_get_wtime() - nstream_time
54.
55.     !
56.     ! 結果を評価して出力
57.
58.     ar = 2.0
59.     br = 2.0
60.     cr = 0.0
61.     do iter = 1, iterations
62.        do i = 1, length
63.           cr = cr + ar + scalar * br
64.        end do
65.     end do
66.
67.     asum = 0.0
68.     !$omp target teams distribute parallel do reduction(+:asum) &
69.     !$omp map(tofrom: asum)
70.     do i = 1, length
71.        asum = asum + abs(C(i))
72.     end do
73.
74.     !
75.     ! target data の終わり
76.
77.     !$omp end target data
78.
79.     if (abs(cr - asum)/asum > epsilon) then
80.        write(*,110) "Failed Validation on output array: Expected =", cr, ", Observed
    =", asum
81.     else
82.        avgtime = nstream_time/iterations
83.        write(*,120) "Solution validates: Checksum =", asum, ", Avg time (s) =",
    avgtime
84.     endif
85.
86. 110 format (A, F20.6, A, F20.6)
87. 120 format (A, F20.6, A, F10.6)
88.
89.     deallocate(A)
90.     deallocate(B)
91.     deallocate(C)
92.
93.     end program main
```

## Fortran の例

メモリー割り当てを適切に選択することで、冗長なデータ転送を回避してパフォーマンスを向上する Fortran の例を説明します。

この Fortran の例では、OpenMP* オフロード機能と、ハイパフォーマンス・アプリケーションである NWChem を使用しています。

この例では、4 レベルの入れ子になったループがあります。最も内側の `k` ループは、`sgemm` への 2 つの呼び出しをデバイスにオフロードし、リダクション計算 (リダクション変数 `emp4i` と `emp5i`) をデバイスにオフロードする `omp_fbody` ルーチンを呼び出します。

ループの入れ子では、いくつかの配列がホストで更新されます (配列 `Tkj`、`Kkj`、`Jia`、`Kia`、`Tia`、`Xia` および `t1v1`)。そのため、デバイス上のこれらの配列の値とホストの値を一致させる必要があります。

## バージョン 1

プログラムの最初のバージョンでは、`allocate` ディレクティブを使用して共有メモリーに 11 個の配列を割り当てています。

```
!$omp allocators allocate(allocator(omp_target_shared_mem_alloc): f1n)
allocate( f1n(1:nvir,1:nvir) )

!$omp allocators allocate(allocator(omp_target_shared_mem_alloc): f2n)
allocate( f2n(1:nvir,1:nvir) )

!$omp allocators allocate(allocator(omp_target_shared_mem_alloc): eorb)
allocate( eorb(1:nbf) )
...
```

共有割り当ては、ホストとデバイスからアクセスでき、必要に応じてホストとデバイス間で自動的に移行されます。そのため、共有割り当てへのポインターは、ホストとデバイスで同じポインターを使用できます。

バージョン 1 を以下に示します。

```
1.  include "mkl_omp_offload.f90"
2.
3.        subroutine omp_fbody(f1n,f2n,eorb,                          &
4.                             ncor,nocc,nvir, emp4,emp5,a,i,j,k,klo, &
5.                             Jia, Kia, Tia, Xia, Tkj, Kkj,          &
6.                             t1v1,t1v2)
7.
8.          use omp_lib
9.          use onemkl_blas_omp_offload_lp64
10.         use iso_fortran_env
11.         implicit none
12.
13.         real, intent(inout) :: emp4,emp5
14.         integer, intent(in) :: ncor,nocc,nvir
15.         integer, intent(in) :: a,i,j,k, klo
16.         real, intent(inout) :: f1n(nvir,nvir)
17.         real, intent(inout) :: f2n(nvir,nvir)
18.         real, intent(in)    :: eorb(*)
19.         real, intent(in)    :: Jia(*), Kia(*), Tia(*), Xia(*)
20.         real, intent(in)    :: Tkj(*), Kkj(*)
21.         real, intent(in)    :: t1v1(nvir),t1v2(nvir)
22.         real    :: emp4i,emp5i
23.         real    :: eaijk,denom
24.         integer :: lnov,lnvv
25.         integer :: b,c
26.         real    :: f1nbc,f1ncb,f2nbc,f2ncb
27.         real    :: t1v1b,t1v2b
28.
29.         lnov=nocc*nvir
30.         lnvv=nvir*nvir
31.         emp4i = 0.0
32.         emp5i = 0.0
33.
34.         !$omp dispatch
35.         call sgemm('n','t',nvir,nvir,nvir,1.0,Jia,nvir,       &
36.                    Tkj(1+(k-klo)*lnvv),nvir,1.0,f1n,nvir)
37.
38.         !$omp dispatch
39.         call sgemm('n','t',nvir,nvir,nvir,1.0,Kia,nvir,       &
40.                    Tkj(1+(k-klo)*lnvv),nvir,1.0, f2n,nvir)
41.
42.         !$omp dispatch
43.         call sgemm('n','n',nvir,nvir,nocc,-1.0,Tia,nvir,      &
44.                    Kkj(1+(k-klo)*lnov),nocc,1.0, f1n,nvir)
45.
```

```
46.         !$omp dispatch
47.         call sgemm('n','n',nvir,nvir,nocc,-1.0,Xia,nvir,      &
48.                    Kkj(1+(k-klo)*lnov),nocc,1.0, f2n,nvir)
49.
50.         eaijk = eorb(a) - ( eorb(ncor+i)+eorb(ncor+j)+eorb(ncor+k) )
51.
52.         !$omp target teams distribute parallel do collapse(2)       &
53.         !$omp     reduction(+:emp4i,emp5i)                          &
54.         !$omp     private(f1nbc,f1ncb,f2nbc,f2ncb)                  &
55.         !$omp     private(t1v1b,t1v2b)                              &
56.         !$omp     private(denom) firstprivate(eaijk,nvir,ncor,nocc)
57.         do b=1,nvir
58.            do c=1,nvir
59.               denom=-1.0/( eorb(ncor+nocc+b)+eorb(ncor+nocc+c)+eaijk )
60.
61.               f1nbc = f1n(b,c);
62.               f1ncb = f1n(c,b);
63.               f2nbc = f2n(b,c);
64.               f2ncb = f2n(c,b);
65.               t1v1b = t1v1(b);
66.               t1v2b = t1v2(b);
67.
68.               emp4i = emp4i + (denom*t1v1b*f1nbc) + (denom*2*f1ncb)
69.               emp5i = emp5i + (denom*t1v2b*f2nbc) + (denom*3*f2ncb)
70.           enddo
71.         enddo
72.         !$omp end target teams distribute parallel do
73.
74.         emp4 = emp4 + emp4i
75.         emp5 = emp5 + emp5i
76.
77.         end
78.
79.
80.       subroutine init_array_1(arr, m)
81.         implicit none
82.         real, intent(inout) :: arr(m)
83.         integer m, i
84.
85.         do i = 1, m
86.            arr(i) = 1.0/(100.0 + i-1)
87.         end do
88.         end subroutine init_array_1
89.
90.
91.       subroutine init_array_2(arr, m, n)
92.         implicit none
93.         real, intent(inout) :: arr(m, n)
94.         integer m, n, i, j
95.
96.         do i = 1, m
97.            do j = 1, n
98.               arr(i,j) = 1.0/(100.0 + ((i-1) * n) + j)
99.            end do
100.        end do
101.      end subroutine init_array_2
102.
103.
104.      program main
105.
106.        use omp_lib
107.        use iso_fortran_env
108.        implicit none
109.
110.        interface
111.           subroutine omp_fbody(f1n,f2n,eorb,                    &
112.                        ncor,nocc,nvir, emp4,emp5,a,i,j,k,klo,  &
113.                        Jia, Kia, Tia, Xia, Tkj, Kkj,           &
114.                        t1v1,t1v2)
```

```
115.           real, intent(inout) :: emp4,emp5
116.           integer, intent(in) :: ncor,nocc,nvir
117.           integer, intent(in) :: a,i,j,k, klo
118.           real, intent(inout) :: f1n(nvir,nvir)
119.           real, intent(inout) :: f2n(nvir,nvir)
120.           real, intent(in)    :: eorb(*)
121.           real, intent(in)    :: Jia(*), Kia(*), Tia(*), Xia(*), Tkj(*), Kkj(*)
122.           real, intent(in)    :: t1v1(nvir),t1v2(nvir)
123.           end subroutine omp_fbody
124.        end interface
125.
126.        integer :: ncor, nocc, nvir, maxiter, nkpass
127.        integer :: nbf, lnvv, lnov, kchunk
128.        real, allocatable :: eorb(:)
129.        real, allocatable :: f1n(:,:)
130.        real, allocatable :: f2n(:,:)
131.
132.        real, allocatable :: Jia(:)
133.        real, allocatable :: Kia(:)
134.        real, allocatable :: Tia(:)
135.        real, allocatable :: Xia(:)
136.        real, allocatable :: Tkj(:)
137.        real, allocatable :: Kkj(:)
138.
139.        real, allocatable :: t1v1(:),t1v2(:)
140.        real emp4, emp5
141.        integer :: a, b, c, i, j, k
142.        integer :: klo, khi, iter
143.        double precision, allocatable :: timers(:)
144.        double precision :: t0, t1, tsum, tmax, tmin, tavg
145.
146.!       実行パラメーター
147.        nocc = 256
148.        nvir = 2048
149.        maxiter = 50
150.        nkpass = 1
151.        ncor = 0
152.
153.        print *, "Run parameters:"
154.        print *, "nocc    =", nocc
155.        print *, "nvir    =", nvir
156.        print *, "maxiter =", maxiter
157.        print *, "nkpass  =", nkpass
158.        print *, "ncor    =", ncor
159.        print *, " "
160.
161.!       配列の割り当てと初期化。
162.
163.        nbf = ncor + nocc + nvir
164.        lnvv = nvir * nvir
165.        lnov = nocc * nvir
166.        kchunk = (nocc - 1)/nkpass + 1
167.
168.        !$omp allocators allocate(allocator(omp_target_shared_mem_alloc): f1n)
169.        allocate( f1n(1:nvir,1:nvir) )
170.
171.        !$omp allocators allocate(allocator(omp_target_shared_mem_alloc): f2n)
172.        allocate( f2n(1:nvir,1:nvir) )
173.
174.        !$omp allocators allocate(allocator(omp_target_shared_mem_alloc): eorb)
175.        allocate( eorb(1:nbf) )
176.
177.        !$omp allocators allocate(allocator(omp_target_shared_mem_alloc): Jia)
178.        allocate( Jia(1:lnvv) )
179.
180.        !$omp allocators allocate(allocator(omp_target_shared_mem_alloc): Kia)
181.        allocate( Kia(1:lnvv) )
182.
```

```
183.        !$omp allocators allocate(allocator(omp_target_shared_mem_alloc): Tia)
184.        allocate( Tia(1:lnov*nocc) )
185.
186.        !$omp allocators allocate(allocator(omp_target_shared_mem_alloc): Xia)
187.        allocate( Xia(1:lnov*nocc))
188.
189.        !$omp allocators allocate(allocator(omp_target_shared_mem_alloc): Tkj)
190.        allocate( Tkj(1:kchunk*lnvv) )
191.
192.        !$omp allocators allocate(allocator(omp_target_shared_mem_alloc): Kkj)
193.        allocate( Kkj(1:kchunk*lnvv) )
194.
195.        !$omp allocators allocate(allocator(omp_target_shared_mem_alloc): t1v1)
196.        allocate( t1v1(1:lnvv) )
197.
198.        !$omp allocators allocate(allocator(omp_target_shared_mem_alloc): t1v2)
199.        allocate( t1v2(1:lnvv) )
200.!
201.        call init_array_1(eorb, nbf)
202.        call init_array_1(Jia, lnvv)
203.        call init_array_1(Kia, lnvv)
204.        call init_array_1(Tia, lnov*nocc)
205.        call init_array_1(Xia, lnov*nocc)
206.        call init_array_1(Tkj, kchunk*lnvv)
207.        call init_array_1(Kkj, kchunk*lnov)
208.        call init_array_1(t1v1, lnvv)
209.        call init_array_1(t1v2, lnvv)
210.
211.        call init_array_2(f1n, nvir, nvir)
212.        call init_array_2(f2n, nvir, nvir)
213.
214.        print *, "End of initialization"
215.
216.        allocate (timers(1:maxiter))
217.
218.        emp4=0.0
219.        emp5=0.0
220.        a=1
221.        iter=1
222.
223.        do klo = 1, nocc, kchunk
224.           khi = MIN(nocc, klo+kchunk-1)
225.           do j = 1, nocc
226.
227.#if defined(DO_UPDATE_ARRAYS)
228.!             Tkj と Kkj の更新
229.              Tkj((khi-klo+1)*lnvv) = Tkj((khi-klo+1)*lnvv) + 1.0
230.              Kkj((khi-klo+1)*lnov) = Kkj((khi-klo+1)*lnov) + 1.0
231.#endif
232.
233.              do i = 1, nocc
234.
235.#if defined(DO_UPDATE_ARRAYS)
236.!                Jia、Kia、Tia、Xia 配列の要素を更新
237.                 Jia(lnvv) = Jia(lnvv) + 1.0
238.                 Kia(lnvv) = Kia(lnvv) + 1.0
239.                 Tia(lnov) = Tia(lnov) + 1.0
240.                 Xia(lnov) = Xia(lnov) + 1.0
241.#endif
242.
243.                 do k = klo, MIN(khi,i)
244.
245.#if defined(DO_UPDATE_ARRAYS)
246.!                   t1v1 配列の要素を更新
247.                    t1v1(:) = Tkj(lnvv-nvir+1:lnvv)
248.#endif
249.
250.                    t0 = omp_get_wtime()
```

```
251.
252.                     call omp_fbody(f1n,f2n,eorb,                       &
253.                               ncor,nocc,nvir, emp4,emp5,a,i,j,k,klo, &
254.                               Jia, Kia, Tia, Xia, Tkj, Kkj,          &
255.                               t1v1,t1v2)
256.
257.                     t1 = omp_get_wtime()
258.                     timers(iter) = (t1-t0)
259.                     if (iter .eq. maxiter) then
260.                         print *, "Stopping after ", iter, "iterations"
261.                         print *, " "
262.                         goto 150
263.                     endif
264.
265.!                    大きな maxiter の NAN を防止
266.                     if (emp4 >  1000.0) then
267.                         emp4 = emp4 - 1000.0
268.                     endif
269.                     if (emp4 < -1000.0) then
270.                         emp4 = emp4 + 1000.0
271.                     endif
272.                     if (emp5 >  1000.0) then
273.                         emp5 = emp5 - 1000.0
274.                     endif
275.                     if (emp5 < -1000.0) then
276.                         emp5 = emp5 + 1000.0
277.                      endif
278.
279.                     iter = iter + 1
280.
281.                 end do ! k = klo, MIN(khi,i)
282.              end do ! do i = 1, nocc
283.           end do ! do j = 1, nocc
284.        end do ! do klo = 1, nocc, kchunk
285.
286. 150    CONTINUE
287.
288.        tsum =  0.0
289.        tmax = -1.0e10
290.        tmin =  1.0e10
291.        do i = 2, iter
292.           tsum = tsum + timers(i)
293.           tmax = MAX(tmax,timers(i))
294.           tmin = MIN(tmin,timers(i))
295.        end do
296.
297.        tavg = tsum / (iter - 1)
298.        print *, "TOTAL ITER: ", iter
299.        write(*, 110) " TIMING: min=", tmin, ", max=", tmax, ", avg of iters after
    first=", tavg, " seconds"
300. 110    format (A, F9.6, A, F9.6, A, F9.6, A)
301.
302.        write(*, 120) " emp4 = ", emp4, " emp5 =", emp5
303. 120    format (A, F15.3, A, F15.3)
304.
305.        print *, "END"
306.
307.        deallocate (f1n)
308.        deallocate (f2n)
309.        deallocate (eorb)
310.        deallocate (Jia)
311.        deallocate (Kia)
312.        deallocate (Tia)
313.        deallocate (Xia)
314.        deallocate (Tkj)
315.        deallocate (Kkj)
316.
317.        deallocate (t1v1)
318.        deallocate (t1v2)
```

```
319.        deallocate (timers)
320.
321.        end program
```

このバージョンは最も簡単なプログラム構成ですが、効率も良くありません。

## バージョン 2

バージョン 2 でが、単純な割り当てを使用して 11 個すべての配列をシステムメモリーに割り当て、`map` 句を使用して配列をデバイスにマップします。

```
real, allocatable :: f1n(:,:)
real, allocatable :: f2n(:,:)
real, allocatable :: eorb(:)
...

!$omp target data                    &
   map(to: f1n, f2n, eorb)           &
   map(to: Jia, Kia, Tia, Xia)       &
   map(to: Tkj, Kkj)                 &
   map(to: t1v1, t1v2)
```

デバイスにマップされた配列がホスト上で更新されたら、OpenMP* の `target update to` ディレクティブを使用して、配列の値をホストからデバイスにコピーして更新します。不要な配列のコピーを排除するため、可能な限り入れ子になったループの最上位レベルにこのディレクティブを配置します。

```
Tkj((khi-klo+1)*lnvv) = Tkj((khi-klo+1)*lnvv) + 1.0
Kkj((khi-klo+1)*lnov) = Kkj((khi-klo+1)*lnov) + 1.0

!$omp target update to (Tkj, Kkj)
```

完全なバージョン 2 を以下に示します。

```
1.  include "mkl_omp_offload.f90"
2.
3.        subroutine omp_fbody(f1n,f2n,eorb,                          &
4.                             ncor,nocc,nvir, emp4,emp5,a,i,j,k,klo, &
5.                             Jia, Kia, Tia, Xia, Tkj, Kkj,          &
6.                             t1v1,t1v2)
7.
8.          use omp_lib
9.          use onemkl_blas_omp_offload_lp64
10.         use iso_fortran_env
11.         implicit none
12.
13.         real, intent(inout) :: emp4,emp5
14.         integer, intent(in) :: ncor,nocc,nvir
15.         integer, intent(in) :: a,i,j,k, klo
16.         real, intent(inout) :: f1n(nvir,nvir)
17.         real, intent(inout) :: f2n(nvir,nvir)
18.         real, intent(in)    :: eorb(*)
19.         real, intent(in)    :: Jia(*), Kia(*), Tia(*), Xia(*)
20.         real, intent(in)    :: Tkj(*), Kkj(*)
21.         real, intent(in)    :: t1v1(nvir),t1v2(nvir)
22.         real    :: emp4i,emp5i
23.         real    :: eaijk,denom
24.         integer :: lnov,lnvv
25.         integer :: b,c
26.         real    :: f1nbc,f1ncb,f2nbc,f2ncb
27.         real    :: t1v1b,t1v2b
28.
29.         lnov=nocc*nvir
30.         lnvv=nvir*nvir
```

```
31.          emp4i = 0.0
32.          emp5i = 0.0
33.
34.          !$omp dispatch
35.          call sgemm('n','t',nvir,nvir,nvir,1.0,Jia,nvir,       &
36.                     Tkj(1+(k-klo)*lnvv),nvir,1.0,f1n,nvir)
37.
38.          !$omp dispatch
39.          call sgemm('n','t',nvir,nvir,nvir,1.0,Kia,nvir,       &
40.                     Tkj(1+(k-klo)*lnvv),nvir,1.0, f2n,nvir)
41.
42.          !$omp dispatch
43.          call sgemm('n','n',nvir,nvir,nocc,-1.0,Tia,nvir,      &
44.                     Kkj(1+(k-klo)*lnov),nocc,1.0, f1n,nvir)
45.
46.          !$omp dispatch
47.          call sgemm('n','n',nvir,nvir,nocc,-1.0,Xia,nvir,      &
48.                     Kkj(1+(k-klo)*lnov),nocc,1.0, f2n,nvir)
49.
50.          eaijk = eorb(a) - ( eorb(ncor+i)+eorb(ncor+j)+eorb(ncor+k) )
51.
52.          !$omp target teams distribute parallel do collapse(2)       &
53.          !$omp     reduction(+:emp4i,emp5i)                          &
54.          !$omp     private(f1nbc,f1ncb,f2nbc,f2ncb)                  &
55.          !$omp     private(t1v1b,t1v2b)                              &
56.          !$omp     private(denom) firstprivate(eaijk,nvir,ncor,nocc)
57.          do b=1,nvir
58.             do c=1,nvir
59.                denom=-1.0/( eorb(ncor+nocc+b)+eorb(ncor+nocc+c)+eaijk )
60.
61.                f1nbc = f1n(b,c);
62.                f1ncb = f1n(c,b);
63.                f2nbc = f2n(b,c);
64.                f2ncb = f2n(c,b);
65.                t1v1b = t1v1(b);
66.                t1v2b = t1v2(b);
67.
68.                emp4i = emp4i + (denom*t1v1b*f1nbc) + (denom*2*f1ncb)
69.                emp5i = emp5i + (denom*t1v2b*f2nbc) + (denom*3*f2ncb)
70.            enddo
71.          enddo
72.          !$omp end target teams distribute parallel do
73.
74.          emp4 = emp4 + emp4i
75.          emp5 = emp5 + emp5i
76.
77.          end
78.
79.
80.        subroutine init_array_1(arr, m)
81.          implicit none
82.          real, intent(inout) :: arr(m)
83.          integer m, i
84.
85.          do i = 1, m
86.             arr(i) = 1.0/(100.0 + i-1)
87.          end do
88.          end subroutine init_array_1
89.
90.
91.        subroutine init_array_2(arr, m, n)
92.          implicit none
93.          real, intent(inout) :: arr(m, n)
94.          integer m, n, i, j
95.
96.          do i = 1, m
97.             do j = 1, n
98.                arr(i,j) = 1.0/(100.0 + ((i-1) * n) + j)
99.             end do
```

```
100.        end do
101.      end subroutine init_array_2
102.
103.
104.      program main
105.
106.        use omp_lib
107.        use iso_fortran_env
108.        implicit none
109.
110.        interface
111.           subroutine omp_fbody(f1n,f2n,eorb,                  &
112.                        ncor,nocc,nvir, emp4,emp5,a,i,j,k,klo, &
113.                        Jia, Kia, Tia, Xia, Tkj, Kkj,          &
114.                        t1v1,t1v2)
115.           real, intent(inout) :: emp4,emp5
116.           integer, intent(in) :: ncor,nocc,nvir
117.           integer, intent(in) :: a,i,j,k, klo
118.           real, intent(inout) :: f1n(nvir,nvir)
119.           real, intent(inout) :: f2n(nvir,nvir)
120.           real, intent(in)    :: eorb(*)
121.           real, intent(in)    :: Jia(*), Kia(*), Tia(*), Xia(*), Tkj(*), Kkj(*)
122.           real, intent(in)    :: t1v1(nvir),t1v2(nvir)
123.           end subroutine omp_fbody
124.        end interface
125.
126.        integer :: ncor, nocc, nvir, maxiter, nkpass
127.        integer :: nbf, lnvv, lnov, kchunk
128.        real, allocatable :: eorb(:)
129.        real, allocatable :: f1n(:,:)
130.        real, allocatable :: f2n(:,:)
131.
132.        real, allocatable :: Jia(:)
133.        real, allocatable :: Kia(:)
134.        real, allocatable :: Tia(:)
135.        real, allocatable :: Xia(:)
136.        real, allocatable :: Tkj(:)
137.        real, allocatable :: Kkj(:)
138.
139.        real, allocatable :: t1v1(:),t1v2(:)
140.        real emp4, emp5
141.        integer :: a, b, c, i, j, k
142.        integer :: klo, khi, iter
143.        double precision, allocatable :: timers(:)
144.        double precision :: t0, t1, tsum, tmax, tmin, tavg
145.
146.!       実行パラメーター
147.        nocc = 256
148.        nvir = 2048
149.        maxiter = 50
150.        nkpass = 1
151.        ncor = 0
152.
153.        print *, "Run parameters:"
154.        print *, "nocc    =", nocc
155.        print *, "nvir    =", nvir
156.        print *, "maxiter =", maxiter
157.        print *, "nkpass  =", nkpass
158.        print *, "ncor    =", ncor
159.        print *, " "
160.
161.!       配列の割り当てと初期化。
162.
163.        nbf = ncor + nocc + nvir
164.        lnvv = nvir * nvir
165.        lnov = nocc * nvir
166.        kchunk = (nocc - 1)/nkpass + 1
167.
```

```
168.         allocate( f1n(1:nvir,1:nvir) )
169.         allocate( f2n(1:nvir,1:nvir) )
170.         allocate( eorb(1:nbf) )
171.         allocate( Jia(1:lnvv) )
172.         allocate( Kia(1:lnvv) )
173.         allocate( Tia(1:lnov*nocc) )
174.         allocate( Xia(1:lnov*nocc))
175.         allocate( Tkj(1:kchunk*lnvv) )
176.         allocate( Kkj(1:kchunk*lnvv) )
177.         allocate( t1v1(1:lnvv) )
178.         allocate( t1v2(1:lnvv) )
179.!
180.         call init_array_1(eorb, nbf)
181.         call init_array_1(Jia, lnvv)
182.         call init_array_1(Kia, lnvv)
183.         call init_array_1(Tia, lnov*nocc)
184.         call init_array_1(Xia, lnov*nocc)
185.         call init_array_1(Tkj, kchunk*lnvv)
186.         call init_array_1(Kkj, kchunk*lnov)
187.         call init_array_1(t1v1, lnvv)
188.         call init_array_1(t1v2, lnvv)
189.
190.         call init_array_2(f1n, nvir, nvir)
191.         call init_array_2(f2n, nvir, nvir)
192.
193.         print *, "End of initialization"
194.
195.         allocate (timers(1:maxiter))
196.
197.         emp4=0.0
198.         emp5=0.0
199.         a=1
200.         iter=1
201.
202.         !$omp target data                    &
203.            map(to: f1n, f2n, eorb)          &
204.            map(to: Jia, Kia, Tia, Xia)      &
205.            map(to: Tkj, Kkj)                &
206.            map(to: t1v1, t1v2)
207.
208.         do klo = 1, nocc, kchunk
209.            khi = MIN(nocc, klo+kchunk-1)
210.            do j = 1, nocc
211.
212.#if defined(DO_UPDATE_ARRAYS)
213.!              Tkj と Kkj の更新
214.              Tkj((khi-klo+1)*lnvv) = Tkj((khi-klo+1)*lnvv) + 1.0
215.              Kkj((khi-klo+1)*lnov) = Kkj((khi-klo+1)*lnov) + 1.0
216.
217.              !$omp target update to (Tkj, Kkj)
218.#endif
219.
220.              do i = 1, nocc
221.
222.#if defined(DO_UPDATE_ARRAYS)
223.!                Jia、Kia、Tia、Xia 配列の要素を更新
224.                 Jia(lnvv) = Jia(lnvv) + 1.0
225.                 Kia(lnvv) = Kia(lnvv) + 1.0
226.                 Tia(lnov) = Tia(lnov) + 1.0
227.                 Xia(lnov) = Xia(lnov) + 1.0
228.
229.                 !$omp target update to (Jia, Kia, Tia, Xia)
230.#endif
231.
232.                 do k = klo, MIN(khi,i)
233.
234.#if defined(DO_UPDATE_ARRAYS)
235.!                   t1v1 配列の要素を更新
```

```
236.                     t1v1(:)= Tkj(lnvv-nvir+1:lnvv)
237.
238.                     !$omp target update to (t1v1)
239.#endif
240.
241.                     t0 = omp_get_wtime()
242.
243.                     call omp_fbody(f1n,f2n,eorb,                      &
244.                               ncor,nocc,nvir, emp4,emp5,a,i,j,k,klo, &
245.                               Jia, Kia, Tia, Xia, Tkj, Kkj,          &
246.                               t1v1,t1v2)
247.
248.                     t1 = omp_get_wtime()
249.                     timers(iter) = (t1-t0)
250.                     if (iter .eq. maxiter) then
251.                         print *, "Stopping after ", iter, "iterations"
252.                         print *, " "
253.                         goto 150
254.                     endif
255.
256.!                    大きな maxiter の NAN を防止
257.                     if (emp4 >  1000.0) then
258.                         emp4 = emp4 - 1000.0
259.                     endif
260.                     if (emp4 < -1000.0) then
261.                         emp4 = emp4 + 1000.0
262.                     endif
263.                     if (emp5 >  1000.0) then
264.                         emp5 = emp5 - 1000.0
265.                     endif
266.                     if (emp5 < -1000.0) then
267.                         emp5 = emp5 + 1000.0
268.                      endif
269.
270.                     iter = iter + 1
271.
272.                 end do ! k = klo, MIN(khi,i)
273.              end do ! do i = 1, nocc
274.           end do ! do j = 1, nocc
275.        end do ! do klo = 1, nocc, kchunk
276.
277. 150    CONTINUE
278.        !$omp end target data
279.
280.        tsum =  0.0
281.        tmax = -1.0e10
282.        tmin =  1.0e10
283.        do i = 2, iter
284.           tsum = tsum + timers(i)
285.           tmax = MAX(tmax,timers(i))
286.           tmin = MIN(tmin,timers(i))
287.        end do
288.
289.        tavg = tsum / (iter - 1)
290.        print *, "TOTAL ITER: ", iter
291.        write(*, 110) " TIMING: min=", tmin, ", max=", tmax, ", avg of iters after
    first=", tavg, " seconds"
292. 110    format (A, F9.6, A, F9.6, A, F9.6, A)
293.
294.        write(*, 120) " emp4 = ", emp4, " emp5 =", emp5
295. 120    format (A, F15.3, A, F15.3)
296.
297.        print *, "END"
298.
299.        deallocate (f1n)
300.        deallocate (f2n)
301.        deallocate (eorb)
302.        deallocate (Jia)
303.        deallocate (Kia)
```

```
304.        deallocate (Tia)
305.        deallocate (Xia)
306.        deallocate (Tkj)
307.        deallocate (Kkj)
308.
309.        deallocate (t1v1)
310.        deallocate (t1v2)
311.        deallocate (timers)
312.
313.        end program
```

## バージョン 3

バージョン 3 では、ホストのみ、デバイスのみ、またはホストとデバイスの両方で利用できる配列を検討します。また、プログラムの実行中に配列の値が更新されることも考慮します。これは、配列を割り当てる場所、配列を初期化する方法、およびデバイスで配列の値を更新するかどうか決定するのに役立ちます。

配列 `f1` と `f2` は、デバイス上で (SGEMM の結果を格納するため) ワーク配列として使用され、ホストからはアクセスできません。この配列は、デバイスに直接割り当てます。

`f1` と `f2` はデバイスに割り当てられているため、`init_array_2()` を呼び出して OpenMP* `target` 構造の配列を初期化します。

残りの 9 つの配列はホストとデバイスからアクセスされるため、ホストに割り当てて `init_array_1()` を呼び出して初期化します。その後、配列は `map` 句を使用してデバイスにマップされます。

OpenMP* `target update to` ディレクティブを使用して、更新された `Tkj`、`Kkj`、`Jia`、`Kia`、`Tia`、`Xia` および `t1v1` の値をホストからデバイスにコピーします。

```
!$omp allocators allocate(allocator(omp_target_device_mem_alloc): f1n)
allocate( f1n(1:nvir,1:nvir) )

!$omp allocators allocate(allocator(omp_target_device_mem_alloc): f2n)
allocate( f2n(1:nvir,1:nvir) )

!$omp allocators allocate(allocator(omp_target_host_mem_alloc): eorb)
allocate( eorb(1:nbf) )

!$omp allocators allocate(allocator(omp_target_host_mem_alloc): Jia)
allocate( Jia(1:lnvv) )

!$omp allocators allocate(allocator(omp_target_host_mem_alloc): Kia)
allocate( Kia(1:lnvv) )
...

!$omp target data                    &
   map(to: eorb)                    &
   map(to: Jia, Kia, Tia, Xia)      &
   map(to: Tkj, Kkj)                &
   map(to: t1v1, t1v2)
   ...

 Tkj((khi-klo+1)*lnvv) = Tkj((khi-klo+1)*lnvv) + 1.0
 Kkj((khi-klo+1)*lnov) = Kkj((khi-klo+1)*lnov) + 1.0

!$omp target update to (Tkj, Kkj)
```

プログラムの完全なバージョン 3 を以下に示します。

```
1.  include "mkl_omp_offload.f90"
2.
3.        subroutine omp_fbody(f1n,f2n,eorb,                          &
4.                             ncor,nocc,nvir, emp4,emp5,a,i,j,k,klo, &
5.                             Jia, Kia, Tia, Xia, Tkj, Kkj,          &
6.                             t1v1,t1v2)
7.
8.          use omp_lib
9.          use onemkl_blas_omp_offload_lp64
10.         use iso_fortran_env
11.         implicit none
12.
13.         real, intent(inout) :: emp4,emp5
14.         integer, intent(in) :: ncor,nocc,nvir
15.         integer, intent(in) :: a,i,j,k, klo
16.         real, intent(inout) :: f1n(nvir,nvir)
17.         real, intent(inout) :: f2n(nvir,nvir)
18.         real, intent(in)    :: eorb(*)
19.         real, intent(in)    :: Jia(*), Kia(*), Tia(*), Xia(*)
20.         real, intent(in)    :: Tkj(*), Kkj(*)
21.         real, intent(in)    :: t1v1(nvir),t1v2(nvir)
22.         real    :: emp4i,emp5i
23.         real    :: eaijk,denom
24.         integer :: lnov,lnvv
25.         integer :: b,c
26.         real    :: f1nbc,f1ncb,f2nbc,f2ncb
27.         real    :: t1v1b,t1v2b
28.
29.         lnov=nocc*nvir
30.         lnvv=nvir*nvir
31.         emp4i = 0.0
32.         emp5i = 0.0
33.
34.         !$omp dispatch
35.         call sgemm('n','t',nvir,nvir,nvir,1.0,Jia,nvir,       &
36.                    Tkj(1+(k-klo)*lnvv),nvir,1.0,f1n,nvir)
37.
38.         !$omp dispatch
39.         call sgemm('n','t',nvir,nvir,nvir,1.0,Kia,nvir,       &
40.                    Tkj(1+(k-klo)*lnvv),nvir,1.0, f2n,nvir)
41.
42.         !$omp dispatch
43.         call sgemm('n','n',nvir,nvir,nocc,-1.0,Tia,nvir,      &
44.                    Kkj(1+(k-klo)*lnov),nocc,1.0, f1n,nvir)
45.
46.         !$omp dispatch
47.         call sgemm('n','n',nvir,nvir,nocc,-1.0,Xia,nvir,      &
48.                    Kkj(1+(k-klo)*lnov),nocc,1.0, f2n,nvir)
49.
50.         eaijk = eorb(a) - ( eorb(ncor+i)+eorb(ncor+j)+eorb(ncor+k) )
51.
52.         !$omp target teams distribute parallel do collapse(2)       &
53.         !$omp     reduction(+:emp4i,emp5i)                         &
54.         !$omp     private(f1nbc,f1ncb,f2nbc,f2ncb)                 &
55.         !$omp     private(t1v1b,t1v2b)                             &
56.         !$omp     private(denom) firstprivate(eaijk,nvir,ncor,nocc)
57.         do b=1,nvir
58.            do c=1,nvir
59.               denom=-1.0/( eorb(ncor+nocc+b)+eorb(ncor+nocc+c)+eaijk )
60.
61.               f1nbc = f1n(b,c);
62.               f1ncb = f1n(c,b);
63.               f2nbc = f2n(b,c);
64.               f2ncb = f2n(c,b);
65.               t1v1b = t1v1(b);
66.               t1v2b = t1v2(b);
```

```
67.
68.                emp4i = emp4i + (denom*t1v1b*f1nbc) + (denom*2*f1ncb)
69.                emp5i = emp5i + (denom*t1v2b*f2nbc) + (denom*3*f2ncb)
70.             enddo
71.           enddo
72.           !$omp end target teams distribute parallel do
73.
74.           emp4 = emp4 + emp4i
75.           emp5 = emp5 + emp5i
76.
77.           end
78.
79.
80.         subroutine init_array_1(arr, m)
81.           implicit none
82.           real, intent(inout) :: arr(m)
83.           integer m, i
84.
85.           do i = 1, m
86.              arr(i) = 1.0/(100.0 + i-1)
87.           end do
88.           end subroutine init_array_1
89.
90.
91.         subroutine init_array_2(arr, m, n)
92.           implicit none
93.           real, intent(inout) :: arr(m, n)
94.           integer m, n, i, j
95.
96.           !$omp target teams distribute parallel do
97.           do i = 1, m
98.              do j = 1, n
99.                 arr(i,j) = 1.0/(100.0 + ((i-1) * n) + j)
100.             end do
101.          end do
102.        end subroutine init_array_2
103.
104.
105.        program main
106.
107.          use omp_lib
108.          use iso_fortran_env
109.          implicit none
110.
111.          interface
112.             subroutine omp_fbody(f1n,f2n,eorb,                    &
113.                          ncor,nocc,nvir, emp4,emp5,a,i,j,k,klo,  &
114.                          Jia, Kia, Tia, Xia, Tkj, Kkj,           &
115.                          t1v1,t1v2)
116.             real, intent(inout) :: emp4,emp5
117.             integer, intent(in) :: ncor,nocc,nvir
118.             integer, intent(in) :: a,i,j,k, klo
119.             real, intent(inout) :: f1n(nvir,nvir)
120.             real, intent(inout) :: f2n(nvir,nvir)
121.             real, intent(in)    :: eorb(*)
122.             real, intent(in)    :: Jia(*), Kia(*), Tia(*), Xia(*), Tkj(*), Kkj(*)
123.             real, intent(in)    :: t1v1(nvir),t1v2(nvir)
124.             end subroutine omp_fbody
125.          end interface
126.
127.          integer :: ncor, nocc, nvir, maxiter, nkpass
128.          integer :: nbf, lnvv, lnov, kchunk
129.          real, allocatable :: eorb(:)
130.          real, allocatable :: f1n(:,:)
131.          real, allocatable :: f2n(:,:)
132.
133.          real, allocatable :: Jia(:)
134.          real, allocatable :: Kia(:)
135.          real, allocatable :: Tia(:)
```

```
136.         real, allocatable :: Xia(:)
137.         real, allocatable :: Tkj(:)
138.         real, allocatable :: Kkj(:)
139.
140.         real, allocatable :: t1v1(:),t1v2(:)
141.         real emp4, emp5
142.         integer :: a, b, c, i, j, k
143.         integer :: klo, khi, iter
144.         double precision, allocatable :: timers(:)
145.         double precision :: t0, t1, tsum, tmax, tmin, tavg
146.
147.!        実行パラメーター
148.         nocc = 256
149.         nvir = 2048
150.         maxiter = 50
151.         nkpass = 1
152.         ncor = 0
153.
154.         print *, "Run parameters:"
155.         print *, "nocc    =", nocc
156.         print *, "nvir    =", nvir
157.         print *, "maxiter =", maxiter
158.         print *, "nkpass  =", nkpass
159.         print *, "ncor    =", ncor
160.         print *, " "
161.
162.!        配列の割り当てと初期化
163.
164.         nbf = ncor + nocc + nvir
165.         lnvv = nvir * nvir
166.         lnov = nocc * nvir
167.         kchunk = (nocc - 1)/nkpass + 1
168.
169.         !$omp allocators allocate(allocator(omp_target_device_mem_alloc): f1n)
170.         allocate( f1n(1:nvir,1:nvir) )
171.
172.         !$omp allocators allocate(allocator(omp_target_device_mem_alloc): f2n)
173.         allocate( f2n(1:nvir,1:nvir) )
174.
175.         !$omp allocators allocate(allocator(omp_target_host_mem_alloc): eorb)
176.         allocate( eorb(1:nbf) )
177.
178.         !$omp allocators allocate(allocator(omp_target_host_mem_alloc): Jia)
179.         allocate( Jia(1:lnvv) )
180.
181.         !$omp allocators allocate(allocator(omp_target_host_mem_alloc): Kia)
182.         allocate( Kia(1:lnvv) )
183.
184.         !$omp allocators allocate(allocator(omp_target_host_mem_alloc): Tia)
185.         allocate( Tia(1:lnov*nocc) )
186.
187.         !$omp allocators allocate(allocator(omp_target_host_mem_alloc): Xia)
188.         allocate( Xia(1:lnov*nocc))
189.
190.         !$omp allocators allocate(allocator(omp_target_host_mem_alloc): Tkj)
191.         allocate( Tkj(1:kchunk*lnvv) )
192.
193.         !$omp allocators allocate(allocator(omp_target_host_mem_alloc): Kkj)
194.         allocate( Kkj(1:kchunk*lnvv) )
195.
196.         !$omp allocators allocate(allocator(omp_target_host_mem_alloc): t1v1)
197.         allocate( t1v1(1:lnvv) )
198.
199.         !$omp allocators allocate(allocator(omp_target_host_mem_alloc): t1v2)
200.         allocate( t1v2(1:lnvv) )
201.!
202.         call init_array_1(eorb, nbf)
203.         call init_array_1(Jia, lnvv)
```

```
204.        call init_array_1(Kia, lnvv)
205.        call init_array_1(Tia, lnov*nocc)
206.        call init_array_1(Xia, lnov*nocc)
207.        call init_array_1(Tkj, kchunk*lnvv)
208.        call init_array_1(Kkj, kchunk*lnov)
209.        call init_array_1(t1v1, lnvv)
210.        call init_array_1(t1v2, lnvv)
211.
212.        call init_array_2(f1n, nvir, nvir)
213.        call init_array_2(f2n, nvir, nvir)
214.
215.        print *, "End of initialization"
216.
217.        allocate (timers(1:maxiter))
218.
219.        emp4=0.0
220.        emp5=0.0
221.        a=1
222.        iter=1
223.
224.        !$omp target data                    &
225.           map(to: eorb)                    &
226.           map(to: Jia, Kia, Tia, Xia)      &
227.           map(to: Tkj, Kkj)                &
228.           map(to: t1v1, t1v2)
229.
230.        do klo = 1, nocc, kchunk
231.           khi = MIN(nocc, klo+kchunk-1)
232.           do j = 1, nocc
233.
234.#if defined(DO_UPDATE_ARRAYS)
235.!             Tkj と Kkj の更新
236.              Tkj((khi-klo+1)*lnvv) = Tkj((khi-klo+1)*lnvv) + 1.0
237.              Kkj((khi-klo+1)*lnov) = Kkj((khi-klo+1)*lnov) + 1.0
238.
239.              !$omp target update to (Tkj, Kkj)
240.#endif
241.
242.              do i = 1, nocc
243.
244.#if defined(DO_UPDATE_ARRAYS)
245.!                Jia、Kia、Tia、Xia 配列の要素を更新
246.                 Jia(lnvv) = Jia(lnvv) + 1.0
247.                 Kia(lnvv) = Kia(lnvv) + 1.0
248.                 Tia(lnov) = Tia(lnov) + 1.0
249.                 Xia(lnov) = Xia(lnov) + 1.0
250.
251.                 !$omp target update to (Jia, Kia, Tia, Xia)
252.#endif
253.
254.                 do k = klo, MIN(khi,i)
255.
256.#if defined(DO_UPDATE_ARRAYS)
257.!                   t1v1 配列の要素を更新
258.                    t1v1(:)= Tkj(lnvv-nvir+1:lnvv)
259.
260.                    !$omp target update to (t1v1)
261.#endif
262.
263.                    t0 = omp_get_wtime()
264.
265.                    call omp_fbody(f1n,f2n,eorb,                    &
266.                              ncor,nocc,nvir, emp4,emp5,a,i,j,k,klo, &
267.                              Jia, Kia, Tia, Xia, Tkj, Kkj,          &
268.                              t1v1,t1v2)
269.
270.                    t1 = omp_get_wtime()
271.                    timers(iter) = (t1-t0)
```

```
272.                     if (iter .eq. maxiter) then
273.                         print *, "Stopping after ", iter, "iterations"
274.                         print *, " "
275.                         goto 150
276.                     endif
277.
278.!                    大きな maxiter の NAN を防止
279.                     if (emp4 >  1000.0) then
280.                         emp4 = emp4 - 1000.0
281.                     endif
282.                     if (emp4 < -1000.0) then
283.                         emp4 = emp4 + 1000.0
284.                     endif
285.                     if (emp5 >  1000.0) then
286.                         emp5 = emp5 - 1000.0
287.                     endif
288.                     if (emp5 < -1000.0) then
289.                         emp5 = emp5 + 1000.0
290.                      endif
291.
292.                     iter = iter + 1
293.
294.                 end do ! k = klo, MIN(khi,i)
295.              end do ! do i = 1, nocc
296.           end do ! do j = 1, nocc
297.        end do ! do klo = 1, nocc, kchunk
298.
299. 150    CONTINUE
300.        !$omp end target data
301.
302.        tsum =  0.0
303.        tmax = -1.0e10
304.        tmin =  1.0e10
305.        do i = 2, iter
306.           tsum = tsum + timers(i)
307.           tmax = MAX(tmax,timers(i))
308.           tmin = MIN(tmin,timers(i))
309.        end do
310.
311.        tavg = tsum / (iter - 1)
312.        print *, "TOTAL ITER: ", iter
313.        write(*, 110) " TIMING: min=", tmin, ", max=", tmax, ", avg of iters after
    first=", tavg, " seconds"
314. 110    format (A, F9.6, A, F9.6, A, F9.6, A)
315.
316.        write(*, 120) " emp4 = ", emp4, " emp5 =", emp5
317. 120    format (A, F15.3, A, F15.3)
318.
319.        print *, "END"
320.
321.        deallocate (f1n)
322.        deallocate (f2n)
323.        deallocate (eorb)
324.        deallocate (Jia)
325.        deallocate (Kia)
326.        deallocate (Tia)
327.        deallocate (Xia)
328.        deallocate (Tkj)
329.        deallocate (Kkj)
330.
331.        deallocate (t1v1)
332.        deallocate (t1v2)
333.        deallocate (timers)
334.
335.        end program
```

## パフォーマンスの比較

次のコマンドで、それぞれのバージョンをコンパイル、リンク、そして実行します。以下の例の `TEST.f`、`TEST.o`、`TEST.exe` は、適切なファイル名に置き換えてください。

```
コンパイル:
$ ifx -O3 -fiopenmp -fopenmp-targets=spir64 -DMKL -qmkl -fpp -free -D DO_UPDATE_ARRAYS
TEST.f -c -o TEST.o

リンク:
$ ifx -fiopenmp -fopenmp-targets=spir64 -qmkl -fsycl -L${MKLROOT}/lib/intel64 -liomp5
-lsycl -lOpenCL -lstdc++ -lpthread -lm -ldl -lmkl_sycl -D DO_UPDATE_ARRAYS TEST.o -o
TEST.exe

実行:
$ export LIBOMPTARGET_LEVEL_ZERO_COMMAND_BATCH=copy
$ export OMP_TARGET_OFFLOAD=MANDATORY
$ export ZE_AFFINITY_MASK=0.0
$ export LIBOMPTARGET_DEBUG=0
$ export LIBOMPTARGET_PLUGIN=level0
$ TEST.exe
```

特定の GPU (1 スタックのみ) で実行した 3 つのバージョンを比較します。

それぞれのバージョンの平均反復時間は次のようになりました。

```
バージョン 1 (test-SharedMem.f): 0.115163 seconds
バージョン 2 (test-Map-UpdateTo.f): 0.002012 seconds
バージョン 3 (test-DeviceMem-Map-UpdateTo.f): 0.001978 seconds
```

## 節: is_device_ptr、use_device_ptr、has_device_addr、use_device_addr

`is_device_ptr`、`use_device_ptr`、`has_device_addr`、および `use_device_addr` 節は、`target`、`target data`、または `dispatch` 構造で参照する変数に関する情報を伝達するために使用します。これらの節の説明を以下に示します。

### is_device_ptr

`is_device_ptr` 節は `target` または `dispatch` ディレクティブで指定されます。リスト項目がデバイスポインターであることを示します。そのため、各リスト項目は構造内でプライベート化され、新しいリスト項目は元のリスト項目を参照するデバイスアドレスで初期化されます。

- C 言語では、各リスト項目はポインターもしくは配列タイプである必要があります。
- C++ 言語では、各リスト項目はポインター、配列、ポインター参照、または配列参照タイプである必要があります。
- Fortran 言語では、各リスト項目は `C_PTR` である必要があります。

次の C/C++ の例では、`is_device_ptr` 節の使い方を示しています。`omp_target_alloc_device` ルーチンは、デバイス上にメモリー領域を確保し、そのメモリー領域のポインターを返してホスト変数 `arr_device` に保存します。`target` ディレクティブで `is_device_ptr(arr_device)` 節を指定して、`arr_device` がデバイスメモリーを指していることを示します。そのため、`target` 構造内では `arr_device` はプライベート化され、`arr_device` が参照するデバイスのアドレスで初期化されます。

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <stdint.h>
4.  #include <math.h>
5.  #include <omp.h>
6.
7.  #define N 100
8.
9.  int main(void)
10. {
11.   int *arr_host = NULL;
12.   int *arr_device = NULL;
13.
14.   arr_host = (int *) malloc(N * sizeof(int));
15.   arr_device = (int *) omp_target_alloc_device(N * sizeof(int),
16.                                         omp_get_default_device());
17.
18.   #pragma omp target is_device_ptr(arr_device) map(from: arr_host[0:N])
19.   {
20.     for (int i = 0; i < N; ++i) {
21.       arr_device[i] = i;
22.       arr_host[i] = arr_device[i];
23.     }
24.   }
25.
26.   printf ("%d, %d, %d \n", arr_host[0], arr_host[N/2], arr_host[N-1]);
27. }
```

## use_device_ptr

use_device_ptr 節は target data ディレクティブで指定されます。各リスト項目が、デバイス上のストレージであるか、デバイス上でアクセスできるオブジェクトへのポインターであることを示します。

リスト項目がデバイスにマップされたオブジェクトへのポインターである場合、構造内のリスト項目への参照は構造内ではローカルで、対応するオブジェクトのデバイスアドレスを参照するデバイスポインターへの参照に変換されます。

リスト項目がマップされたオブジェクトを指していない場合、有効なデバイスアドレスを保持している必要があり、リスト項目への参照は、このデバイスを参照するローカル・デバイス・ポインターへの参照に変換されます。

各リスト項目は、デバイス環境に対応するストレージを持つ、またはターゲットデバイス上でアクセス可能なオブジェクトのアドレスを持つポインターでなければなりません。

C 言語では、各リスト項目はポインターもしくは配列タイプである必要があります。

C++ 言語では、各リスト項目はポインター、配列、ポインター参照、または配列参照タイプである必要があります。

Fortran 言語では、各リスト項目は C_PTR である必要があります。

次の C/C++ の例では、use_device_ptr 節の使い方を示しています。デバイス上にメモリーを割り当てるため、omp_target_alloc_device ルーチンが 3 回呼び出されています。割り当てられたメモリーアドレスは、ホスト上のポインター変数 A、B および C に保持されます。target data ディレクティブに use_device_ptr(A, B, C) 節を指定して、A、B、C が有効なデバイスアドレスを持っていることを明示します。

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <stdint.h>
4.  #include <math.h>
5.  #include <omp.h>
6.
```

```
7.  #define length 65536
8.
9.  int main(void)
10. {
11.   int device_id = omp_get_default_device();
12.   size_t bytes = length*sizeof(double);
13.   double * __restrict A;
14.   double * __restrict B;
15.   double * __restrict C;
16.   double scalar = 3.0;
17.   double ar;
18.   double br;
19.   double cr;
20.   double asum;
21.
22.   // デバイスメモリーに配列を割り当て
23.
24.   A = (double *) omp_target_alloc_device(bytes, device_id);
25.   if (A == NULL){
26.      printf(" ERROR: Cannot allocate space for A using
    omp_target_alloc_device().\n");
27.      exit(1);
28.   }
29.
30.   B = (double *) omp_target_alloc_device(bytes, device_id);
31.   if (B == NULL){
32.      printf(" ERROR: Cannot allocate space for B using
    omp_target_alloc_device().\n");
33.      exit(1);
34.   }
35.
36.   C = (double *) omp_target_alloc_device(bytes, device_id);
37.   if (C == NULL){
38.      printf(" ERROR: Cannot allocate space for C using
    omp_target_alloc_device().\n");
39.      exit(1);
40.   }
41.
42.   #pragma omp target data use_device_ptr(A,B,C)
43.   {
44.       // 配列を初期化
45.
46.       #pragma omp target teams distribute parallel for
47.       for (size_t i=0; i<length; i++) {
48.           A[i] = 2.0;
49.           B[i] = 2.0;
50.           C[i] = 0.0;
51.       }
52.
53.       // 計算を実行
54.
55.       #pragma omp target teams distribute parallel for
56.       for (size_t i=0; i<length; i++) {
57.           C[i] += A[i] + scalar * B[i];
58.       }
59.
60.       // 結果を評価して出力
61.
62.       ar = 2.0;
63.       br = 2.0;
64.       cr = 0.0;
65.       for (int i=0; i<length; i++) {
66.           cr += ar + scalar * br;
67.       }
68.
69.       asum = 0.0;
70.       #pragma omp target teams distribute parallel for reduction(+:asum)
```

```
71.       for (size_t i=0; i<length; i++) {
72.           asum += fabs(C[i]);
73.       }
74.
75.   } // end target data
76.
77.   omp_target_free(A, device_id);
78.   omp_target_free(B, device_id);
79.   omp_target_free(C, device_id);
80.
81.   double epsilon=1.e-8;
82.   if (fabs(cr - asum)/asum > epsilon) {
83.       printf("Failed Validation on output array\n"
84.              "       Expected checksum: %lf\n"
85.              "       Observed checksum: %lf\n"
86.              "ERROR: solution did not validate\n", cr, asum);
87.       return 1;
88.   } else {
89.       printf("Solution validates. Checksum = %lf\n", asum);
90.   }
91.
92.   return 0;
93. }
```

## has_device_addr

has_device_addr 節は target ディレクティブで指定されます。これは、リスト項目がすでに有効なデバイスアドレスを保持しており、デバイスから直接アクセスできることを示します。

各リスト項目は、デバイスデータ環境の有効なデバイスアドレスを持っている必要があります。配列セクションを含むすべてのタイプに使用できます。

has_device_addr 節は Fortran では特に有用であり、C_PTR に限らずあらゆるタイプのリスト項目で使用でき、リスト項目がデバイスアドレスであることを明示できます。

次の Fortran コードは、has_device_addr 節の使い方を示します。この例では、3 つの配列 A、B および C をデバイス上に割り当てています。has_device_addr 領域で配列が参照される場合、has_device_addr(A, B, C) 節を指定して A、B、C がすでにデバイスアドレスを持っていることを明示します。

```
1.  program main
2.    use iso_fortran_env
3.    use omp_lib
4.    implicit none
5.
6.    integer, parameter :: iterations=1000
7.    integer, parameter :: length=64*1024*1024
8.    real(kind=REAL64), parameter ::  epsilon=1.D-8
9.    real(kind=REAL64), allocatable ::  A(:)
10.   real(kind=REAL64), allocatable ::  B(:)
11.   real(kind=REAL64), allocatable ::  C(:)
12.   real(kind=REAL64) :: scalar=3.0
13.   real(kind=REAL64) :: ar, br, cr, asum
14.   real(kind=REAL64) :: nstream_time, avgtime
15.   integer :: i, iter
16.
17.   !
18.   ! デバイスメモリーに配列を割り当て
19.
20.   !$omp allocators allocate(allocator(omp_target_device_mem_alloc): A)
21.   allocate(A(length))
22.
23.   !$omp allocators allocate(allocator(omp_target_device_mem_alloc): B)
24.   allocate(B(length))
```

```
25.
26.   !$omp allocators allocate(allocator(omp_target_device_mem_alloc): C)
27.   allocate(C(length))
28.
29.   !
30.   ! 配列の初期化
31.
32.   !$omp target teams distribute parallel do has_device_addr(A, B, C)
33.   do i = 1, length
34.       A(i) = 2.0
35.       B(i) = 2.0
36.       C(i) = 0.0
37.   end do
38.
39.   !
40.   ! 計算を実行
41.
42.   nstream_time = omp_get_wtime()
43.   do iter = 1, iterations
44.      !$omp target teams distribute parallel do has_device_addr(A, B, C)
45.      do i = 1, length
46.         C(i) = C(i) + A(i) + scalar * B(i)
47.      end do
48.   end do
49.   nstream_time = omp_get_wtime() - nstream_time
50.
51.   !
52.   ! 結果を評価して出力
53.
54.   ar = 2.0
55.   br = 2.0
56.   cr = 0.0
57.   do iter = 1, iterations
58.      do i = 1, length
59.         cr = cr + ar + scalar * br
60.      end do
61.   end do
62.
63.   asum = 0.0
64.   !$omp target teams distribute parallel do reduction(+:asum) has_device_addr(C)
65.   do i = 1, length
66.      asum = asum + abs(C(i))
67.   end do
68.
69.   if (abs(cr - asum)/asum > epsilon) then
70.      print *, "Failed Validation on output array:", "Expected =", cr, "Observed =",
    asum
71.   else
72.      avgtime = nstream_time/iterations
73.      print *, "Solution validates:", "Checksum =", asum, "Avg time (s) =",  avgtime
74.   endif
75.
76.   deallocate(A)
77.   deallocate(B)
78.   deallocate(C)
79.
80. end program main
```

### use_device_addr

`use_device_addr` 節は `target data` ディレクティブで指定されます。リスト項目がデバイス上のストレージであるか、またデバイス上でアクセス可能であることを示します。

リスト項目がマップされている限り、構造内のリスト項目の参照は対応するリスト項目への参照に変換されます。リスト項目がマップされていない場合、デバイス上でアクセス可能であるとみなされます。

リスト項目は配列セクションであってもかまいません。

has_device_addr と同様に、use_device_addr 節は Fortran では特に有用であり、C_PTR に限らずあらゆるタイプのリスト項目で使用でき、リスト項目がデバイスアドレスであることを明示できます。

次の Fortran コードは、use_device_addr 節の使い方を示します。この例では、array_d が alloc マップタイプでデバイスにマップされているため、デバイス上に array_d のストレージが確保されますが、ホストとデバイス間でデータ転送は発生しません。target data ディレクティブで use_device_addr(array_d) 節を指定し、array_d がデバイス上にストレージを持っていることを示します。

```
1.  program target_use_device_addr
2.
3.    use omp_lib
4.    use iso_fortran_env, only : real64
5.    implicit none
6.
7.    integer, parameter :: N1 = 1024
8.    real(kind=real64), parameter :: aval = real(42, real64)
9.    real(kind=real64), allocatable :: array_d(:), array_h(:)
10.   integer :: i,err
11.
12.   ! Allocate host data
13.   allocate(array_h(N1))
14.
15.   !$omp target data map (from:array_h(1:N1)) map(alloc:array_d(1:N1))
16.   !$omp target data use_device_addr(array_d)
17.   !$omp target
18.       do i=1, N1
19.         array_d(i) = aval
20.         array_h(i) = array_d(i)
21.      end do
22.   !$omp end target
23.   !$omp end target data
24.   !$omp end target data
25.
26.   ! 結果をチェック
27.   write (*,*) array_h(1), array_h(N1)
28.   if (any(array_h /= aval)) then
29.     err = 1
30.   else
31.     err = 0
32.   end if
33.
34.   deallocate(array_h)
35.   if (err == 1) then
36.     stop 1
37.   else
38.     stop 0
39.   end if
40.
41. end program target_use_device_addr
```

次の表は、このセクションで説明する節の特性をまとめたものです。

| 節 | ディレクティブ | リスト項目のタイプ | 説明 |
|---|---|---|---|
| is_device_ptr | target、dispatch | C/C++: ポインター、配列または参照<br>Fortran: C_PTR | リスト項目がデバイスポインターであることを示します (有効なデバイスアドレスを持つ)。 |
| use_device_ptr | target data | C/C++:ポインター、配列または参照<br>Fortran: C_PTR | リスト項目がデバイス上のストレージであるか、デバイス上でアクセスできるオブジェクトへのポインターであることを示します。 |
| has_device_addr | target | 任意のタイプ (配列セクションでも可能) | リスト項目が有効なデバイスアドレスを持つことを示します。 |
| use_device_addr | target data | 任意のタイプ (配列セクションでも可能) | リスト項目がデバイス上のストレージであるか、デバイス上でアクセス可能であることを示します。 |

## プリフェッチ

ユーザーガイドに示されるデータ・プリフェッチ方法は、下位レベルのキャッシュミスやメインメモリーのアクセスから生じるレイテンシーを隠匿するのに役立ちます。インテル® GPU 向けの OpenMP* オフロードでは、次の構文で prefetch プラグマを指定することでこの機能を有効にできます。

**C OpenMP* プリフェッチ:**

```
#pragma ompx prefetch data([prefetch-hint-modifier:],arrsect,[,arrsect])[if
(condition)]
```

**Fortran OpenMP* プリフェッチ:**

```
!$omp prefetch data([prefetch-hint-modifier:] arrsect [,arrsect])[if(condition)]
```

この prefetch プラグマはインテルの拡張機能であり、インテル® データセンター GPU マックス・シリーズ以降の製品で機能します。プラグマのパラメーターは次のとおりです。

- **Prefetch-hint:** プリフェッチされたデータの格納先は、オプションの prefetch-hint-modifier で指定します。有効な値は、0 (プリフェッチしない)、2 (L2 のみにプリフェッチ) および 4 (L1 と L2 にプリフェッチ) です。値が省略されるとデフォルト値は 0 です。
- **配列セクション:** C では OpenMP* 構文 [lower-bound : length] を使用し、Fortran では (lower-bound: upper-bound) を使用して、連続する配列セクション arrsect を指定します。例えば、a[10] または a(10) で始まる 4 つの要素をプリフェッチするには、C では a[10:4] を、Fortran では a(10:13) を使用します。このドキュメントの執筆時点では、コンパイラーは単一要素のプリフェッチを要求することに注意してください。したがって、上記の例は、C の a[10:1]、Fortran の a(10:10) に相当します。
- **デフォルト・プリフェッチ・サイズ:** 配列の単一要素のプリフェッチが要求された場合でも、ハードウェアはその要素を含むキャッシュライン全体をプリフェッチします。インテル® データセンター GPU マックス・シリーズのキャッシュラインのサイズは 64 バイトです。
- **フォールト:** インテル® データセンター GPU マックス・シリーズのプリフェッチ命令でエラーが発生し、無効なアドレスへのアクセスによりセグメンテーション・フォールトが発生する可能性があることを意味します。プラグマの if 節に condition を指定して、範囲外のアクセスを保護できます。
- **非ブロック:** prefetch プラグマは実行をブロックせず、プリフェッチが完了するのを待機しません。

## C OpenMP* のプリフェッチ

次の例は、N 体フォースカーネルを簡略化した 1 次元バージョンを示しています。外部 `for` ループは、力が計算される粒子に対し反復します。内部ループは、`TILE_SIZE` 粒子のバッチで相互に作用する粒子を反復処理します。現在の粒子スタックの計算中に、次の粒子スタックをプリフェッチできます。前述したように、プリフェッチは常に 64 バイトを読み取ります。したがって、`if ( (next_tile % 16) == 0 )` によって、16 個の単精度浮動小数点要素ごとに 1 回プリフェッチを発行します。このマスク条件が常に有効であるとは限りません。以下のコード例の後にある追加の注意事項を参照してください。`prefetch hint` には 4 を使用します (L1 と L2 キャッシュへプリフェッチ)。以下には、オフロードカーネルのみを示します。

```
1.  #define WORKGROUP_SIZE 1024
2.  #define PREFETCH_HINT 4 // 4 = L1 と L3 へプリフェッチ; 2 = L3 へプリフェッチ
3.  #define TILE_SIZE 64
4.
5.  void nbody_1d_gpu(float *c, float *a, float *b, int n1, int n2) {
6.  #pragma omp target teams distribute parallel for thread_limit(WORKGROUP_SIZE)
7.    for (int i = 0; i < n1; i++) {
8.      const float ma0 = 0.269327f, ma1 = -0.0750978f, ma2 = 0.0114808f;
9.      const float ma3 = -0.00109313f, ma4 = 0.0000605491f, ma5 = -0.00000147177f;
10.     const float eps = 0.01f;
11.
12.     float dx = 0.0;
13.     float bb[TILE_SIZE];
14.     for (int j = 0; j < n2; j += TILE_SIZE) {
15.       // タイルを b からロード
16.       for (int u = 0; u < TILE_SIZE; ++u) {
17.         bb[u] = b[j + u];
18. #ifdef PREFETCH
19.         int next_tile = j + TILE_SIZE + u;
20.         if ((next_tile % 16) == 0) {
21. #pragma ompx prefetch data(PREFETCH_HINT : b[next_tile]) if (next_tile < n2)
22.         }
23. #endif
24.       }
25. #pragma unroll(TILE_SIZE)
26.       for (int u = 0; u < TILE_SIZE; ++u) {
27.         float delta = bb[u] - a[i];
28.         float r2 = delta * delta;
29.         float s0 = r2 + eps;
30.         float s1 = 1.0f / sqrtf(s0);
31.         float f =
32.             (s1 * s1 * s1) -
33.             (ma0 + r2 * (ma1 + r2 * (ma2 + r2 * (ma3 + r2 * (ma4 + ma5)))));
34.         dx += f * delta;
35.       }
36.     }
37.     c[i] = dx * 0.23f;
38.   }
39. }
```

条件 `if ( (next_tile % 16) == 0 )` では、配列インデックスがベクトル化されていない場合、プリフェッチのオーバーヘッドを軽減できます。上記の例では、インデックス `i` のみがベクトル化されるため、`j` によってインデックス付けされた `b[]` をプリフェッチする場合に、16 要素ごとに一度だけプリフェッチを発行します。一方、インデックス `i` によって配列をプリフェッチすると、プリフェッチはベクトル化されるため、マスクの条件は何の利点ももたらしません。ユーザーは、アプリケーションに最適なアプローチを実験的に決定する必要があります。

**コンパイルコマンド**

プリフェッチなし:

```
$ icpx -O3 -g -fiopenmp -fopenmp-targets=spir64 -mcmodel=medium nbody_c.cpp -o test_c
```

プリフェッチあり:

```
$ icpx -O3 -g -fiopenmp -fopenmp-targets=spir64 -mcmodel=medium -DPREFETCH nbody_c.cpp
-o test_c
```

**実行コマンド**

```
$ export LIBOMPTARGET_LEVEL0_COMPILATION_OPTIONS="-cl-strict-aliasing -cl-fast-
relaxed-math"
$ export ZE_AFFINITY_MASK=0.0
$ export LIBOMPTARGET_PLUGIN_PROFILE=T,usec
$ export GC_ForceOCLSIMDWidth=16 .
$ /test_c
```

デフォルトの SIMD 幅は、インテル® データセンター GPU マックス・シリーズ向けのバックエンド・デバイス・コンパイラー (インテル® グラフィックス・コンパイラーまたは IGC) によって、カーネルのレジスター・プレッシャーなどの要因を考慮したコンパイラーのヒューリスティックによって 16 または 32 に自動設定されます。IGC 環境変数 `IGC_ForceOCLSIMDWidth=16` を使用して、IGC コンパイラーに SIMD 幅 16 を強制できます。SIMD16 に設定することで、上記カーネルのパフォーマンスを向上できました。コマンドの実行時に、`LIBOMPTARGET_PLUGIN_PROFILE=T,usec` を設定して、OpenMP* ビルトイン・プロファイラーを有効にします。プリフェッチなしで実行した出力は次のとおりです。

```
Obtained output = 222700231.430
Expected output = 222700339.016

Total time =    205.4 milliseconds
================================================================================================
================================
LIBOMPTARGET_PLUGIN_PROFILE(LEVEL0) for OMP DEVICE(0) Intel(R) Graphics [0x0bd6],
Thread 0
------------------------------------------------------------------------------------------------
--------------------------------
Kernel 0                  : __omp_offloading_46_3c0d785c__Z12nbody_1d_gpuPfS_S_ii_l115
Kernel 1                  : __omp_offloading_46_3c0d785c__Z15clean_cache_gpuPdi_l69
Kernel 2                  : __omp_offloading_46_3c0d785c__Z4main_l98
------------------------------------------------------------------------------------------------
--------------------------------
                          : Host Time (usec)                        Device Time (usec)
Name                      :      Total   Average       Min       Max     Total
Average       Min       Max     Count
------------------------------------------------------------------------------------------------
--------------------------------
Compiling                 :  598283.05 598283.05 598283.05 598283.05      0.00
0.00      0.00      0.00      1.00
DataAlloc                 :    9578.23    798.19      0.00   8728.03      0.00
0.00      0.00      0.00     12.00
DataRead (Device to Host) :      77.01     77.01     77.01     77.01      5.68
5.68      5.68      5.68      1.00
DataWrite (Host to Device):     713.11    356.55    179.05    534.06     15.76
7.88      5.04     10.72      2.00
Kernel 0                  :  205292.22   2052.92   2033.95   2089.98 203572.32
2035.72   1984.96   2073.12    100.00
Kernel 1                  :  109194.28   1091.94   1076.94   1681.09 107051.52
1070.52   1062.40   1107.04    100.00
Kernel 2                  :    1746.89   1746.89   1746.89   1746.89      3.84
3.84      3.84      3.84      1.00
```

```
Linking                    :       0.00      0.00      0.00      0.00      0.00
0.00      0.00      0.00      1.00
OffloadEntriesInit         :    2647.88   2647.88   2647.88   2647.88      0.00
0.00      0.00      0.00      1.00
================================================================================
================================
```

上記の出力では、GPU カーネル実行 (`Kernel 0`) の平均デバイス時間は 2036 マイクロ秒です。プリフェッチを有効にしてバイナリーを実行すると、平均カーネルデバイス時間は 1841 マイクロ秒になりました。

```
Obtained output = 222700231.430
Expected output = 222700339.016

Total time =    185.9 milliseconds
================================================================================
================================
LIBOMPTARGET_PLUGIN_PROFILE(LEVEL0) for OMP DEVICE(0) Intel(R) Graphics [0x0bd6],
Thread 0
--------------------------------------------------------------------------------
--------------------------------
Kernel 0                   : __omp_offloading_43_3c0d785c__Z12nbody_1d_gpuPfS_S_ii_l15
Kernel 1                   : __omp_offloading_43_3c0d785c__Z15clean_cache_gpuPdi_l69
Kernel 2                   : __omp_offloading_43_3c0d785c__Z4main_l98
--------------------------------------------------------------------------------
--------------------------------
                           : Host Time (usec)                        Device Time (usec)
Name                       :      Total   Average       Min       Max     Total
Average       Min       Max     Count
--------------------------------------------------------------------------------
--------------------------------
Compiling                  :  499351.98 499351.98 499351.98 499351.98      0.00
0.00      0.00      0.00      1.00
DataAlloc                  :    9609.94    800.83      0.00   8740.19      0.00
0.00      0.00      0.00     12.00
DataRead (Device to Host) :      77.01     77.01     77.01     77.01      4.96
4.96      4.96      4.96      1.00
DataWrite (Host to Device):     722.17    361.08    185.01    537.16     16.40
8.20      5.44     10.96      2.00
Kernel 0                   :  185793.88   1857.94   1839.88   1919.03 184075.20
1840.75   1824.00   1874.56    100.00
Kernel 1                   :  109442.95   1094.43   1076.94   1590.01 107334.56
1073.35   1062.40   1115.68    100.00
Kernel 2                   :    1821.99   1821.99   1821.99   1821.99      3.84
3.84      3.84      3.84      1.00
Linking                    :       0.00      0.00      0.00      0.00      0.00
0.00      0.00      0.00      1.00
OffloadEntriesInit         :    2493.14   2493.14   2493.14   2493.14      0.00
0.00      0.00      0.00      1.00
================================================================================
================================
```

達成されるパフォーマンスは、利用するハードウェアとソフトウェア・スタックに依存するため、ユーザー環境では異なるパフォーマンス値となる可能性があることに留意してください。

## Fortran OpenMP* のプリフェッチ

同じ nbody1d カーネルを Fortran で見てみます。prefetch ディレクティブは、PREFETCH_HINT 4 で C の例と同じ位置に挿入され、16 要素ごとに 1 回のプリフェッチを行います。

```
1.  #define WORKGROUP_SIZE 1024
2.  #define PREFETCH_HINT 4     ! 4 = prefetch to L1 and L3; 2 = prefetch to L3
3.  #define TILE_SIZE 64
4.
5.        module gpu_kernels
6.        contains
7.        subroutine nbody_1d_gpu(c, a, b, n1, n2)
8.        implicit none
9.        integer n1, n2
10.       real a(0:n1-1), b(0:n2-1), c(0:n1-1)
11.       real dx, bb(0:TILE_SIZE-1), delta, r2, s0, s1, f
12.       integer i,j,u,next
13.       real ma0, ma1, ma2, ma3, ma4, ma5, eps
14.       parameter (ma0=0.269327, ma1=-0.0750978, ma2=0.0114808)
15.       parameter (ma3=-0.00109313, ma4=0.0000605491, ma5=-0.00000147177)
16.       parameter (eps=0.01)
17.
18. !$omp target teams distribute parallel do thread_limit(WORKGROUP_SIZE)
19. !$omp& private(i,dx,j,u,bb,next,delta,r2,s0,s1,f)
20.       do i = 0, n1-1
21.         dx = 0.0
22.         do j = 0, n2-1, TILE_SIZE
23.           ! タイルを b からロード
24.           do u = 0, TILE_SIZE-1
25.             bb(u) = b(j+u)
26. #ifdef PREFETCH
27.             next = j + TILE_SIZE + u
28.             if (mod(next,16).eq.0) then
29. !$omp prefetch data(PREFETCH_HINT:b(next:next))if(next<n2)
30.             endif
31. #endif
32.           enddo
33.           ! 計算
34.           !DIR$ unroll(TILE_SIZE)
35.           do u = 0, TILE_SIZE-1
36.             delta = bb(u) - a(i)
37.             r2 = delta*delta
38.             s0 = r2 + eps
39.             s1 = 1.0 / sqrt(s0)
40.             f = (s1*s1*s1)-(ma0+r2*(ma1+r2*(ma2+r2*(ma3+r2*(ma4+ma5)))))
41.             dx = dx + f*delta
42.           enddo
43.         enddo
44.         c(i) = dx*0.23
45.       enddo
46.       end subroutine
```

### コンパイルコマンド

プリフェッチなし:

```
$ ifx -O3 -g -fiopenmp -fopenmp-targets=spir64 -fpconstant -fpp -ffast-math -fno-sycl-
instrument-device-code -mcmodel=medium nbody_f.f -o test_f
```

プリフェッチあり:

```
$ ifx -O3 -g -fiopenmp -fopenmp-targets=spir64 -fpconstant -fpp -ffast-math -fno-sycl-
instrument-device-code -mcmodel=medium -DPREFETCH nbody_f.f -o test_f
```

**実行コマンド**

```
$ export LIBOMPTARGET_LEVEL0_COMPILATION_OPTIONS="-cl-strict-aliasing -cl-fast-
relaxed-math"
$ export ZE_AFFINITY_MASK=0.0
$ export LIBOMPTARGET_PLUGIN_PROFILE=T,usec
$ export IGC_ForceOCLSIMDWidth=16
$ ./test_f
```

C の例と類似しているため、ここでは出力を示しません。プリフェッチを使用しない場合と使用した場合の平均カーネル時間は、それぞれ 2017µs と 1923µs でした。繰り返しますが、実際のハードウェアと利用するソフトウェア・スタックによって、異なるパフォーマンス値となる可能性を忘れないでください。

## C OpenMP* SIMD のプリフェッチ

OpenMP* オフロードは、インテル® データセンター GPU マックス・シリーズの 16 または 32 SIMD レーンを構成する EU スレッドで、すべての計算を指定する SIMD プログラミング・モデルをサポートしています。これに対応して、OpenMP* の `thread_limit()` 節の意味も変わり、work-group ごとの EU スレッド数を指定するようになりました。OpenMP* SIMD バージョンの `nbody1d` カーネルを以下に示します。`VECLEN=16` で SIMD 幅を明示的に指定する必要があります。現時点では、`prefetch` プラグマは `simd` 句の範囲外で指定することを推奨します。これは、1 つの SIMD レーンのみがプリフェッチを発行することを意図しています。この例では、16 レーンのうち 1 レーンがプリフェッチを発行します。そのため、前述した `if ( (next_tile % 16) == 0 )` 節はここでは必要ありません。

```
1.  #define WORKGROUP_SIZE 1024
2.  #define PREFETCH_HINT 4 // 4 = L1 と L3 へプリフェッチ; 2 = L3 へプリフェッチ
3.  #define TILE_SIZE 64
4.
5.  void nbody_1d_gpu(float *c, float *a, float *b, int n1, int n2) {
6.  #pragma omp target teams distribute parallel for thread_limit(WORKGROUP_SIZE / \
7.                                                                  VECLEN)
8.    for (int i = 0; i < n1; i += VECLEN) {
9.      const float ma0 = 0.269327f, ma1 = -0.0750978f, ma2 = 0.0114808f;
10.     const float ma3 = -0.00109313f, ma4 = 0.0000605491f, ma5 = -0.00000147177f;
11.     const float eps = 0.01f;
12.
13.     float dx[VECLEN];
14.     float aa[VECLEN], bb[TILE_SIZE];
15. #pragma omp simd simdlen(VECLEN)
16. #pragma unroll(0)
17.     for (int v = 0; v < VECLEN; ++v) {
18.       dx[v] = 0.0f;
19.       aa[v] = a[i + v];
20.     }
21.     for (int j = 0; j < n2; j += TILE_SIZE) {
22.       // タイルを b からロード
23.       for (int u = 0; u < TILE_SIZE; u += VECLEN) {
24. #pragma omp simd simdlen(VECLEN)
25. #pragma unroll(0)
26.         for (int v = 0; v < VECLEN; ++v)
27.           bb[u + v] = b[j + u + v];
28. #ifdef PREFETCH
29.         int next_tile = j + TILE_SIZE + u;
30. #pragma ompx prefetch data(PREFETCH_HINT : b[next_tile]) if (next_tile < n2)
31. #endif
32.       }
33. // 現在のタイルを計算
34. #pragma omp simd simdlen(VECLEN)
35. #pragma unroll(0)
36.       for (int v = 0; v < VECLEN; ++v) {
37. #pragma unroll(TILE_SIZE)
38.         for (int u = 0; u < TILE_SIZE; ++u) {
```

```
39.           float delta = bb[u] - aa[v];
40.           float r2 = delta * delta;
41.           float s0 = r2 + eps;
42.           float s1 = 1.0f / sqrtf(s0);
43.           float f =
44.               (s1 * s1 * s1) -
45.               (ma0 + r2 * (ma1 + r2 * (ma2 + r2 * (ma3 + r2 * (ma4 + ma5)))));
46.           dx[v] += f * delta;
47.         }
48.       }
49.     }
50. #pragma omp simd simdlen(VECLEN)
51. #pragma unroll(0)
52.     for (int v = 0; v < VECLEN; ++v) {
53.       c[i + v] = dx[v] * 0.23f;
54.     }
55.   }
56. }
```

**コンパイルコマンド**

SIMD プログラミング・モデルを有効にするには、`-fopenmp-target-simd` コンパイルオプションを追加する必要があります。以下にコンパイルコマンドを示します。

プリフェッチなし:

```
$ icpx -O3 -g -fiopenmp -fopenmp-targets=spir64 -mcmodel=medium -fopenmp-target-simd
nbody_c_simd.cpp -o test_c_simd
```

プリフェッチあり:

```
$ icpx -O3 -g -fiopenmp -fopenmp-targets=spir64 -mcmodel=medium -DPREFETCH -fopenmp-
target-simd nbody_c_simd.cpp -o test_c_simd
```

**実行コマンド**

```
$ export LIBOMPTARGET_LEVEL0_COMPILATION_OPTIONS="-cl-strict-aliasing -cl-fast-
relaxed-math"
$ export ZE_AFFINITY_MASK=0.0
$ export LIBOMPTARGET_PLUGIN_PROFILE=T,usec
$ ./test_c_simd
```

ソースコードで SIMD 幅が明示的に指定されているため、環境変数 `IGC_ForceOCLSIMDWidth=16` は必要ありません。

出力は前の例と類似するためここでは示しません。プリフェッチを使用しない場合と使用した場合の平均カーネル時間は、それぞれ 2008µs と 1810µs でした。繰り返しますが、実際のハードウェアと利用するソフトウェア・スタックによって、異なるパフォーマンス値となる可能性を忘れないでください。

# マルチ GPU、マルチスタック、およびマルチ C スライス・アーキテクチャーとプログラミング

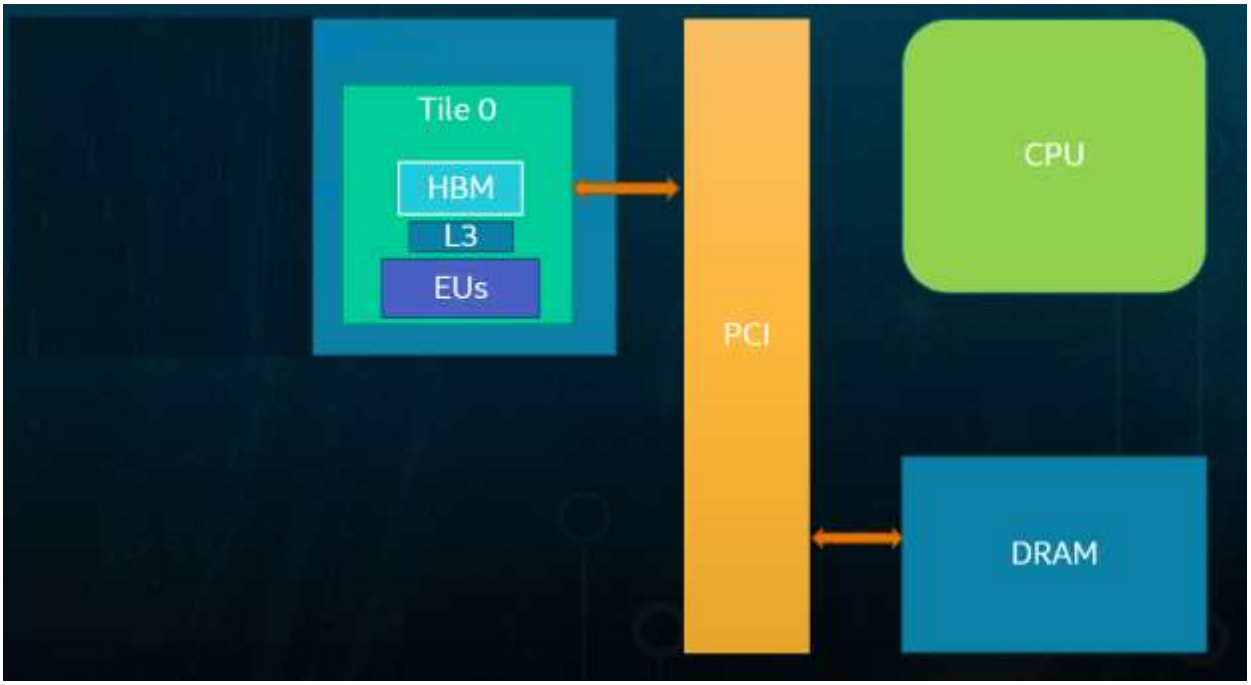
インテル® データセンター GPU マックス・シリーズは、1 から 2 スタックのマルチスタック GPU アーキテクチャーを採用しています。この新しい GPU アーキテクチャー製品により、マルチ GPU、マルチスタック、およびマルチ C スライス計算が可能になります。

この章では、マルチ GPU、マルチスタック、およびマルチ C スライスのヘテロジニアス・コンピューティングに関する次のトピックを説明します。

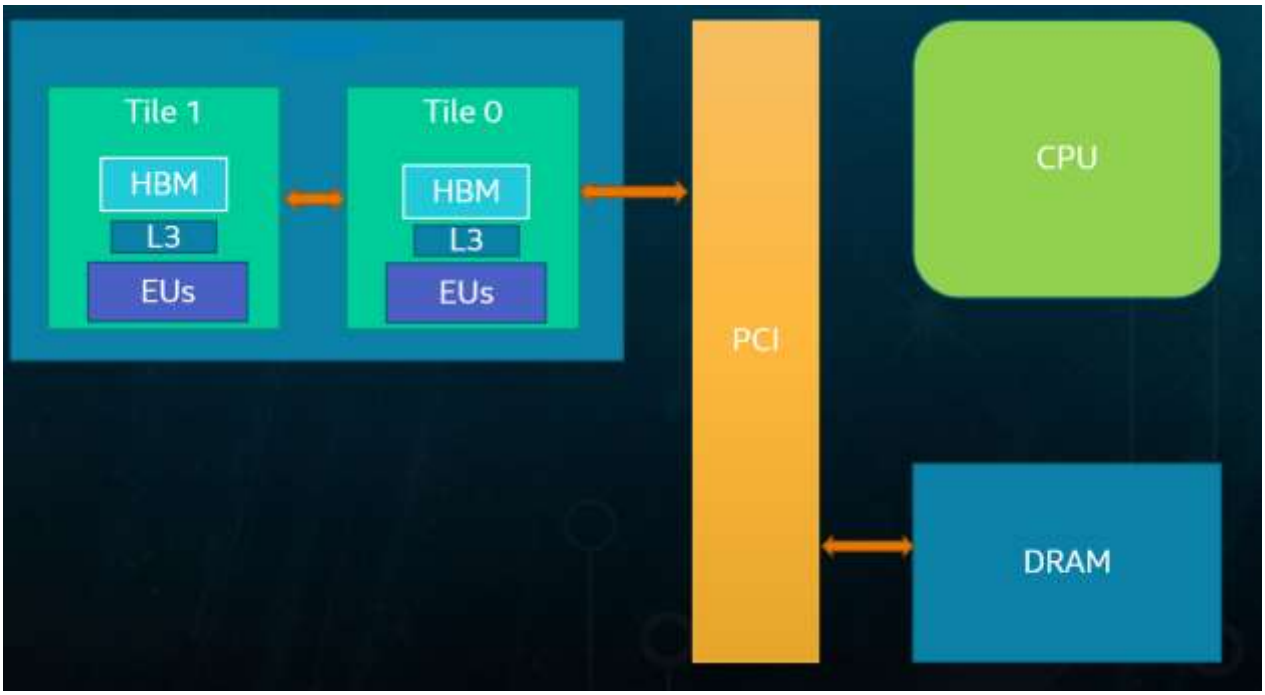

- マルチスタック GPU アーキテクチャー
- GPU メモリーシステム
- 暗黙のスケーリング
- 明示的なスケーリング
- GPU クラスター向けのインテル® MPI
- 用語


## マルチスタック GPU アーキテクチャー

インテル® データセンター GPU マックス・シリーズは、1 から 2 スタックのマルチスタック GPU アーキテクチャーを採用しています。



### インテル® Iris® Xᵉ GPU マルチスタック・アーキテクチャー




この図は、1 スタックと 2 スタックのインテル® データセンター GPU マックス・シリーズ製品を示しており、それぞれに専用のリソースがあります。

- **ベクトルエンジン (VE):** 計算ユニットはスタックに属します。
- **高帯域幅メモリー (HBM):** HBM はスタックに直接接続されます。
- **レベル 2 キャッシュ (L2):** レベル 2 キャッシュはスタックに属します。

一般的なアプリケーションでは、マルチスタック GPU は単一の GPU デバイスとして認識されます。アプリケーションでは、GPU が小さなスタックで内部構成されていることを意識しません。これにより、プログラミング・モデルは簡素化され、コードを変更することなく既存のアプリケーションを実行できます。インテル® GPU ドライバー、SYCL*、および OpenMP* の並列言語ランタイムが、連携してスタック間にワークロードを自動的にディスパッチします。

スタックは、HBM を介してスタック間の効率良い通信を可能にする高速インターコネクトで接続されています。次の操作が考えらえます。

- **任意のスタックが、任意の HBM メモリーに対し読み取り書き込みが可能**
  例えば、スタック 0 はスタック 1 のローカル HBM メモリーを読み取ることができます。この場合、スタック 0 とスタック 1 間のインターコネクトが通信に使用されます。

- **スタック 0 は PCIe* に接続され、どのスタックでもシステムメモリーを読み書き可能**
  データ転送には、同じスタック間相互接続が利用されます。つまり、スタック 0 は、すべてのスタックにおいて、システムメモリーへの最短パスを持ちます。

  システムメモリーの読み書きは CPU の補助を必要とせず、GPU は PCIe* 経由でシステムメモリーへの DMA (ダイレクト・メモリー・アクセス) を実行できます。

- **各スタックは独立したエンティティー**
  スタックは単独でワークロードを実行できます。

スタックのローカル HBM アクセスはスタック間相互接続を必要としないため、クロススタック HBM アクセスよりも効率良く低レイテンシーであり、スタック間の帯域幅消費が少なくなります。熟練者は、メモリーの局所性を活用してより高いパフォーマンスを実現できます。

マルチスタック GPU を適切に利用するため、2 つのアプリケーション・プログラミング・モードが用意されています。

- **暗黙のスケーリング・モード**
  ドライバーと言語ランタイムは、ワークの分散とマルチスタック・メモリーの配置を行います。アプリケーションは GPU を単一のモノリシック・デバイスと見なし、マルチスタック・アーキテクチャーを意識することはありません。

- **明示的なスケーリング・モード**
  ユーザーが、ワークの分散とマルチスタック・メモリーの配置を行います。ドライバーと言語ランタイムは、各スタックを個別のサブデバイスとして公開するツールを提供し、ほかのスタックとは独立してプログラミングできます。

どちらのモードでもドライバーの実行を制御してデバイス利用します。次の 2 つのデバイスタイプがあります。

- **ルートデバイス**
  サブデバイス (スタック) を含むマルチスタック・デバイスです。

- **サブデバイス**
  各スタックに相当します。

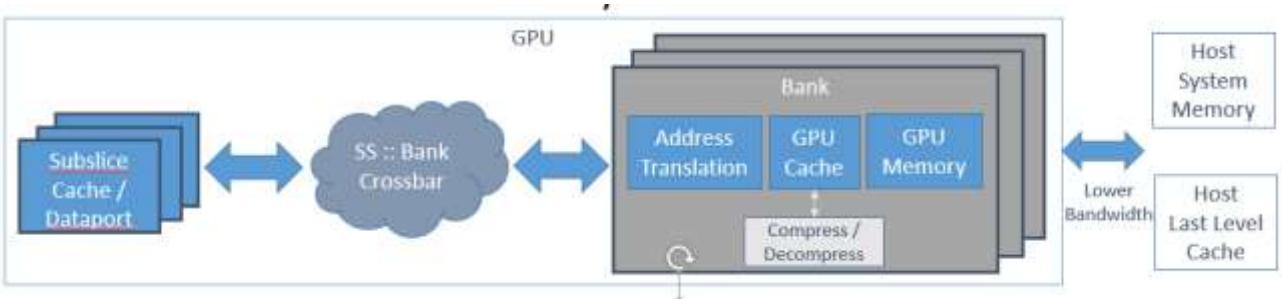
ドライバーと言語ランタイムは、デバイスを抽象化してコマンドのディスパッチとメモリーリソースの割り当てを行います。ルートデバイスとサブデバイスはともに実際のデバイスであるため、各モードでデバイスの意味合いを切り替えるだけで、同じコードを異なるモードのターゲットにできます。

## GPU メモリーシステム

汎用エンジンのメモリーは、図に示すようにホスト側のメモリーとデバイス側のメモリーに分離され、統合共有メモリー (USM) を使用してそれらの間でオブジェクトを移動します。各アドレスは一意のバンクにハッシュされます。シーケンシャル、ストライド、ランダムアドレスの一様分布の近似。

- $X^e$ コアと同じバンク数では全帯域幅 (キューイングの密集、ホットスポット、バンクの密集により減少)。
- 仮想アドレス変換の TLB ミスにより、メモリー・レイテンシーが増加します。さらに多くのメモリーアクセスとキャッシュの排出を招く可能性があります。
- GPU キャッシュのミス率が高い場合、アクセスが過度に分散されていると GPU メモリー・コントローラーの効率が低下することがあります。
- 圧縮ユニットは隣接するキャッシュラインを圧縮します。高速クリアサーフェスのリード/ライトもサポートします。帯域幅を改善できますが、レイテンシーも増加します。

### GPU メモリーシステム




VE で測定された GPU メモリーアクセス:

- ファブリック帯域幅の持続はピーク時の 90% 程度。
- GPU キャッシュヒットは、最大 150 サイクル、キャッシュミスは最大 300 サイクル。TLB ミスは 50～100 サイクルを加算。
- キャッシュラインへの書き込み後に同じ GPU キャッシュラインを読み取ると、最大 30 サイクルが追加されます。

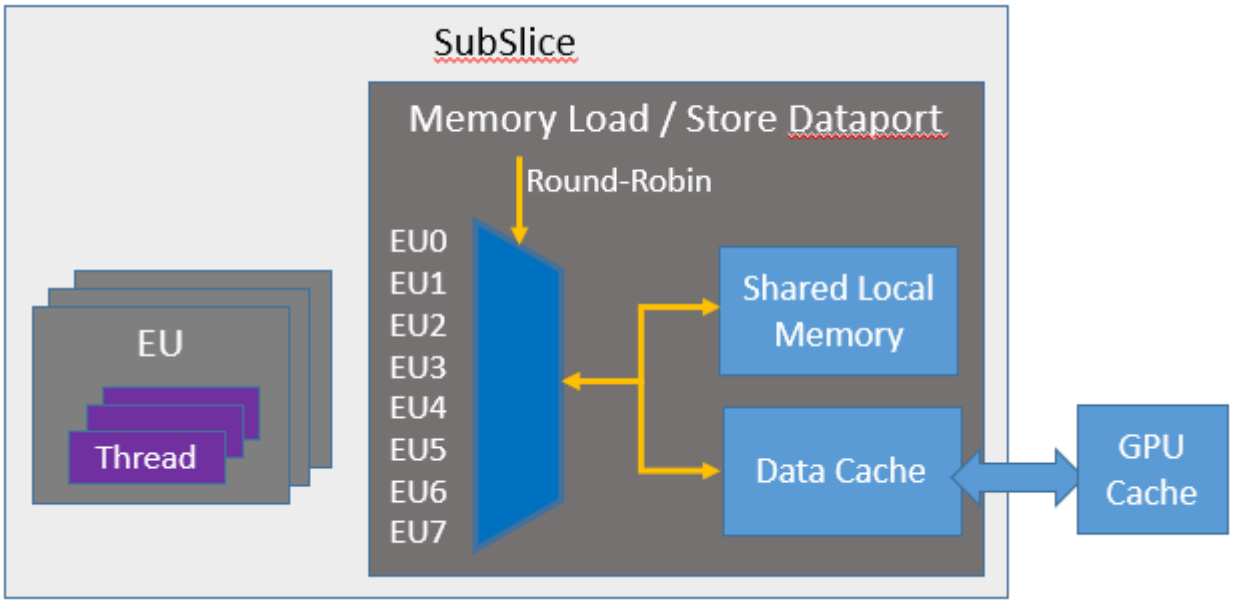
異なるスタック上のデバイスメモリーにアクセスするスタックは、スタック間の新しい GAM-to-GAM 高帯域幅インターフェイス (GT Link) を利用します。このインターフェイスの帯域幅は、任意の単一スタックのデバイス・メモリー・サブシステムで処理できるメモリー帯域幅と一致します。

## VE スレッドのロード/ストア/アトミック

VE スレッドは、メッセージをデータポートに送信してメモリーアクセスを行います。ロード命令はアドレスを送信してデータを受信し、ストア命令はアドレスとデータを送信します。Xe コアのすべての VE は、以下の図のように 1 つのメモリーロード/ストア・データポートを共有します。

- Xe コア内部: サイクルごとに ~128 から 256 バイト
- Xe コア外部: サイクルごとに ~64 バイト
- 読み取り帯域幅が、書き込み帯域幅よりも高い場合があります

### VE スレッドとメモリーアクセス



サイクルごとに新しいメモリーアクセスを開始でき、32 ビットの SIMD16 送信操作は、6 サイクルとメモリー・レイテンシー (12 サイクルとメモリー・レイテンシーで完了する 4 要素のベクトル) で完了します。メモリー帯域幅を最小化するため独立したアドレスがマージされます。メモリー・レイテンシーに注意する必要があります。

**メモリー・レイテンシーの表**

| アクセスタイプ | レイテンシー |
|---|---|
| 共有ローカルメモリー | ~30 サイクル |
| Xe コア・データ・キャッシュ・ヒット | ~50 サイクル |
| GPU キャッシュヒット | ~150 - ~200 サイクル |
| GPU キャッシュミス | ~300 - ~500 サイクル |

すべてのロードとストアは緩和された順序付け (ISO C11 メモリーモデルのリード/ライト) であり、同じスレッドからの同じアドレスに対して順序付けが行われます。同じスレッド内の異なるアドレスは順不同で完了する可能性があり、リード/ライトの順番はスレッド、VE、Xe コア間で維持されないため、開発者はアトミックやフェンス操作によって順番を保証する必要があります。

アトミック操作には、メモリー位置からの読み取りと書き込みの両方が含まれる場合があります。アトミック操作は、順序付けされていないアクセスビュー、またはスレッドグループの共有メモリーのいずれかにのみ適用されます。スレッドがメモリーアドレスに対しアトミック操作を発行すると、アトミックリードとライトの間に、外部から同じアドレスへの書き込みが発生しないことが保証されます。

異なるスレッドからの複数のアトミック操作が同じアドレスである場合、操作は未定義の順番でシリアル化されます。このシリアル化は、同じアドレスへの L2 シリアル化の規則によって発生します。アトミック操作は、メモリーフェンスまたはスレッドフェンスとは異なります。プログラマーやコンパイラーがフェンスを適切に使用しない場合、すべてのスレッドが任意のメモリー操作の結果を同時に取得することや、ほかのメモリーアドレスの更新を特定の順番で行うことは保証されません。ただし、アトミック操作は、常にグローバルレベルで記述され (共有ローカルメモリーを除く)、アトミック操作が完了すると、結果はすべてのスレッドグループで参照できます。Gen7 以降の世代ごとにアトミック操作の機能とパフォーマンスは向上しています。

次の SYCL* コードの例では、work-item ごとに 1024 の同一アドレスに対するアトミック操作が実行されます。各 work-item は異なる (一意の) アドレスを使用し、コンパイラーは VE スレッドごとに SIMD32 カーネルを生成します。これにより、2 つのキャッシュラインで 2 つの SIMD16 アトミック操作が実行され、コンパイラーはループを最大 8 回アンロールして、レジスター依存関係によるストールを減らすことができます。

```
1.  #include <CL/sycl.hpp>
2.  #include <chrono>
3.  #include <iostream>
4.  #include <string>
5.  #include <unistd.h>
6.  #include <vector>
7.
8.  #ifndef SCALE
9.  #define SCALE  1
10. #endif
11.
12. #define N         1024*SCALE
13. #define SG_SIZE   32
14.
15. // 繰り返し回数
16. constexpr int repetitions = 16;
17. constexpr int warm_up_token = -1;
18.
19. static auto exception_handler = [](sycl::exception_list eList) {
20.   for (std::exception_ptr const &e : eList) {
21.     try {
22.       std::rethrow_exception(e);
```

```
23.     } catch (std::exception const &e) {
24.       std::cout << "Failure" << std::endl;
25.       std::terminate();
26.     }
27.   }
28. };
29.
30. class Timer {
31. public:
32.   Timer() : start_(std::chrono::steady_clock::now()) {}
33.
34.   double Elapsed() {
35.     auto now = std::chrono::steady_clock::now();
36.     return std::chrono::duration_cast<Duration>(now - start_).count();
37.   }
38.
39. private:
40.   using Duration = std::chrono::duration<double>;
41.   std::chrono::steady_clock::time_point start_;
42. };
43.
44. #ifdef FLUSH_CACHE
45. void flush_cache(sycl::queue &q, sycl::buffer<int> &flush_buf) {
46.   auto flush_size = flush_buf.get_size()/sizeof(int);
47.   auto ev = q.submit([&](auto &h) {
48.     sycl::accessor flush_acc(flush_buf, h, sycl::write_only, sycl::no_init);
49.     h.parallel_for(flush_size, [=](auto index) { flush_acc[index] = 1; });
50.   });
51.   ev.wait_and_throw();
52. }
53. #endif
54.
55. void atomicLatencyTest(sycl::queue &q, sycl::buffer<int> inbuf,
56.                        sycl::buffer<int> flush_buf, int &res, int iter) {
57.   const size_t data_size = inbuf.byte_size()/sizeof(int);
58.
59.   sycl::buffer<int> sum_buf(&res, 1);
60.
61.   double elapsed = 0;
62.
63.   for (int k = warm_up_token; k < iter; k++) {
64. #ifdef FLUSH_CACHE
65.     flush_cache(q, flush_buf);
66. #endif
67.
68.     Timer timer;
69.
70.     q.submit([&](auto &h) {
71.       sycl::accessor buf_acc(inbuf, h, sycl::write_only, sycl::no_init);
72.
73.       h.parallel_for(sycl::nd_range<1>(sycl::range<>{N}, sycl::range<>{SG_SIZE}),
    [=](sycl::nd_item<1> item)
74.
    [[intel::reqd_sub_group_size(SG_SIZE)]] {
75.         int i = item.get_global_id(0);
76.         for (int ii = 0; ii < 1024; ++ii) {
77.         auto v =
78.           #ifdef ATOMIC_RELAXED
79.              sycl::atomic_ref<int, sycl::memory_order::relaxed,
80.                               sycl::memory_scope::device,
81.
    sycl::access::address_space::global_space>(buf_acc[i]);
82.           #else
83.              sycl::atomic_ref<int, sycl::memory_order::acq_rel,
84.                               sycl::memory_scope::device,
85.
    sycl::access::address_space::global_space>(buf_acc[i]);
86.           #endif
87.           v.fetch_add(1);
```

```
88.          }
89.        });
90.      });
91.      q.wait();
92.      elapsed += (iter == warm_up_token) ? 0 : timer.Elapsed();
93.    }
94.    std::cout << "SUCCESS: Time atomicLatency = " << elapsed << "s" << std::endl;
95. }
96.
97. int main(int argc, char *argv[]) {
98.    sycl::queue q{sycl::gpu_selector_v, exception_handler};
99.    std::cout << q.get_device().get_info<sycl::info::device::name>() << std::endl;
100.
101.   std::vector<int> data(N);
102.   std::vector<int> extra(N);
103.
104.   for (size_t i = 0; i < N ; ++i) {
105.     data[i]  = 1;
106.     extra[i] = 1;
107.   }
108.   int res=0;
109.
110.   const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
111.   sycl::buffer<int> buf(data.data(), data.size(), props);
112.   sycl::buffer<int> flush_buf(extra.data(), extra.size(), props);
113.   atomicLatencyTest(q, buf, flush_buf, res, 16);
114.}
```

アトミック操作を使用する実際のワークロードでは、最適な帯域幅を達成するアトミック操作を選択するには、メモリーアクセスの動作とデータセットのサイズを理解する必要があります。

## 暗黙のスケーリング

ルートデバイスは、スタックとも呼ばれる複数のサブデバイスで構成されます。これらのスタックは、明示的なスタック間の通信を必要としない、ルートデバイスをモノリシック・デバイスとして扱うことを可能にする共有メモリー空間を形成します。この節では、暗黙のスケーリングによるマルチスタック・プログラミングの原則について説明します。暗黙のスケーリングを使用する場合、ルートデバイスのドライバーは、アプリケーション・コードがカーネルを起動する際にすべてのスタックにワークを分散する役割を持ちます。

- はじめに
- ワークのスケジュールとメモリー分散
- 簡単な例
- プログラミングの原則

### はじめに

### 使用法

暗黙のスケーリングを有効にするには、環境変数 `EnableImplicitScaling=1` をエクスポートします。

この環境変数は、SYCL*/OpenMP* デバイスをルートデバイスに関連付けます。アプリケーションのコードを変更する必要はありません。SYCL*/OpenMP* デバイスに送信されたカーネルは、すべてのスタックを利用します。同様に、SYCL*/OpenMP* デバイスのメモリー割り当ては、すべてのスタックにまたがります。ドライバーの動作については、ワークのスケジュールとメモリー分散で説明します。

**注:**

- デフォルトで `EnableImplicitScaling=1` に設定されます。
- 暗黙のスケジュールは、SYCL*/OpenMP* サブデバイスのセマンティクスと組み合わせてはなりません。
- `ZE_AFFINITY_MASK` でサブデバイス構文を使用しないでください。つまり、ルートデバイス 0 (`ZE_AFFINITY_MASK=0.0`) からスタック 0 を公開する代わりに、`ZE_AFFINITY_MASK=0` を設定、または `ZE_AFFINITY_MASK` の設定を解除して、ルートデバイス全体をドライバーに公開する必要があります。
- すべての VE を使用するため、暗黙のスケジュールで利用できるのは 1 つの計算コマンド・ストリーマー (CCS) のみです。
- 暗黙のスケーリングでは、スタック 0 のコピーエンジンのみが使用されます。これは、将来のドライバーで変更される可能性があります。

## 期待されるパフォーマンス

暗黙のスケーリングでは、すべてのスタックのリソースは一度のカーネル起動で公開されます。2 つのスタックを持つルートデバイスは、カーネルは 2 倍の計算ピーク、2 倍のメモリー帯域幅、および 2 倍のメモリー容量を利用できます。理想的なケースでは、ワークロードのパフォーマンスは 2 倍向上します。ただし、キャッシュサイズとキャッシュ帯域幅も 2 倍に増加するため、キャッシュ容量にワークロードが収まる場合、線形よりも優れたスケーリングとなることがあります。

各スタックは NUMA ドメインに相当するため、最適な暗黙のスケーリング・パフォーマンスを達成するには、メモリー・アクセス・パターンとメモリー割り当てが重要になります。局所性を意識するワークロードは、クロススタック・メモリーのアクセスが自然に最小化されるため、このプログラミング・モデルで最適に機能することが期待されます。計算集約型のカーネルは NUMA ドメインの影響を受けないため、暗黙のスケーリングを使用して複数のスタックに容易にスケーリングできます。

MPI アプリケーションは、明示的なスケーリングよりも暗黙のスケーリングのほうが効率的です。これは、1 つのランクでルートデバイス全体を利用できるため、スタック間の明示的な同期と通信が不要であるためです。暗黙のスケーリングは、1 回のカーネル起動でローカル・メモリー・アクセスとスタック間のメモリーアクセスを自動的にオーバーラップします。

暗黙のスケーリングは、カーネルの実行時間のみを改善します。シリアル・ボトルネックは改善されません。大規模なシリアル・ボトルネックがある場合、暗黙のスケーリングではアプリケーションはスピードアップしません。一般的なシリアル・ボトルネックには以下があります。

- 高い CPU 使用
- カーネル起動レイテンシー
- PCIe* 転送

これらは、暗黙のスケーリングによってカーネルの実行時間が短縮されるにつれて顕著になります。スタック 0 のみがホストへ PCIe* 接続されることに注意してください。暗黙のスケーリングが有効化されているインテル® データセンター GPU マックス・シリーズでは、カーネル起動のレイテンシーが 3 マイクロ秒程度増加します。

## ワークのスケジュールとメモリー分散

ルートデバイスのドライバーは、暗黙のスケーリングが使用されると、決定論的なヒューリスティックによって work-group とメモリーページをすべてのスタックに分散します。このヒューリスティックについては、次の節で説明します。

## メモリーの区分

共有割り当てやデバイス割り当てに対応する SYCL*/OpenMP* 割り当ては、すべてのスタックで区分されます。つまり、割り当ては、スタック数のチャンクに分割され、スタック間でラウンドロビン分散されます。次のルートデバイス割り当てについて考えてみます。

OpenMP*:

```
int *a = (int*)omp_target_alloc( sizeof(int)*N, device_id );
```

SYCL*:

```
int *a = sycl::malloc_device<int>(N, q);
```

2 スタックのルートデバイスでは、前半 (`a[0]` から `a[N/2-1]` の要素) はスタック 0 に物理的に割り当てられます。残りの半分 (`a[N/2]` から `a[N-1]` の要素) は、スタック 1 に割り当てられます。将来的には、ユーザー定義のメモリー区分を可能にする API を導入する予定です。

**注:**

- 上記のメモリー区分は、ページサイズ単位で適用されます。
  - 3 ページの割り当てでは、2 ページがスタック 0 に常駐します。
- ページサイズ以下の割り当てでは、スタック 0 のみに常駐します。
- 単一割り当てのメモリープールを使用すると、メモリー区分のロジックが壊れます。オブジェクト・データがすべてのスタックに分散されるように、アプリケーションではオブジェクトごとに 1 つの割り当てを行うことを推奨します。

## 静的パーティション化

スタックへの work-group のスケジュールは決定論的であり、静的パーティション化と呼ばれます。パーティション化は単純な規則に従います。最も移動の低速な次元はスタック数のチャンクに分割され、スタック間でラウンドロビン分散されます。ルートデバイスにおける 1 次元のカーネル起動について考えてみます。

OpenMP*:

```
#pragma omp target teams distribute parallel for simd
for (int i = 0; i < N; ++i)
{
//
}
```

SYCL*:

```
q.parallel_for(N, [=](auto i) {
//
});
```

次元は 1 つしかないため、自動的に最も低速な次元になり、ドライバーによってスタック間で分割されます。2 スタックのルートデバイスでは反復 0 から N/2-1 はスタック 0 にスケジュールされます。残りの反復 N/2 から N-1 は、スタック 1 で実行されます。

OpenMP* で `collapse` 節が使用されると、最も移動の低速な次元は、最も外側のループになります。SYCL* では、最も低速な次元はグローバルレンジの最初の要素です。例えば、3D カーネルの起動について考えてみます。

OpenMP*:

```
#pragma omp target teams distribute parallel for simd collapse(3)
for (int z = 0; z < nz; ++z)
{
    for (int y = 0; y < ny; ++y)
    {
        for (int x = 0; x < nx; ++x)
        {
            //
        }
    }
}
```

SYCL*:

```
range<3> global{nz, ny, nx};
range<3> local{1, 1, 16};

cgh.parallel_for(nd_range<3>(global, local), [=](nd_item<3> item) {
    //
});
```

最も低速な次元は `z` であり、スタック間で分割されます。これは、2 スタックのルートデバイスの場合、`z=0` から `z=nz/2-1` の反復がスタック 0 で実行されます。残りの反復 `z=nz/2` から `z=nz-1` は、スタック 1 にスケジュールされます。

最も低速な次元をスタック間で均等に分割できず、5% を超えるインバランスが生じる場合、ドライバーはインバランスを解消するため、次の次元を分割します。これは、奇数次元が 19 未満のカーネルにのみ影響します。以下にさまざまなカーネル起動の例を示します (ローカルレンジ {1,1,16} を想定)。

### スタックへの work-group のパーティション化

| nz | ny | nx | パーティション |
|---|---|---|---|
| 512 | 512 | 512 | z |
| 21 | 512 | 512 | z |
| 19 | 512 | 512 | y |
| 18 | 512 | 512 | z |
| 19 | 19 | 512 | x |

SYCL* の多次元ローカルレンジでは、分割された次元が変更される可能性があります。例えば、ローカルレンジ {2,1,8} を持つグローバルレンジ {38,512,512} の場合、ドライバーは y 次元を分割し、ローカルレンジ {1,1,16} の場合、ドライバーは z 次元を分割します。OpenMP* では、最も内側のループで作成された 1 次元ローカルレンジを持つことが許されるため、静的パーティション化には影響しません。3 レベル以上の `collapse` 節を持つ OpenMP* カーネルは、入れ子になったすべての `for` ループが線形化された 1 次元のカーネルに相当します。この線形化されたループは、1D カーネルの起動ヒューリスティックに従って分割されます。

**注:**

- 静的パーティション化は work-group 単位で行われます。
  - つまり、work-group 内のすべての work-item が、同じスタックにスケジュールされるということです。
- 単一 work-group を持つカーネルはスタック 0 にのみ常駐します。

## 簡単な例

カーネルの例:

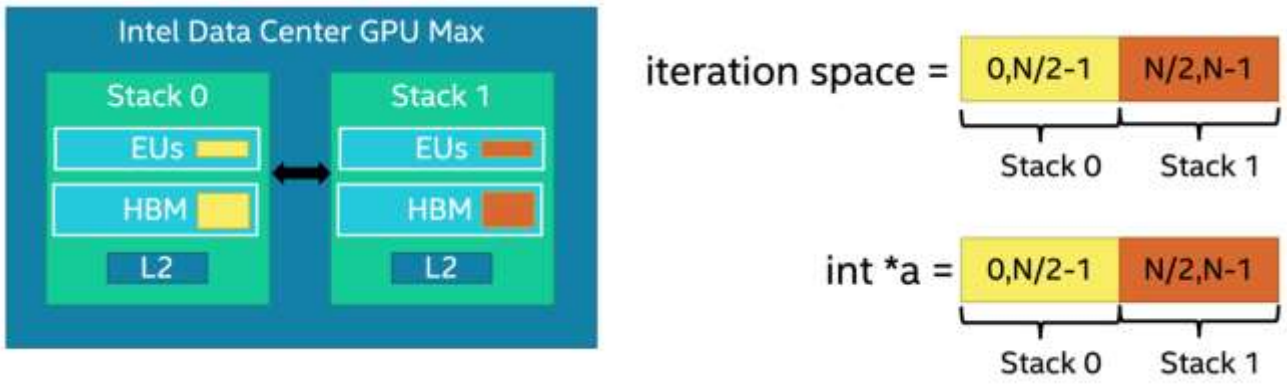
OpenMP*:

```
int *a = (int *)omp_target_alloc(sizeof(int) * N, device_id);

#pragma omp target teams distribute parallel for simd
for (int i = 0; i < N; ++i)
{
    a[i] = i;
}
```

SYCL*:

```
int *a = sycl::malloc_device<int>(N, q);

q.parallel_for(N, [=](auto i) {
    a[i] = i;
});
```

暗黙のスケーリングによって、ローカル・メモリー・アクセスは 100% 保証されます。静的パーティション化とメモリー区分の動作を以下に示します。




次の節では、インテル® データセンター GPU マックス・シリーズで 1D および 3D カーネルを起動して、STREAM ベンチマークの暗黙のスケーリングのパフォーマンスを示しています。

## STREAM

OpenMP* で記述された STREAM ベンチマークについて考えてみます。メインのカーネルは 38-41 行目にあります。

```
1.  // STREAM のコード:
2.  #include <iostream>
3.  #include <omp.h>
4.  #include <cstdint>
5.
6.  // 以下でコンパイル:
7.  // icpx -O2 -fiopenmp -fopenmp-targets=spir64 ./stream.cpp
8.
9.  int main()
10. {
11.   constexpr int64_t N = 256 * 1e6;
12.   constexpr int64_t bytes = N * sizeof(int64_t);
```

```
13.
14.   int64_t *a = static_cast<int64_t *>(malloc(bytes));
15.   int64_t *b = static_cast<int64_t *>(malloc(bytes));
16.   int64_t *c = static_cast<int64_t *>(malloc(bytes));
17.
18.   #pragma omp target enter data map(alloc:a[0:N])
19.   #pragma omp target enter data map(alloc:b[0:N])
20.   #pragma omp target enter data map(alloc:c[0:N])
21.
22.   for (int i = 0; i < N; ++i)
23.   {
24.     a[i] = i + 1;
25.     b[i] = i - 1;
26.   }
27.
28.   #pragma omp target update to(a[0:N])
29.   #pragma omp target update to(b[0:N])
30.
31.   const int no_max_rep = 100;
32.   double time;
33.   for (int irep = 0; irep < no_max_rep + 10; ++irep)
34.   {
35.     if (irep == 10)
36.       time = omp_get_wtime();
37.
38.     #pragma omp target teams distribute parallel for simd
39.     for (int i = 0; i < N; ++i)
40.     {
41.       c[i] = a[i] + b[i];
42.     }
43.   }
44.   time = omp_get_wtime() - time;
45.   time = time / no_max_rep;
46.
47.   #pragma omp target update from(c[0:N])
48.
49.   for (int i = 0; i < N; ++i)
50.   {
51.     if (c[i] != 2 * i)
52.     {
53.       std::cout << "wrong results!" << std::endl;
54.       exit(1);
55.     }
56.   }
57.
58.   const int64_t streamed_bytes = 3 * N * sizeof(int64_t);
59.
60.   std::cout << "bandwidth = " << (streamed_bytes / time) * 1E-9
61.             << " GB/s" << std::endl;
62. }
```

ベンチマークは、環境変数 `EnableImplicitScaling=1` を設定して、暗黙のスケーリングを有効にすることで、ルートデバイス全体で実行されます。コードを変更する必要はありません。静的パーティション化とメモリー区分のヒューリスティックにより、各スタックがローカルメモリーにのみアクセスすることが保証されます。2 スタックのインテル® データセンター GPU マックス・シリーズのシステムでは、単一スタックと比較して STEAM は 2 倍のスピードアップを示しました。測定された帯域幅を以下に示します。

**1D カーネル起動の帯域幅を測定**

| 配列サイズ [MB] | 1 スタック帯域幅 [GB/秒] | 暗黙のスケーリング帯域幅 [GB/秒)] | 暗黙のスケーリング 1 スタック以上のスピードアップ |
|---|---|---|---|
| 512 | 1056 | 2074 | 1.96x |
| 1024 | 1059 | 2127 | 2x |
| 2048 | 1063 | 2113 | 1.99x |

## 3D STREAM

STREAM ベンチマークは、OpenMP* の `collapse` 節を使用して 3D カーネル起動を使用するように変更できます。ここでは、ドライバーのヒューリスティックにより、3D カーネル起動をスタック間で分割するパフォーマンスを理解することです。カーネルは 52-62 行目にあります。

```
1.  // 3D STREAM のコード
2.  #include <iostream>
3.  #include <omp.h>
4.  #include <cassert>
5.
6.  // 以下でコンパイル:
7.  // icpx -O2 -fiopenmp -fopenmp-targets=spir64 ./stream_3D.cpp
8.
9.  int main(int argc, char **argv)
10. {
11.   const int device_id = omp_get_default_device();
12.   const int desired_total_size = 32 * 512 * 16384;
13.   const std::size_t bytes = desired_total_size * sizeof(int64_t);
14.
15.   std::cout << "memory footprint = " << 3 * bytes * 1E-9 << " GB" << std::endl;
16.
17.   int64_t *a = static_cast<int64_t*>(omp_target_alloc_device(bytes, device_id));
18.   int64_t *b = static_cast<int64_t*>(omp_target_alloc_device(bytes, device_id));
19.   int64_t *c = static_cast<int64_t*>(omp_target_alloc_device(bytes, device_id));
20.
21.   const int min = 64;
22.   const int max = 32768;
23.
24.   for (int lx = min; lx < max; lx *= 2)
25.   {
26.     for (int ly = min; ly < max; ly *= 2)
27.     {
28.       for (int lz = min; lz < max; lz *= 2)
29.       {
30.         const int total_size = lx * ly * lz;
31.         if (total_size != desired_total_size)
32.           continue;
33.
34.         std::cout << "lx=" << lx << " ly=" << ly << " lz=" << lz << ", ";
35.
36.         #pragma omp target teams distribute parallel for simd
37.         for (int i = 0; i < total_size; ++i)
38.         {
39.           a[i] = i + 1;
40.           b[i] = i - 1;
41.           c[i] = 0;
42.         }
43.
44.         const int no_max_rep = 40;
45.         const int warmup = 10;
46.         double time;
47.         for (int irep = 0; irep < no_max_rep + warmup; ++irep)
48.         {
49.           if (irep == warmup) time = omp_get_wtime();
50.
51.           #pragma omp target teams distribute parallel for simd collapse(3)
52.           for (int iz = 0; iz < lz; ++iz)
53.           {
54.             for (int iy = 0; iy < ly; ++iy)
55.             {
56.               for (int ix = 0; ix < lx; ++ix)
57.               {
58.                 const int index = ix + iy * lx + iz * lx * ly;
59.                 c[index] = a[index] + b[index];
60.               }
```

```
61.              }
62.            }
63.          }
64.          time = omp_get_wtime() - time;
65.          time = time / no_max_rep;
66.
67.          const int64_t streamed_bytes = 3 * total_size * sizeof(int64_t);
68.
69.          std::cout << "bandwidth = " << (streamed_bytes / time) * 1E-9
70.                    << " GB/s" << std::endl;
71.
72.          #pragma omp target teams distribute parallel for simd
73.          for (int i = 0; i < total_size; ++i)
74.          {
75.            assert(c[i] == 2 * i);
76.          }
77.        }
78.      }
79.    }
80.
81.    omp_target_free(a, device_id);
82.    omp_target_free(b, device_id);
83.    omp_target_free(c, device_id);
84. }
```

最も内側のループのメモリー・アクセス・パターンは、ストライド 1 であることに注目してください。z ループまたは y ループが最も内側のループである場合、キャッシュラインの利用率が低下する分散ロードとストアが行われるため、パフォーマンスが低下します。2GB アレイサイズの 2 スタックのインテル® データセンター GPU マックス・シリーズのシステムでは、次のパフォーマンスが測定されました。

**3D カーネル起動の帯域幅を測定:**

| nx | ny | nz | 1 スタック帯域幅 [GB/秒] | 暗黙の大規模帯域幅 [GB/秒)] | 1 スタック以上の暗黙のスケーリングのスピードアップ |
|---|---|---|---|---|---|
| 64 | 256 | 16834 | 1040 | 2100 | 2.01x |
| 16834 | 64 | 256 | 1040 | 2077 | 1.99x |
| 256 | 16834 | 64 | 1037 | 2079 | 2x |

「静的パーティション化」の項で説明されているように、これらのループ境界では、ドライバーは両方のスタック間で最も低速な次元である z を分割します。これにより、各スタックがローカルメモリーのみにアクセスすることが保証され、単一スタックと比較して 2 倍近い暗黙のスケーリングが可能になります。

## プログラミングの原則

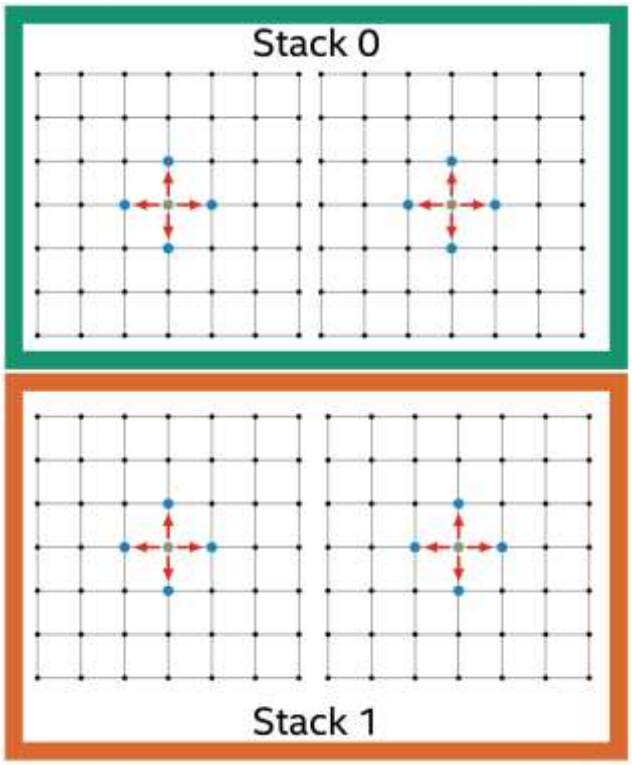
暗黙的なスケーリングで優れたパフォーマンスを達成するには、スタック間のメモリーアクセスを最小限にする必要がありますが、そのすべてを排除する必要はありません。ローカルメモリーと同時にスタック間のメモリーがアクセスされる場合、一定のスタック間のトラフィックは相互接続で処理できます。メモリー帯域幅が制限されるワークロードでは、許容されるスタック間のアクセス量は、ローカルメモリー帯域幅とスタック間の帯域幅の比率によって決定されます (「スタック間のトラフィック」を参照)。

暗黙のスケーリングを使用するワークロードは、以下の原則に従う必要があります。

- カーネルには両方のスタックを利用するのに十分な work-item が必要です。
  - 必要となる work-item の最小数は、<VE 数> * <VE ごとのハードウェア・スレッド数> * <SIMD 幅> になります。
  - 1024 の VE と SIMD32 の 2 スタックのインテル® データセンター GPU マックス・シリーズでは、最低 262,144 の work-item です。

- アプリケーション全体のスケーリングを観察するには、デバイス時間が実行時間の大部分でなければなりません。
- アルゴリズムの局所性により、スタック間のアクセスを最小化します。
- スタック間の負荷インバランスを避けるため、最も低速な次元を大きくする必要があります。
- スタック間のメモリーアクセスとローカル・メモリー・アクセスは、インターリーブする必要があります。
- 2D および 3D カーネルの起動でも、最も低速な次元でストライド 1 のメモリーアクセスを回避します。
- メモリー・アクセス・パターンが時間経過とともに動的に変化する場合、スタック間のメモリーアクセスを最小化するため、n 回の反復ごとにソートステップを実行する必要があります。
- 単一割り当てのメモリープールを使用してはなりません (メモリーの区分を参照)。

ほとんどのアプリケーションには、自然に局所性の概念が備わっています。それらのアプリケーションは、スタック間のトラフィックが少ないため、暗黙のスケーリングが適していると考えられます。これを説明するため、ステンシルカーネルについて考えてみます。ステンシルはブロックに分割可能なグリッド上で動作し、ブロック内のステンシル計算のほとんどはスタックのローカルデータを使用します。ブロック境界のステンシル操作のみが、別のブロック (別のスタック) のデータを必要とします。スタック間/境界間のアクセス量は、ハローとローカルの体積比によって制限されます。この概念を以下に示します。




## スタック間のトラフィック

前述したように、スタック間のトラフィックは最小限にする必要があります。アプリケーションのパフォーマンスに大きく影響しないトラフィック量を求めるには、さまざまな量のスタック間トラフィックで STREAM カーネルを実行し、スタックローカルの STREAM のパフォーマンスと比較します。最悪のケースは 100% のスタック間トラフィックです。これは、STREAM カーネルのループの順番を反転することで検証できます (「STREAM」を参照)。

```
#pragma omp target teams distribute parallel for simd
for (int i = N - 1; i <= 0; --i)
{
    c[i] = a[i] + b[i];
}
```

スタック 0 の work-group がスタック 1 のメモリーに配置されている配列要素 `N-1` から `N/2` にアクセスするため、スタック間のメモリー・トラフィックは 100% になります。このカーネルは、スタック間の双方向帯域幅をベンチマークします。このアプローチは、以下に示す STREAM によって 0% のスタック間アクセスと 100% のスタック間アクセスを補完するように一般化できます。

```
1.  // スタック間 stream のコード
2.  #include <iostream>
3.  #include <omp.h>
4.
5.  // 以下でコンパイル:
6.  // icpx -O2 -fiopenmp -fopenmp-targets=spir64 ./stream_cross_stack.cpp
7.  // 以下で実行:
8.  // EnableWalkerPartition=1 ZE_AFFINITY_MASK=0 ./a.out
9.
10. template <int cross_stack_fraction>
11. void cross_stack_stream() {
12.
13.   constexpr int64_t size = 256*1e6;
14.   constexpr int64_t bytes = size * sizeof(int64_t);
15.
16.   int64_t *a = static_cast<int64_t*>(malloc( bytes ));
17.   int64_t *b = static_cast<int64_t*>(malloc( bytes ));
18.   int64_t *c = static_cast<int64_t*>(malloc( bytes ));
19.
20. #pragma omp target enter data map( alloc:a[0:size] )
21. #pragma omp target enter data map( alloc:b[0:size] )
22. #pragma omp target enter data map( alloc:c[0:size] )
23.
24.   for ( int i = 0; i < size; ++i ) {
25.     a[i] = i + 1;
26.     b[i] = i - 1;
27.     c[i] = 0;
28.   }
29.
30. #pragma omp target update to( a[0:size] )
31. #pragma omp target update to( b[0:size] )
32. #pragma omp target update to( c[0:size] )
33.
34.   const int num_max_rep = 100;
35.   double time;
36.
37.   for ( int irep = 0; irep < num_max_rep+10; ++irep ) {
38.     if ( irep == 10 ) time = omp_get_wtime();
39. #pragma omp target teams distribute parallel for simd
40.     for ( int j = 0; j < size; ++j ) {
41.       const int cache_line_id = j / 16;
42.       int i;
43.
44.       if ( (cache_line_id%cross_stack_fraction) == 0 ) {
45.         i = (j+size/2)%size;
46.       }
47.       else {
48.         i = j;
49.       }
50.       c[i] = a[i] + b[i];
51.     }
52.   }
53.   time = omp_get_wtime() - time;
54.   time = time/num_max_rep;
```

```
55.
56. #pragma omp target update from( c[0:size] )
57.   for ( int i = 0; i < size; ++i ) {
58.     if ( c[i] != 2*i ) {
59.       std::cout << "wrong results!"<< std::endl;
60.       exit(1);
61.     }
62.   }
63.
64.   const int64_t streamed_bytes = 3 * size * sizeof(int64_t);
65.   std::cout << "cross_stack_percent = " << (1/(double)cross_stack_fraction)*100
66.       << "%, bandwidth = " << (streamed_bytes/time) * 1E-9 << " GB/s" << std::endl;
67. }
68.
69. int main() {
70.   cross_stack_stream< 1>();
71.   cross_stack_stream< 2>();
72.   cross_stack_stream< 4>();
73.   cross_stack_stream< 8>();
74.   cross_stack_stream<16>();
75.   cross_stack_stream<32>();
76. }
```

40–51 行目のカーネルは、配列アクセスを `(j+N/2)%N` でオフセットすることで、`cross_stack_fraction` 番目のキャッシュラインごとにスタック間にアクセスします。`cross_stack_fraction==1` の場合、100% のスタック間メモリーアクセスを行います。`cross_stack_fraction` を 2 倍にすることで、スタック間のトラフィックを半分にできます。このカーネルは、スタック間とローカルメモリーのアクセスが work-group 内でインターリーブされ、ハードウェアの利用率を最大化するように設計されています。2GB アレイサイズの 2 スタックのインテル® データセンター GPU マックス・シリーズで測定したパフォーマンスを以下に示します。

**スタック間アクセスの帯域幅を測定**

| 部分的スタック間の STREAM 帯域幅 [GB/秒] | スタック間比率 | スタック間アクセスの % | 最大ローカル 2 スタック STREAM 帯域幅の % |
|---|---|---|---|
| 355 | 1 | 100% | 17% |
| 696 | 2 | 50% | 33% |
| 1223 | 4 | 25% | 58% |
| 1450 | 8 | 12.5% | 69% |
| 1848 | 16 | 6.25% | 87% |
| 2108 | 32 | 3.125% | 99% |

上の表から分かるように、アプリケーションはスタック間のトラフィックを全メモリー・トラフィックの 10% 未満に制限して、持続的なメモリー帯域幅の低下を回避する必要があります。スタック間のアクセスが 12.5% の STREAM では、ローカル STREAM ベンチマークの帯域幅の 69% でした。これらの数値から、スタック間のメモリーアクセスがアプリケーションのカーネル実行に与える影響を算出できます。

## 明示的なスケーリング

明示的なスケーリングでは、プログラマーが work-group の分散とメモリー配置を直接制御します。この章では、以下について説明します。

- マルチ GPU、マルチスタック、およびマルチ C スライスでの SYCL* による明示的なスケーリング
- SYCL* でインテル® oneMKL マス・カーネル・ライブラリー (oneMKL) を使用した明示的なスケーリング
- マルチ GPU、マルチスタック、およびマルチ C スライスでの OpenMP* による明示的なスケーリング
- OpenMP* でインテル® oneMKL マス・カーネル・ライブラリー (oneMKL) を使用した明示的なスケーリング
- 明示的なスケーリングのまとめ

## マルチ GPU、マルチスタック、およびマルチ C スライスでの SYCL* による明示的なスケーリング

ここでは、SYCL* の明示的なスケーリングの言語 API について説明し、インテル® データセンター GPU マックス・シリーズ・プラットフォームでのマルチ GPU とマルチスタックの使い方を説明します。

### デバイスの検出

アプリケーションを実行する前に、`sycl-ls` コマンドを実行して、プラットフォームで利用可能なデバイスを確認してください。これは、パフォーマンスを測定する場合、特に重要です。これにより、実行のフォールバック・パスを確認できます。

### ルートデバイス

インテル® GPU は、SYCL* GPU デバイスおよびルートデバイスとして示されます。ルートデバイスを検出するには、`sycl-ls` コマンドを使用するのが最も容易です。次に例を示します。

```
$ sycl-ls

[opencl:0] GPU : Intel(R) OpenCL HD Graphics 3.0 [21.35.020776] [level_zero:0] GPU :
Intel(R) Level-Zero 1.1 [1.1.20776] [host:0] HOST: SYCL host platform 1.2 [1.2]
```

`sycl-ls` は、SYCL* ランタイムで認識されるすべての SYCL* バックエンドのすべてのプラットフォーム・デバイスを表示します。上記の例では、単一の物理 GPU (OpenCL* またはレベルゼロ・バックエンドで管理される) ルートデバイスに対応する GPU があります。

「SYCL* ランタイム環境変数」で説明される環境変数 `ONEAPI_DEVICE_SELECTOR` を使用できます。

```
$ export ONEAPI_DEVICE_SELECTOR=level_zero:*
$ sycl-ls

[level_zero:0] GPU : Intel(R) Level-Zero 1.1 [1.1.20776]
```

システムに複数の GPU が搭載されている場合、それらは異なる複数のルートデバイスとして認識されます。Linux* では、これらは同一 SYCL* プラットフォーム (レベルゼロ・ドライバーを示す) の複数の SYCL* ルートデバイスになります。Windows* では、これらは異なる複数の SYCL* プラットフォーム (レベルゼロ・ドライバーを示す) のルートデバイスとして表示されます。

`CreateMultipleRootDevices=N` と `NEOReadDebugKeys=1` 環境変数を設定して、複数の GPU カードをエミュレートできます。次に例を示します。

```
$ export CreateMultipleRootDevices=2
$ export NEOReadDebugKeys=1
$ export SYCL_DEVICE_FILTER=level_zero
$ sycl-ls

[level_zero:0] GPU : Intel(R) Level-Zero 1.1 [1.1.20776]
[level_zero:1] GPU : Intel(R) Level-Zero 1.1 [1.1.20776]
```

### サブデバイス

インテル® データセンター GPU マックス 1350 または 1550 には、2 つのスタックがあります。GPU 全体を示すルートデバイスは、2 つのサブデバイスに分割でき、各サブデバイスは物理スタックに対応します。

```
try {
```

```
vector<device> SubDevices = RootDevice.create_sub_devices<
cl::sycl::info::partition_property::partition_by_affinity_domain>(
cl::sycl::info::partition_affinity_domain::numa);
}
```

`create_sub_devices` を呼び出すたびに、同じサブデバイスが永続的な順番で返されます。レベルゼロ UMD によって公開されるサブデバイスを制御するには、`ZE_AFFINITY_MASK` 環境変数を使用します。インテル® GPU でサポートされるパーティション化は、`partition_by_affinity_domain` のみであることを忘れないでください。

同様に、サポートされるパーティション化のプロパティーは、`next_partitionable` と `numa` のみです (これらは同じ動作となります)。`CreateMultipleRootDevices=N` と `NEOReadDebugKeys=1` 環境変数を設定して、マルチスタックの GPU をエミュレートできます。

## サブのサブデバイス

サブデバイス (スタック) は、さらにサブのサブデバイス (計算スライス) に分解できます。サブのサブデバイスに関連付けるコンテキスト作成できます。このスキームでは、実行リソースはサブのサブデバイスに制限され、プログラムは計算スライスレベルで制御できます。次のコードは、デバイスのすべてのサブデバイスとサブのサブデバイスを検索します。

```
1.  #include <CL/sycl.hpp>
2.  #include <iostream>
3.  using namespace sycl;
4.  int main() {
5.    sycl::device d(sycl::gpu_selector_v);
6.    std::vector<sycl::device> *subdevices = new std::vector<sycl::device>();
7.    std::vector<sycl::device> *CCS = new std::vector<sycl::device>();
8.    auto part_prop = d.get_info<sycl::info::device::partition_properties>();
9.    size_t num_of_tiles;
10.   size_t num_of_ccs;
11.   if (part_prop.empty()) {
12.     num_of_tiles = 1;
13.   } else {
14.     for (int i = 0; i < part_prop.size(); i++) {
15.       if (part_prop[i] ==
16.           sycl::info::partition_property::partition_by_affinity_domain) {
17.         auto sub_devices = d.create_sub_devices<
18.             sycl::info::partition_property::partition_by_affinity_domain>(
19.             sycl::info::partition_affinity_domain::numa);
20.         num_of_tiles = sub_devices.size();
21.         for (int j = 0; j < num_of_tiles; j++)
22.           subdevices->push_back(sub_devices[j]);
23.         break;
24.       } else {
25.         num_of_tiles = 1;
26.       }
27.     }
28.   }
29.   std::cout << "List of Tiles:\n";
30.   for (int i = 0; i < num_of_tiles; i++) {
31.     std::cout << i << ") Device name: "
32.               << (*subdevices)[i].get_info<sycl::info::device::name>() << "\n";
33.     std::cout
34.         << "  Max Compute Units: "
35.         << (*subdevices)[i].get_info<sycl::info::device::max_compute_units>()
36.         << "\n";
37.   }
38.   for (int j = 0; j < num_of_tiles; j++) {
39.     auto part_prop1 =
40.         (*subdevices)[j].get_info<sycl::info::device::partition_properties>();
41.     if (part_prop1.empty()) {
42.       std::cout << "No partition support\n";
43.     } else {
```

```
44.       for (int i = 0; i < part_prop1.size(); i++) {
45.         if (part_prop1[i] ==
46.             sycl::info::partition_property::partition_by_affinity_domain) {
47.           auto ccses =
48.               (*subdevices)[j]
49.                   .create_sub_devices<sycl::info::partition_property::
50.                                       partition_by_affinity_domain>(
51.                       sycl::info::partition_affinity_domain::numa);
52.           num_of_ccs = ccses.size();
53.           for (int k = 0; k < num_of_ccs; k++)
54.             CCS->push_back(ccses[k]);
55.           break;
56.         } else {
57.           num_of_ccs = 1;
58.         }
59.       }
60.     }
61.   }
62.   std::cout << "List of Compute Command Streamers:\n";
63.   for (int i = 0; i < CCS->size(); i++) {
64.     std::cout << i << ") Device name: "
65.               << (*CCS)[i].get_info<sycl::info::device::name>() << "\n";
66.     std::cout << "  Max Compute Units: "
67.               << (*CCS)[i].get_info<sycl::info::device::max_compute_units>()
68.               << "\n";
69.   }
70.   return 0;
71. }
```

## コンテキスト

コンテキストは、リソースの分離と共有に使用されます。SYCL* コンテキストは、1 つ以上のデバイスで構成されます。ルートデバイスとサブデバイスは単一コンテキスト内に配置できますが、それらは同じ SYCL* プラットフォームでなければなりません。複数のデバイスを持つコンテキストに対して作成された SYCL* プログラム (`kernel_bundle`) は、コンテキスト内のそれぞれのルートデバイス向けにビルドされます。同じルートデバイスにある複数のサブデバイスで構成されるコンテキストの場合、ルートデバイスに対する単一のビルドのみが必要です。

## 統合共有メモリー

ルートデバイスに割り当てられたメモリーは、そのすべてのサブデバイス (スタック) からアクセスできます。つまり、同じルートデバイスにある複数のサブデバイスを持つコンテキストで操作する場合、低速の `malloc_host` を使用する代わりに、ルートデバイスで `malloc_device` を使用できます。`malloc_device` を使用してホストでデータを参照する場合、ホストへの明示的なコピーが必要です。3 種類の USM 割り当ての詳細については、「統合共有メモリーの割り当て」を参照してください。

## バッファー

SYCL* バッファーもまたコンテキストに対して作成され、レベルゼロ USM 割り当てにマップされます。現在のマッピングを以下に示します。

- 統合デバイスでは、割り当てはホスト上で行われ、ホストとデバイスはコピーなしでアクセスできます。
- 同じルートデバイスのサブデバイス (ルートデバイス自身も含む) のコンテキスト向けのメモリーバッファーは、ルートデバイスに割り当てられます。したがって、それらはすべてのデバイスから容易にアクセスできます。ホストとの同期は SYCL* RT によって行われ、必要に応じてマップ/アンマップが暗黙のコピーを実行します。
- 異なるルートデバイスのデバイスのコンテキストのメモリーバッファーは、ホストに割り当てられるため、すべてのデバイスからアクセスできます。

## キュー

SYCL* キューは、マルチデバイス・コンテキストで常に 1 つのデバイスに接続されることがあります。以下に、パフォーマンスの高いものから低いものへ、典型的なシナリオを示します。

### 単一サブデバイスに関連付けられたコンテキスト

単一のサブデバイスを含むコンテキストを作成し、キューをそのサブデバイス (スタック) に接続します。このスキームでは、実行と可視性は単一のサブデバイスに限定され、スタックごとに最高のパフォーマンスが期待できます。コード例を示します。

```
try {
vector<device> SubDevices = ...;
for (auto &D : SubDevices) {
// 各キューは個別のコンテキスト内にあり、それらの間でデータが共有されることはありません
auto Q = queue(D);
Q.submit([&](handler &cgh) { ... });
}
}
```

### 複数のサブデバイスに関連付けられたコンテキスト

同じルートデバイスに複数のサブデバイス (複数スタック) を持つコンテキストを作成します。明示的なスケーリングを効率良く実装するサブデバイスにキューが接続されます。このスキームでは、パフォーマンスを向上するため、ルートデバイスをこのコンテキストに渡すべきではありません。コード例を示します。

```
try {
vector<device> SubDevices = ...;
auto C = context(SubDevices);
for (auto &D : SubDevices) {
// すべてのキューは同じコンテキストを共有し、データはキュー間で共有できます
//
auto Q = queue(C, D);
Q.submit([&](handler &cgh) { ...});
}
}
```

### ルートデバイスに関連するコンテキスト

単一のルートデバイスを持つコンテキストを作成し、キューをそのルートデバイスに接続します。このスキームでは、ワークは GPU ドライバーによる暗黙のスケーリングによって、すべてのサブデバイス (スタック) に自動的に分散されます。これは、マルチスタック・ハードウェアを有効にする最も容易な方法ですが、特定のスタックをターゲットにすることはありません。コード例を示します。

```
try {
// キューはルートデバイスに接続され、ドライバーはサブデバイスに分散します
// (存在する場合)
auto D = device(gpu_selector{});
auto Q = queue(D);
Q.submit([&](handler &cgh) { ...});
}
```

## 複数のルートデバイスに関連付けられたコンテキスト

複数のルートデバイス (複数のカード) でコンテキストを作成します。このスキームでは、キューが異なるルートデバイスに接続される最も自由度の高いコンテキストです。ホストメモリーや明示的なコピーによる低速アクセスと引き換えに共有の可能性を高めます。コード例を示します。

```
try {
  auto P = platform(gpu_selector{});
  auto RootDevices = P.get_devices();
  auto C = context(RootDevices);
  for (auto &D : RootDevices) {
    // コンテキストには複数のルートデバイスがあり、データは複数のカード間で共有できます
    // (明示的なコピーが必要です)
    auto Q = queue(C, D);
    Q.submit([&](handler &cgh) { ...});
  }
}
```

明示的に指定するサブデバイスとアルゴリズムに応じて、適切なメモリー割り当て/同期を行います。次のプログラムは、明示的にサブデバイスを指定する例です (スタックごとに 1 つのキュー)。

```
1.  #include <CL/sycl.hpp>
2.  #include <algorithm>
3.  #include <cassert>
4.  #include <cfloat>
5.  #include <iostream>
6.  #include <string>
7.  using namespace sycl;
8.
9.  constexpr int num_runs = 10;
10. constexpr size_t scalar = 3;
11.
12. cl_ulong triad(size_t array_size) {
13.
14.   cl_ulong min_time_ns0 = DBL_MAX;
15.   cl_ulong min_time_ns1 = DBL_MAX;
16.
17.   device dev = device(gpu_selector_v);
18.
19.   std::vector<device> subdev = {};
20.   subdev = dev.create_sub_devices<sycl::info::partition_property::
21.     partition_by_affinity_domain>(sycl::info::partition_affinity_domain::numa);
22.
23.   queue q[2] = {queue(subdev[0], property::queue::enable_profiling{}),
24.     queue(subdev[1], property::queue::enable_profiling{})};
25.
26.   std::cout << "Running on device: " <<
27.     q[0].get_device().get_info<info::device::name>() << "\n";
28.   std::cout << "Running on device: " <<
29.     q[1].get_device().get_info<info::device::name>() << "\n";
30.
31.   double *A0 = malloc_shared<double>(array_size/2 * sizeof(double), q[0]);
32.   double *B0 = malloc_shared<double>(array_size/2 * sizeof(double), q[0]);
33.   double *C0 = malloc_shared<double>(array_size/2 * sizeof(double), q[0]);
34.
35.   double *A1 = malloc_shared<double>(array_size/2 * sizeof(double), q[1]);
36.   double *B1 = malloc_shared<double>(array_size/2 * sizeof(double), q[1]);
37.   double *C1 = malloc_shared<double>(array_size/2 * sizeof(double), q[1]);
38.
39.   for ( int i = 0; i < array_size/2; i++) {
40.      A0[i]= 1.0; B0[i]= 2.0; C0[i]= 0.0;
41.      A1[i]= 1.0; B1[i]= 2.0; C1[i]= 0.0;
42.   }
43.
```

```
  for (int i = 0; i< num_runs; i++) {
    auto q0_event = q[0].submit([&](handler& h) {
        h.parallel_for(array_size/2, [=](id<1> idx) {
            C0[idx] = A0[idx] + B0[idx] * scalar;
            });
        });

    auto q1_event = q[1].submit([&](handler& h) {
        h.parallel_for(array_size/2, [=](id<1> idx) {
            C1[idx] = A1[idx] + B1[idx] * scalar;
            });
        });

    q[0].wait();
    q[1].wait();

    cl_ulong exec_time_ns0 =
      q0_event.get_profiling_info<info::event_profiling::command_end>() -
      q0_event.get_profiling_info<info::event_profiling::command_start>();

    std::cout << "Tile-0 Execution time (iteration " << i << ") [sec]: "
      << (double)exec_time_ns0 * 1.0E-9 << "\n";
    min_time_ns0 = std::min(min_time_ns0, exec_time_ns0);

    cl_ulong exec_time_ns1 =
      q1_event.get_profiling_info<info::event_profiling::command_end>() -
      q1_event.get_profiling_info<info::event_profiling::command_start>();

    std::cout << "Tile-1 Execution time (iteration " << i << ") [sec]: "
      << (double)exec_time_ns1 * 1.0E-9 << "\n";
    min_time_ns1 = std::min(min_time_ns1, exec_time_ns1);
  }

  // 正当性のチェック
  bool error = false;
  for ( int i = 0; i < array_size/2; i++) {
    if ((C0[i] != A0[i] + scalar * B0[i]) || (C1[i] != A1[i] + scalar * B1[i])) {
      std::cout << "\nResult incorrect (element " << i << " is " << C0[i] << ")!\n";
      error = true;
    }
  }

  sycl::free(A0, q[0]);
  sycl::free(B0, q[0]);
  sycl::free(C0, q[0]);

  sycl::free(A1, q[1]);
  sycl::free(B1, q[1]);
  sycl::free(C1, q[1]);

  if (error) return -1;

  std::cout << "Results are correct!\n\n";
  return std::max(min_time_ns0, min_time_ns1);
}

int main(int argc, char *argv[]) {

  size_t array_size;
  if (argc > 1 ) {
    array_size =  std::stoi(argv[1]);
  }
  else {
    std::cout << "Run as ./<progname> <arraysize in elements>\n";
    return 1;
  }
  std::cout << "Running with stream size of " << array_size
    << " elements (" << (array_size * sizeof(double))/(double)1024/1024 << "MB)\n";

```

```
113.  cl_ulong min_time = triad(array_size);
114.
115.  if (min_time == -1) return 1;
116.  size_t triad_bytes = 3 * sizeof(double) * array_size;
117.  std::cout << "Triad Bytes: " << triad_bytes << "\n";
118.  std::cout << "Time in sec (fastest run): " << min_time * 1.0E-9 << "\n";
119.  double triad_bandwidth = 1.0E-09 * triad_bytes/(min_time*1.0E-9);
120.  std::cout << "Bandwidth of fastest run in GB/s: " << triad_bandwidth << "\n";
121.  return 0;
122.}
```

事前 (AOT) コンパイルを行うコマンドを以下に示します。

```
$ icpx -fsycl -fsycl-targets=spir64_gen -O2 -ffast-math -Xs "-device pvc" explicit-
subdevice.cpp -o run.exe
```

## SYCL* でインテル® oneAPI マス・カーネル・ライブラリーを使用した明示的なスケーリング

このセクションでは、マルチ・スタック・プラットフォーム上でインテル® oneAPI マス・カーネル・ライブラリー (oneMKL) で明示的なスケーリングを使用する方法を説明します。

次のコードは、GEMM 計算を 2 つのスタックに分割する完全な例を示します。この例では、{S,D,C,Z}GEMM API の明示的なスケーリングを示します。

```
#include "mkl.h"
#include "oneapi/mkl/blas.hpp"
#include <CL/sycl.hpp>

// Helper functions
template <typename fp> fp set_fp_value(fp arg1, fp arg2 = 0.0) { return arg1; }

template <typename fp>
std::complex<fp> set_fp_value(std::complex<fp> arg1,
                              std::complex<fp> arg2 = 0.0) {
  return std::complex<fp>(arg1.real(), arg2.real());
}

template <typename fp>
void init_matrix(fp *M, oneapi::mkl::transpose trans, int64_t m, int64_t n,
                 int64_t ld) {
  if (trans == oneapi::mkl::transpose::nontrans) {
    for (int64_t j = 0; j < n; j++)
      for (int64_t i = 0; i < m; i++)
        M[i + j * ld] = fp(i + j);
  } else {
    for (int64_t i = 0; i < m; i++)
      for (int64_t j = 0; j < n; j++)
        M[j + i * ld] = fp(i + j);
  }
}

// GEMM operation is performed as shown below.
// C = alpha * op(A) * op(B) + beta * C

template <typename fp> void run_gemm(const sycl::device &dev) {
  // Initialize data for GEMM
  oneapi::mkl::transpose transA = oneapi::mkl::transpose::nontrans;
  oneapi::mkl::transpose transB = oneapi::mkl::transpose::nontrans;

  // Matrix data sizes
  int64_t m = 1024;
  int64_t n = 1024;
  int64_t k = 1024;
```

```
// Leading dimensions of matrices
int64_t ldA = 1024;
int64_t ldB = 1024;
int64_t ldC = 1024;

// Set scalar fp values
fp alpha = set_fp_value(2.0, -0.5);
fp beta = set_fp_value(3.0, -1.5);

// Create sub devices for multi stack device
std::vector<sycl::device> subdev = {};
sycl::queue subdev_queue0;
sycl::queue subdev_queue1;
try {
  using namespace sycl::info;
  subdev = dev.create_sub_devices<
      partition_property::partition_by_affinity_domain>(
      partition_affinity_domain::numa);
  std::cout << "\tUse " << subdev.size() << " subdevices" << std::endl;
} catch (sycl::exception &e) {
  std::cout << "\tSubdevices not found, use root device" << std::endl;
}

int64_t nb_device = subdev.size();

// Catch asynchronous exceptions
auto exception_handler = [](sycl::exception_list exceptions) {
  for (std::exception_ptr const &e : exceptions) {
    try {
      std::rethrow_exception(e);
    } catch (sycl::exception const &e) {
      std::cout << "Caught asynchronous SYCL exception during GEMM:\n"
                << e.what() << std::endl;
    }
  }
};

// Create context and execution queue
subdev.push_back(dev);
sycl::context cxt(subdev);
sycl::queue main_queue(cxt, dev, exception_handler);

// If two subdevices are detected, create queue for each subdevice
if (nb_device > 1) {
  subdev_queue0 = sycl::queue(cxt, subdev[0], exception_handler);
  subdev_queue1 = sycl::queue(cxt, subdev[1], exception_handler);
}

// Allocate and initialize arrays for matrices
int64_t sizea;
if (transA == oneapi::mkl::transpose::nontrans)
  sizea = ldA * k;
else
  sizea = ldA * m;
int64_t sizeb = ldB * n;
int64_t sizec = ldC * n;

auto A_host = sycl::malloc_host<fp>(sizea, main_queue);
auto B_host = sycl::malloc_host<fp>(sizeb, main_queue);
auto C_host = sycl::malloc_host<fp>(sizec, main_queue);

init_matrix(A_host, transA, m, k, ldA);
init_matrix(B_host, transB, k, n, ldB);
init_matrix(C_host, oneapi::mkl::transpose::nontrans, m, n, ldC);

// Copy A/B/C from host to device/subdevices
// When two stacks are detected,
// GEMM operation is split between stacks in n direction
// All A matrix is copied to both stacks.
```

```
// B and C matrices are split between stacks, so only half of B and C are
// copied to each stack.

fp *A_dev0, *A_dev1, *B_dev0, *B_dev1, *C_dev0, *C_dev1, *A_dev, *B_dev,
    *C_dev;

if (nb_device > 1) {
  A_dev0 = sycl::malloc_device<fp>(sizea, subdev_queue0);
  A_dev1 = sycl::malloc_device<fp>(sizea, subdev_queue1);
  B_dev0 = sycl::malloc_device<fp>(sizeb / 2, subdev_queue0);
  B_dev1 = sycl::malloc_device<fp>(sizeb / 2, subdev_queue1);
  C_dev0 = sycl::malloc_device<fp>(sizec / 2, subdev_queue0);
  C_dev1 = sycl::malloc_device<fp>(sizec / 2, subdev_queue1);
  // Entire A matrix is copied to both stacks
  subdev_queue0.memcpy(A_dev0, A_host, sizea * sizeof(fp));
  subdev_queue1.memcpy(A_dev1, A_host, sizea * sizeof(fp));
  // Half of B and C are copied to each stack
  subdev_queue0.memcpy(B_dev0, B_host, (sizeb / 2) * sizeof(fp));
  subdev_queue1.memcpy(B_dev1, B_host + ldB * n / 2,
                       (sizeb / 2) * sizeof(fp));
  subdev_queue0.memcpy(C_dev0, C_host, (sizec / 2) * sizeof(fp));
  subdev_queue1.memcpy(C_dev1, C_host + ldC * n / 2,
                       (sizec / 2) * sizeof(fp));
  subdev_queue0.wait();
  subdev_queue1.wait();
} else {
  A_dev = sycl::malloc_device<fp>(sizea, main_queue);
  B_dev = sycl::malloc_device<fp>(sizeb, main_queue);
  C_dev = sycl::malloc_device<fp>(sizec, main_queue);
  main_queue.memcpy(A_dev, A_host, sizea * sizeof(fp));
  main_queue.memcpy(B_dev, B_host, sizeb * sizeof(fp));
  main_queue.memcpy(C_dev, C_host, sizec * sizeof(fp));
  main_queue.wait();
}

// Call oneMKL GEMM API
// When two stacks are detected, GEMM call is launched on each stack
sycl::event gemm_done0;
sycl::event gemm_done1;
std::vector<sycl::event> gemm_dependencies;
try {
  if (nb_device > 1) {
    // Split B and C for each stack
    int64_t n_subdev = n / 2;
    gemm_done0 = oneapi::mkl::blas::gemm(
        subdev_queue0, transA, transB, m, n_subdev, k, alpha, A_dev0, ldA,
        B_dev0, ldB, beta, C_dev0, ldC, gemm_dependencies);
    gemm_done1 = oneapi::mkl::blas::gemm(
        subdev_queue1, transA, transB, m, n_subdev, k, alpha, A_dev1, ldA,
        B_dev1, ldB, beta, C_dev1, ldC, gemm_dependencies);
  } else {
    gemm_done0 = oneapi::mkl::blas::gemm(main_queue, transA, transB, m, n, k,
                                         alpha, A_dev, ldA, B_dev, ldB, beta,
                                         C_dev, ldC, gemm_dependencies);
  }
} catch (sycl::exception const &e) {
  std::cout << "\t\tCaught synchronous SYCL exception during GEMM:\n"
            << e.what() << std::endl;
}

// Wait for GEMM calls to finish
gemm_done0.wait();
if (nb_device > 1)
  gemm_done1.wait();

// Copy C from device/subdevices to host
if (nb_device > 1) {
  subdev_queue0.memcpy(C_host, C_dev0, (sizec / 2) * sizeof(fp));
  subdev_queue1.memcpy(C_host + ldC * n / 2, C_dev1,
```

```
                          (sizec / 2) * sizeof(fp));
    subdev_queue0.wait();
    subdev_queue1.wait();
  } else {
    main_queue.memcpy(C_host, C_dev, sizec * sizeof(fp));
    main_queue.wait();
  }

  // Clean-up
  free(A_host, cxt);
  free(B_host, cxt);
  free(C_host, cxt);
  if (nb_device > 1) {
    sycl::free(A_dev0, subdev_queue0);
    sycl::free(A_dev1, subdev_queue1);
    sycl::free(B_dev0, subdev_queue0);
    sycl::free(B_dev1, subdev_queue1);
    sycl::free(C_dev0, subdev_queue0);
    sycl::free(C_dev1, subdev_queue1);
  } else {
    sycl::free(A_dev, main_queue);
    sycl::free(B_dev, main_queue);
    sycl::free(C_dev, main_queue);
  }

  if (nb_device > 1)
    std::cout << "\tGEMM operation is complete on 2 stacks" << std::endl;
  else
    std::cout << "\tGEMM operation is complete on 1 stack" << std::endl;
}

// Main entry point for example.
// GEMM example is run on GPU for 4 data types.

int main(int argc, char **argv) {
  // Create GPU device
  sycl::device dev = sycl::device(sycl::gpu_selector_v);

  std::cout << "Running with single precision real data type:" << std::endl;
  run_gemm<float>(dev);

  std::cout << "Running with double precision real data type:" << std::endl;
  run_gemm<double>(dev);

  std::cout << "Running with single precision complex data type:" << std::endl;
  run_gemm<std::complex<float>>(dev);

  std::cout << "Running with double precision complex data type:" << std::endl;
  run_gemm<std::complex<double>>(dev);
  return 0;
}
```

この例では、GEMM の最初のスカラー・パラメーターと次元パラメーターが初期化されます。

```
  // Initialize data for GEMM
  oneapi::mkl::transpose transA = oneapi::mkl::transpose::nontrans;
  oneapi::mkl::transpose transB = oneapi::mkl::transpose::nontrans;

  // Matrix data sizes
  int64_t m = 1024;
  int64_t n = 1024;
  int64_t k = 1024;

  // Leading dimensions of matrices
  int64_t ldA = 1024;
  int64_t ldB = 1024;
  int64_t ldC = 1024;
```

```
// Set scalar fp values
fp alpha = set_fp_value(2.0, -0.5);
fp beta = set_fp_value(3.0, -1.5);
```

次に、利用可能であればルートデバイスからサブデバイスが作成され、SYCL* ベクトルに保存されます。プラットフォームにサブデバイスがない場合 (シングル・スタック・デバイスなど)、ルートデバイスが計算に使用されます。この例では、非同期例外を処理する方法も示しています。

```
// Create sub devices for multi stack device
std::vector<sycl::device> subdev = {};
sycl::queue subdev_queue0;
sycl::queue subdev_queue1;
try {
  using namespace sycl::info;
  subdev = dev.create_sub_devices<
      partition_property::partition_by_affinity_domain>(
      partition_affinity_domain::numa);
  std::cout << "\tUse " << subdev.size() << " subdevices" << std::endl;
} catch (sycl::exception &e) {
  std::cout << "\tSubdevices not found, use root device" << std::endl;
}

int64_t nb_device = subdev.size();

// Catch asynchronous exceptions
auto exception_handler = [](sycl::exception_list exceptions) {
  for (std::exception_ptr const &e : exceptions) {
    try {
      std::rethrow_exception(e);
    } catch (sycl::exception const &e) {
      std::cout << "Caught asynchronous SYCL exception during GEMM:\n"
                << e.what() << std::endl;
    }
  }
};
```

その後、コンテキストとキューが作成されます。サブデバイスが早期に検出された場合、サブデバイスごとに個別のキューが作成されます。

```
// Create context and execution queue
subdev.push_back(dev);
sycl::context cxt(subdev);
sycl::queue main_queue(cxt, dev, exception_handler);

// If two subdevices are detected, create queue for each subdevice
if (nb_device > 1) {
  subdev_queue0 = sycl::queue(cxt, subdev[0], exception_handler);
  subdev_queue1 = sycl::queue(cxt, subdev[1], exception_handler);
}
```

oneMKL 呼び出しが起動される前に、入力行列と出力行列を割り当てて、初期化してからホストからデバイスに転送する必要があります。2 つのサブデバイスが検出されると、最初のスタックで GEMM 計算の A 行列と B 行列の前半半分が実行され、後半は 2 番目のスタックで実行されます。したがって、それぞれに応じた行列を各スタックにコピーする必要があります。

```
// Allocate and initialize arrays for matrices
int64_t sizea;
if (transA == oneapi::mkl::transpose::nontrans)
  sizea = ldA * k;
else
  sizea = ldA * m;
int64_t sizeb = ldB * n;
int64_t sizec = ldC * n;
```

```
auto A_host = sycl::malloc_host<fp>(sizea, main_queue);
auto B_host = sycl::malloc_host<fp>(sizeb, main_queue);
auto C_host = sycl::malloc_host<fp>(sizec, main_queue);

init_matrix(A_host, transA, m, k, ldA);
init_matrix(B_host, transB, k, n, ldB);
init_matrix(C_host, oneapi::mkl::transpose::nontrans, m, n, ldC);

// Copy A/B/C from host to device/subdevices
// When two stacks are detected,
// GEMM operation is split between stacks in n direction
// All A matrix is copied to both stacks.
// B and C matrices are split between stacks, so only half of B and C are
// copied to each stack.

fp *A_dev0, *A_dev1, *B_dev0, *B_dev1, *C_dev0, *C_dev1, *A_dev, *B_dev,
    *C_dev;

if (nb_device > 1) {
  A_dev0 = sycl::malloc_device<fp>(sizea, subdev_queue0);
  A_dev1 = sycl::malloc_device<fp>(sizea, subdev_queue1);
  B_dev0 = sycl::malloc_device<fp>(sizeb / 2, subdev_queue0);
  B_dev1 = sycl::malloc_device<fp>(sizeb / 2, subdev_queue1);
  C_dev0 = sycl::malloc_device<fp>(sizec / 2, subdev_queue0);
  C_dev1 = sycl::malloc_device<fp>(sizec / 2, subdev_queue1);
  // Entire A matrix is copied to both stacks
  subdev_queue0.memcpy(A_dev0, A_host, sizea * sizeof(fp));
  subdev_queue1.memcpy(A_dev1, A_host, sizea * sizeof(fp));
  // Half of B and C are copied to each stack
  subdev_queue0.memcpy(B_dev0, B_host, (sizeb / 2) * sizeof(fp));
  subdev_queue1.memcpy(B_dev1, B_host + ldB * n / 2,
                       (sizeb / 2) * sizeof(fp));
  subdev_queue0.memcpy(C_dev0, C_host, (sizec / 2) * sizeof(fp));
  subdev_queue1.memcpy(C_dev1, C_host + ldC * n / 2,
                       (sizec / 2) * sizeof(fp));
  subdev_queue0.wait();
  subdev_queue1.wait();
} else {
  A_dev = sycl::malloc_device<fp>(sizea, main_queue);
  B_dev = sycl::malloc_device<fp>(sizeb, main_queue);
  C_dev = sycl::malloc_device<fp>(sizec, main_queue);
  main_queue.memcpy(A_dev, A_host, sizea * sizeof(fp));
  main_queue.memcpy(B_dev, B_host, sizeb * sizeof(fp));
  main_queue.memcpy(C_dev, C_host, sizec * sizeof(fp));
  main_queue.wait();
}
```

最後に、oneMKL GEMM API が対応するデバイス/サブデバイスのキューを使用して呼び出され、GEMM 呼び出しが完了すると、更新された C 行列がホストに転送されます。

```
// Call oneMKL GEMM API
// When two stacks are detected, GEMM call is launched on each stack
sycl::event gemm_done0;
sycl::event gemm_done1;
std::vector<sycl::event> gemm_dependencies;
try {
  if (nb_device > 1) {
    // Split B and C for each stack
    int64_t n_subdev = n / 2;
    gemm_done0 = oneapi::mkl::blas::gemm(
        subdev_queue0, transA, transB, m, n_subdev, k, alpha, A_dev0, ldA,
        B_dev0, ldB, beta, C_dev0, ldC, gemm_dependencies);
    gemm_done1 = oneapi::mkl::blas::gemm(
        subdev_queue1, transA, transB, m, n_subdev, k, alpha, A_dev1, ldA,
        B_dev1, ldB, beta, C_dev1, ldC, gemm_dependencies);
  } else {
    gemm_done0 = oneapi::mkl::blas::gemm(main_queue, transA, transB, m, n, k,
```

```
                                          alpha, A_dev, ldA, B_dev, ldB, beta,
                                          C_dev, ldC, gemm_dependencies);
    }
  } catch (sycl::exception const &e) {
    std::cout << "\t\tCaught synchronous SYCL exception during GEMM:\n"
              << e.what() << std::endl;
  }

  // Wait for GEMM calls to finish
  gemm_done0.wait();
  if (nb_device > 1)
    gemm_done1.wait();

  // Copy C from device/subdevices to host
  if (nb_device > 1) {
    subdev_queue0.memcpy(C_host, C_dev0, (sizec / 2) * sizeof(fp));
    subdev_queue1.memcpy(C_host + ldC * n / 2, C_dev1,
                         (sizec / 2) * sizeof(fp));
    subdev_queue0.wait();
    subdev_queue1.wait();
  } else {
    main_queue.memcpy(C_host, C_dev, sizec * sizeof(fp));
    main_queue.wait();
  }
```

この例を実行するには、MKLROOT 環境変数を設定した後に次のビルドコマンドを実行します。

```
$ icpx -fsycl -qmkl=sequential -DMKL_ILP64 -L${MKLROOT}/lib/intel64 -lsycl -lOpenCL -
lpthread -lm -ldl -o onemkl_dpcpp_gemm onemkl_dpcpp_gemm.cpp
$ export LD_LIBRARY_PATH=${MKLROOT}/lib/intel64/:${LD_LIBRARY_PATH}
$ ./onemkl_dpcpp_gemm
```

問題のサイズが大きい (m=n=k=8192) の場合、2 つのスタックで明示的にスケーリングすると、2 倍近いパフォーマンスのスケーリングが期待できます。

## マルチ GPU、マルチスタック、およびマルチ C スライスでの OpenMP* による明示的なスケーリング

### デバイスの検出

アプリケーションを実行する前に、`sycl-ls` コマンドを実行して、プラットフォームで利用可能なデバイスを確認してください。これは、パフォーマンスを測定する場合、特に重要です。これにより、実行のフォールバック・パスを確認できます。

### コンテキスト

OpenMP* コンテキストは暗黙的です。ランタイムはコンテキストを作成して管理しますが、相互運用サポート向けに OpenMP* コンテキスト (`omp_target_get_context()`) を照会するインテルの拡張機能が追加されています。

### 統合共有メモリー

USM メモリー割り当て向けのインテル拡張機能として、3 つの OpenMP* API が追加されました。USM をサポートする `omp_target_alloc` API も追加されています。詳細は、「OpenMP* USM 割り当て API」を参照してください。

## サブのサブデバイスに関連付けられたコンテキスト

単一のサブのサブデバイスを持つコンテキストを作成し、キューをそのサブのサブデバイス (計算スライス) に接続します。このスキームでは、実行と可視性はサブのサブデバイスのみに限定され、計算スライス (C スライス) レベルの粒度でのパーティション化と制御が提供されます。コード例を示します。

```
#define DEVKIND 1 // C スライス

int root_id = omp_get_default_device();

#pragma omp parallel for
for (int id = 0; id < NUM_SUBSUBDEVICES; ++id) {
#pragma omp target teams distribute parallel for device(root_id)                \
    subdevice(DEVKIND, id) map(…)
  for (int i = lb(id), i < ub(id); i++) {
    ...;
  }
}
```

サブのサブデバイスのスキームは、計算スライスレベルでスタック上の複数の小さなカーネルを同時実行する際に推奨されます。これらのカーネルは、スタックのすべての計算スライスを活用するには、十分な並列性がありません。インテル® データセンター GPU マックス・シリーズの各スタックには、カーネルを関連する計算スライスにディスパッチする 4 つの計算コマンド・ストリーマーがあります。スタックごとに 4 つの小さなカーネルを同時に実行できます。つまり、2 スタックシステムでは、8 つの小さなカーネルを同時に実行できます。

## サブデバイスに関連付けられたコンテキスト

同じルートデバイスで複数のサブデバイス (複数のスタック) を持つコンテキストを作成します。このスキームでは、明示的なスケーリングを効率良く実装するサブデバイスにキューが接続されます。この場合、コードは複数のサブデバイスで実行されます。コード例を示します。

```
#define DEVKIND 0 // タイル

int root_id = omp_get_default_device();

#pragma omp parallel for
for (int id = 0; id < NUM_SUBDEVICES; ++id) {
#pragma omp target teams distribute parallel for device(root_id)
    subdevice(DEVKIND, id) map(…)
  for (int i = lb(id), i < ub(id); i++) {
    …
  }
}
```

## ルートデバイスに関連付けられたコンテキスト

単一のルートデバイスとそれに接続されたキューを使用してコンテキストを作成すると、GPU ドライバーによる暗黙のスケーリングにより、ワークはすべてのサブデバイス (スタック) に自動分散されます。これは、特定のスタックをターゲットにしないマルチスタックを最も効率良く利用する方法です。コード例を示します。

```
int root_id = omp_get_default_device();

#pragma omp target teams distribute parallel for device(root_id) map(…)
for (int i = 0, i < N; i++) {
  …
}
```

## 複数のルートデバイスに関連付けられたコンテキスト

複数のルートデバイス (複数カード) でオフロード領域を起動します。このスキームでは、キューが異なるルートデバイスに接続される最も自由度の高いコンテキストが、ホストメモリーや明示的なコピーによる低速アクセスと引き換えに共有の可能性を高めます。コード例を示します。

```
Int num_devices = omp_get_num_devices();

#pragma omp parallel for
for (int root_id = 0; root_id < num_devices; root_id++) {
#pragma omp target teams distribute parallel for device(root_id) map(…)
  for (int i = lb(root_id); I < ub(root_id); i++) {
    …
  }
}
```

明示的に指定するサブデバイスとアルゴリズムに応じて、適切なメモリー割り当て/同期を行います。次のプログラムは、明示的にサブデバイスを使用する例です。

```
1.  #include <assert.h>
2.  #include <iostream>
3.  #include <omp.h>
4.  #include <stdint.h>
5.  #ifndef NUM_SUBDEVICES
6.  #define NUM_SUBDEVICES 1
7.  #endif
8.
9.  #ifndef DEVKIND
10. #define DEVKIND 0
11. #endif
12.
13. template <int num_subdevices> struct mptr {
14.   float *p[num_subdevices];
15. };
16.
17. int main(int argc, char **argv) {
18.   constexpr int SIZE = 8e6;
19.   constexpr int SIMD_SIZE = 32;
20.   constexpr std::size_t TOTAL_SIZE = SIZE * SIMD_SIZE;
21.   constexpr int num_subdevices = NUM_SUBDEVICES;
22.
23.   mptr<num_subdevices> device_ptr_a;
24.   mptr<num_subdevices> device_ptr_b;
25.   mptr<num_subdevices> device_ptr_c;
26.
27.   const int device_id = omp_get_default_device();
28.   std::cout << "device_id = " << device_id << std::endl;
29.
30.   for (int sdev = 0; sdev < num_subdevices; ++sdev) {
31.     device_ptr_a.p[sdev] =
32.         static_cast<float *>(malloc(TOTAL_SIZE * sizeof(float)));
33.     device_ptr_b.p[sdev] =
34.         static_cast<float *>(malloc(TOTAL_SIZE * sizeof(float)));
35.     device_ptr_c.p[sdev] =
36.         static_cast<float *>(malloc(TOTAL_SIZE * sizeof(float)));
37.
38. #pragma omp target enter data map(alloc                                    \
39.                                  : device_ptr_a.p[sdev] [0:TOTAL_SIZE])       \
40.     device(device_id) subdevice(DEVKIND, sdev)
41. #pragma omp target enter data map(alloc                                    \
42.                                  : device_ptr_b.p[sdev] [0:TOTAL_SIZE])       \
43.     device(device_id) subdevice(DEVKIND, sdev)
44. #pragma omp target enter data map(alloc                                    \
45.                                  : device_ptr_c.p[sdev] [0:TOTAL_SIZE])       \
```

```
46.     device(device_id) subdevice(DEVKIND, sdev)
47.   }
48.   std::cout << "memory footprint per GPU = "
49.             << 3 * (std::size_t)(TOTAL_SIZE) * sizeof(float) * 1E-9 << " GB"
50.             << std::endl;
51.
52. #pragma omp parallel for
53.   for (int sdev = 0; sdev < num_subdevices; ++sdev) {
54.     float *a = device_ptr_a.p[sdev];
55.     float *b = device_ptr_b.p[sdev];
56.
57. #pragma omp target teams distribute parallel for device(device_id)             \
58.     subdevice(LEVEL, sdev)
59.     for (int i = 0; i < TOTAL_SIZE; ++i) {
60.       a[i] = i + 0.5;
61.       b[i] = i - 0.5;
62.     }
63.   }
64.
65.   const int no_max_rep = 200;
66.   double time = 0.0;
67.   for (int irep = 0; irep < no_max_rep + 1; ++irep) {
68.     if (irep == 1)
69.       time = omp_get_wtime();
70.
71. #pragma omp parallel for num_threads(num_subdevices)
72.     for (int sdev = 0; sdev < num_subdevices; ++sdev) {
73.       float *a = device_ptr_a.p[sdev];
74.       float *b = device_ptr_b.p[sdev];
75.       float *c = device_ptr_c.p[sdev];
76.
77. #pragma omp target teams distribute parallel for device(device_id)             \
78.     subdevice(LEVEL, sdev)
79.       for (int i = 0; i < TOTAL_SIZE; ++i) {
80.         c[i] = a[i] + b[i];
81.       }
82.     }
83.   }
84.
85.   time = omp_get_wtime() - time;
86.   time = time / no_max_rep;
87.
88.   const std::size_t streamed_bytes =
89.       3 * (std::size_t)(TOTAL_SIZE)*num_subdevices * sizeof(float);
90.   std::cout << "bandwidth = " << (streamed_bytes / time) * 1E-9 << " GB/s"
91.             << std::endl;
92.   std::cout << "time = " << time << " s" << std::endl;
93.   std::cout.precision(10);
94.
95.   for (int sdev = 0; sdev < num_subdevices; ++sdev) {
96. #pragma omp target update from(device_ptr_c.p[sdev][:TOTAL_SIZE])              \
97.     device(device_id) subdevice(LEVEL, sdev)
98.     std::cout << "-GPU: device id = : " << sdev << std::endl;
99.     std::cout << "target result:" << std::endl;
100.    std::cout << "c[" << 0 << "] = " << device_ptr_c.p[sdev][0] << std::endl;
101.    std::cout << "c[" << SIMD_SIZE - 1
102.              << "] = " << device_ptr_c.p[sdev][SIMD_SIZE - 1] << std::endl;
103.    std::cout << "c[" << TOTAL_SIZE / 2
104.              << "] = " << device_ptr_c.p[sdev][TOTAL_SIZE / 2] << std::endl;
105.    std::cout << "c[" << TOTAL_SIZE - 1
106.              << "] = " << device_ptr_c.p[sdev][TOTAL_SIZE - 1] << std::endl;
107.  }
108.
109.  for (int sdev = 0; sdev < num_subdevices; ++sdev) {
110.    for (int i = 0; i < TOTAL_SIZE; ++i) {
111.      assert((int)(device_ptr_c.p[sdev][i]) ==
112.             (int)(device_ptr_c.p[sdev][i] +
113.                   device_ptr_a.p[sdev][i] * device_ptr_b.p[sdev][i]));
114.    }
```

```
115.  }
116.
117.  for (int sdev = 0; sdev < num_subdevices; ++sdev) {
118.#pragma omp target exit data map(release : device_ptr_a.p[sdev][:TOTAL_SIZE])
119.#pragma omp target exit data map(release : device_ptr_b.p[sdev][:TOTAL_SIZE])
120.#pragma omp target exit data map(release : device_ptr_a.p[sdev][:TOTAL_SIZE])
121.  }
122.}
```

OpenMP* ワークロードの明示的なスケーリングでは、事前 (AOT) コンパイルを使用してインテル® データセンター GPU マックス・シリーズ搭載システムで最大 2 倍の線形スケーリングを達成できます。以下にビルドコマンドを示します。

```
$ icpx -fp-contract=fast -O2 -ffast-math -DNUM_SUBDEVICES=2 -fopenmp \
-fopenmp-targets=spir64_gen -Xopenmp-target-backend "-device pvc" \
openmp_explicit_subdevice.cpp -o run.x
```

`subdevice` 節の言語拡張に加え、環境変数が OpenMP* オフロード・コンパイラーとランタイムによって提供されるため、OpenMP* `device` を GPU カード (デバイス)、スタック (サブデバイス) および計算スライス (サブのサブデバイス) にマップできます。環境変数は、`LIBOMPTARGET_DEVICES=[device | subdevice | subsubdevice]` です。

## OpenMP* でインテル® oneAPI マス・カーネル・ライブラリーを使用した明示的なスケーリング

このセクションでは、OpenMP* オフロードの例として、マルチ・スタック・プラットフォーム上でインテル® oneAPI マス・カーネル・ライブラリー (oneMKL) で明示的なスケーリングを行う方法について説明します。

次のコードは、AXPY の計算を 2 つのスタックに分割する方法を示しています。この例では、`cblas_daxpy` API による明示的なスケーリングを示します。

```
#include "mkl.h"
#include "mkl_omp_offload.h"
#include <omp.h>
#include <stdio.h>

int main() {

  double *x, *y, *y_ref, alpha;
  MKL_INT n, incx, incy, i;

  // Initialize data for AXPY
  alpha = 1.0;
  n = 8192;
  incx = 1;
  incy = 1;

  // Allocate and initialize arrays for vectors
  x = (double *)mkl_malloc(n * sizeof(double), 128);
  y = (double *)mkl_malloc(n * sizeof(double), 128);
  if ((x == NULL) || (y == NULL)) {
    printf("Error in vector allocation\n");
    return 1;
  }
  for (i = 0; i < n; i++) {
    x[i] = rand() / (double)RAND_MAX - .5;
    y[i] = rand() / (double)RAND_MAX - .5;
  }

  printf("First 10 elements of the output vector Y before AXPY:\n");
  for (i = 0; i < 10; i++) {
    printf("%lf ", y[i]);
```

```
  }
  printf("\n\n");

  // Detect number of available stacks
  int nb_device = omp_get_num_devices();

  // Copy data to device and perform computation
  if (omp_get_num_devices() > 1) {
    printf("2 stacks are detected. AXPY operation is divided into 2 to take "
           "advantage of explicit scaling\n");
    printf("Copy x[0..%lld] and y[0..%lld] to stack 0\n", n / 2 - 1, n / 2 - 1);
#pragma omp target data map(to                                                 \
                            : x [0:n / 2]) map(tofrom                          \
                                               : y [0:n / 2]) device(0)
    {
      printf("Copy x[%lld..%lld] and y[%lld..%lld] to stack 1\n", n / 2, n - 1,
             n / 2, n - 1);
#pragma omp target data map(to                                                 \
                            : x [n / 2:n - n / 2]) map(tofrom                  \
                                                       : y [n / 2:n - n / 2])  \
    device(1)
      {
        double *x1 = &x[n / 2];
        double *y1 = &y[n / 2];
#pragma omp target variant dispatch use_device_ptr(x, y) device(0) nowait
        cblas_daxpy(n / 2, alpha, x, incx, y, incy);
#pragma omp target variant dispatch use_device_ptr(x1, y1) device(1)
        cblas_daxpy(n / 2, alpha, x1, incx, y1, incy);
#pragma omp taskwait
      }
    }
  } else {
    printf("1 stack is detected. Entire AXPY operation is performed on that "
           "stack\n");
    printf("Copy x[0..%lld] and y[0..%lld] to stack 0\n", n - 1, n - 1);
#pragma omp target data map(to : x [0:n]) map(tofrom : y [0:n]) device(0)
    {
#pragma omp target variant dispatch use_device_ptr(x, y) device(0)
      cblas_daxpy(n, alpha, x, incx, y, incy);
#pragma omp taskwait
    }
  }
  // End of computation

  printf("\nFirst 10 elements of the output vector Y after AXPY:\n");
  for (i = 0; i < 10; i++) {
    printf("%lf ", y[i]);
  }
  printf("\n");

  mkl_free(x);
  mkl_free(y);
  return 0;
}
```

この例では、最初に AXPY のパラメーターとベクトルが割り当てられ、初期化されます。

```
  // Initialize data for AXPY
  alpha = 1.0;
  n = 8192;
  incx = 1;
  incy = 1;

  // Allocate and initialize arrays for vectors
  x = (double *)mkl_malloc(n * sizeof(double), 128);
  y = (double *)mkl_malloc(n * sizeof(double), 128);
  if ((x == NULL) || (y == NULL)) {
    printf("Error in vector allocation\n");
```

```
    return 1;
  }
  for (i = 0; i < n; i++) {
    x[i] = rand() / (double)RAND_MAX - .5;
    y[i] = rand() / (double)RAND_MAX - .5;
  }

  printf("First 10 elements of the output vector Y before AXPY:\n");
  for (i = 0; i < 10; i++) {
    printf("%lf ", y[i]);
  }
  printf("\n\n");
```

次に、omp_get_num_devices API を使用して、利用可能なスタック数を検出します。

```
  // Detect number of available stacks
  int nb_device = omp_get_num_devices();
```

最後に、データが転送され、OpenMP* から oneMKL cblas_daxpy 呼び出しが GPU にオフロードされます。2 つのスタックが検出されると、x ベクトルと y ベクトルの半分ずつがそれぞれのスタックにコピーされ、明示的なスケーリングのため AXPY 計算が 2 つに分割されます。

```
  // Copy data to device and perform computation
  if (omp_get_num_devices() > 1) {
    printf("2 stacks are detected. AXPY operation is divided into 2 to take "
           "advantage of explicit scaling\n");
    printf("Copy x[0..%lld] and y[0..%lld] to stack 0\n", n / 2 - 1, n / 2 - 1);
#pragma omp target data map(to                                                 \
                            : x [0:n / 2]) map(tofrom                          \
                                               : y [0:n / 2]) device(0)
    {
      printf("Copy x[%lld..%lld] and y[%lld..%lld] to stack 1\n", n / 2, n - 1,
             n / 2, n - 1);
#pragma omp target data map(to                                                 \
                            : x [n / 2:n - n / 2]) map(tofrom                  \
                                                       : y [n / 2:n - n / 2])  \
    device(1)
      {
        double *x1 = &x[n / 2];
        double *y1 = &y[n / 2];
#pragma omp target variant dispatch use_device_ptr(x, y) device(0) nowait
        cblas_daxpy(n / 2, alpha, x, incx, y, incy);
#pragma omp target variant dispatch use_device_ptr(x1, y1) device(1)
        cblas_daxpy(n / 2, alpha, x1, incx, y1, incy);
#pragma omp taskwait
      }
    }
  } else {
    printf("1 stack is detected. Entire AXPY operation is performed on that "
           "stack\n");
    printf("Copy x[0..%lld] and y[0..%lld] to stack 0\n", n - 1, n - 1);
#pragma omp target data map(to : x [0:n]) map(tofrom : y [0:n]) device(0)
    {
#pragma omp target variant dispatch use_device_ptr(x, y) device(0)
      cblas_daxpy(n, alpha, x, incx, y, incy);
#pragma omp taskwait
    }
  }
```

この例を実行するには、MKLROOT 環境変数を設定した後に次のビルドコマンドを使用します。`LIBOMPTARGET_DEVICES=subdevice` は、OpenMP* からサブデバイス/スタックにアクセスできるようにするため必要になります。

```
$ export LIBOMPTARGET_DEVICES=subdevice
$ icpx -fiopenmp -fopenmp-targets=spir64 -m64 -DMKL_ILP64  -qmkl-ilp64=parallel -
lstdc++ -o onemkl_openmp_axpy onemkl_openmp_axpy.cpp
$ export LD_LIBRARY_PATH=${MKLROOT}/lib/intel64/:${LD_LIBRARY_PATH}
$ ./onemkl_openmp_axpy
```

大きな問題サイズ (m=150000000) の場合、2 つのスタックで明示的にスケーリングすると、2 倍近いパフォーマンスのスケーリングが期待できます。

### 明示的なスケーリングのまとめ

マルチスタック GPU のパフォーマンス・チューニングは、並列化の粒度が細かくなるため、面倒な作業が発生します。しかし、基本は CPU のパフォーマンス・チューニングと同様です。パフォーマンス・スケーリングを理解するには、次の点に注意してください。

- VE 利用効率 - カーネルが異なるスタックの実行リソースをどのように利用するか
- データ配置 - 割り当てが異なるスタックの HBM にどのように分散されるか
- スレッドデータのアフィニティー - データがどこに配置され、システム内でどのようにアクセスされるか

さらに、アプリケーション開発者が生産性、移植性、およびパフォーマンスのため、スケーリング・スキームを選択する際に留意すべき重要なプログラミング・モデルの概念があります。

- サブデバイス (`numa_domains`) とサブのサブデバイス (`subnuma_domains`)
- 暗黙のおよび明示的なスケーリング
- コンテキストとキュー
- 環境変数とプログラミング言語の API や構造

## GPU クラスター向けのインテル® MPI

GPGPU は HPC クラスターで一般的になり、インテル® MPI ライブラリーは、ヘテロジニアス・マシンでの大規模アプリケーションのスケールアップおよびスケールアウト実行をサポートするようになりました。GPU サポートは、インテル® MPI ライブラリー 2019 Update 8 で導入されました (「インテル® MPI ライブラリー・リリースノート」(英語) を参照)。その後、いくつかの機能強化が行われています。特に明記しない限り、この節で示すすべての検証はインテル® MPI ライブラリー 2021 Update 5 を使用しています。

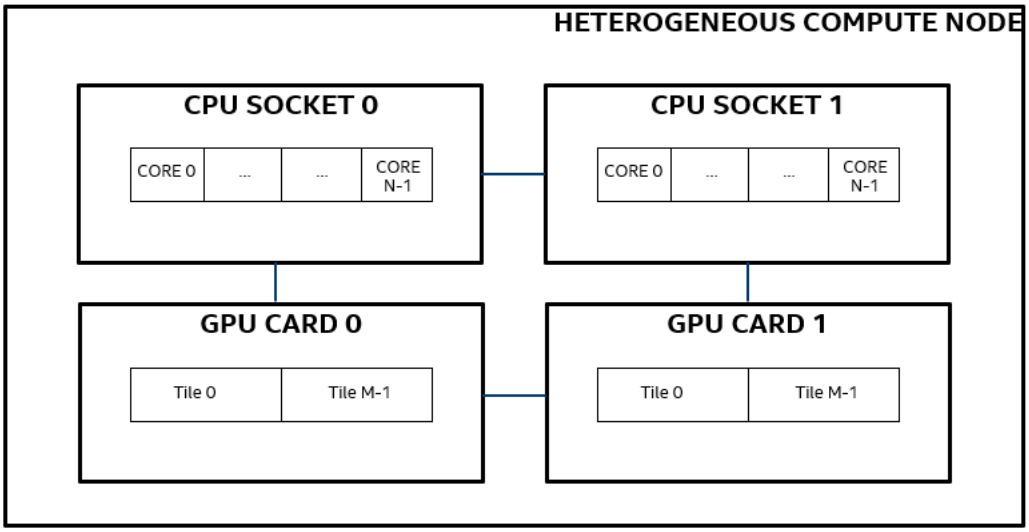
この節では、インテル® MPI ライブラリーにおける以下の GPU 固有の機能について説明します。

- トポロジーの検出
- 実行モデル
- GPU のピニング
- マルチスタック、マルチ GPU、およびマルチノードのサポート
- IMB の GPU ベンチマーク
- GPU で MPI アプリケーションをプロファイル
- 推奨事項

## トポロジーの検出

次の図では、2 つの CPU ソケットを備えたシステムの一般的な GPU 計算ノードを示しています。各 CPU ソケットには N 個のコアがあり、M 個のスタックを持つ単一の GPU カードに接続され、ノードあたり 2M スタックがあります。

### 一般的な GPU 計算ノード




上記のトポロジーにはバリエーションがあります。`cpuinfo` コマンドを使用して、詳細を表示します。

```
$ cpuinfo -g
```

```
Intel(R) processor family information utility, Version 2021.5 Build
20211102 (id: 9279b7d62)

Copyright (C) 2005-2021 Intel Corporation. All rights reserved.

===== Processor composition =====
Processor name : Intel(R) Xeon(R) Gold 6346
Packages(sockets) : 2
Cores : 32
Processors(CPUs) : 64
Cores per package : 16
Threads per core : 2
```

この例では、ソケットあたり 16 コアのデュアルソケットのノードがあり、ノードあたりの 32 個の物理コアがあります。

```
$ clinfo -l
```

```
Platform #0: Intel(R) FPGA Emulation Platform for OpenCL(TM)
 `-- Device #0: Intel(R) FPGA Emulation Device
Platform #1: Intel(R) OpenCL
 `-- Device #0: Intel(R) Xeon(R) Gold 6346 CPU @ 3.10GHz
Platform #2: Intel(R) OpenCL HD Graphics
 +-- Device #0: Intel(R) Graphics [0x020a]
 `-- Device #1: Intel(R) Graphics [0x020a]
```

ここでは、プラットフォーム #2 の下に、PCI ID 020a の 2 つの GPU カードが存在することが分かります。プラットフォームの開発コード名を確認するには、「インテル® グラフィックス・プロセッサーの表」(英語) を参照してください。

```
$ clinfo

… …
Number of devices 2
Device Name Intel(R) Graphics [0x020a] …
…

Device Partition (core)
Max number of sub-devices 2 …
…
```

上記に示す出力は、GPU カードごとに 2 つのサブデバイス (スタック) があることを示しています。この情報をまとめるとノードのトポロジーを理解できます。

## 実行モデル

開発者は、さまざまな方法で GPU 環境でインテル® MPI ライブラリーを利用できます。最も一般的ないくつかのシナリオを以下に示します。

## ナイーブ (自然な)

このモデルでは、MPI 関数はホスト (汎用 CPU) のみで実行され、ベースとなる MPI ライブラリーはデバイス (ここでは GPU) を認識する必要はありません。下図に示すように、開発者はホストとデバイス間のデータ移動 (`to:` と `tofrom:` を使用) を処理する責任があります。このモデルでは、MPI 通信を頻繁に行うアプリケーションでは、ホストとデバイス間での過度なデータ移動が発生する可能性があります。ただし、このアプローチはメッセージサイズが小さな場合は有益です。

### OpenMP* を使用したナイーブ実行モデル

```
// Mark data to be copied from host to device and device to host
#pragma omp target data map(to: rank, num_values) map(tofrom:values[0:num_values])
{
   // Compute on GPU
   #pragma omp target parallel for
   for (i = 0; i < num_values; ++i) {
       values[i] = values[i] + rank + 1;
       printf("[Within GPU:] values[%d]=%d \n",i,values[i]);
       }
}
    //Send device buffer to rank 0 after copy back from GPU
     MPI_Send(values, num_values, MPI_INT, dest_rank, tag, MPI_COMM_WORLD);
```

上図では、`MPI_Send` が両方のオフロードプラグマの範囲外にあることに注意してください。ホストとデバイス間のデータ移動には、SYCL* と OpenMP* オフロード構造の両方が使用できます。

## GPU を意識した MPI

このモデルでは、ベースとなる MPI ライブラリーが GPU バッファーを認識するため、開発者はホストとデバイス間で明示的なデータ移動タスクから解放されます。ただし、オフロード可能なメモリー領域のポインターを MPI に渡すには、特定のコンパイラー・ディレクティブを使用するか、対応するアクセラレーターのランタイム API から取得する必要があります。インテル® MPI ライブラリーは、必要に応じてデータを移動するだけではなく、ホストとデバイス間のデータ移動を最適化し、データ移動のコストを可能な限り隠匿します。MPI 呼び出しはホスト上で実行されます。GPU バッファーをサポートするため、`I_MPI_OFFLOAD` 環境変数が導入されました。次の図のように、`MPI_Send` は `omp target parallel for` プラグマのスコープ外にありますが、データ移動を指示する `target` 句がスコープ内にあります。また、ポインター値は `use_device_ptr` と `is_device_ptr` キーワードでマークされ、ポインター値はインテル® MPI ライブラリーで適切に処理されます。インテル® MPI ライブラリーの GPU 対応機能は、SYCL* と OpenMP* オフロードを使用するアプリケーションで使用できます。

### OpenMP* オフロードを使用した GPU 対応実行モデル

```
// Mark data to be copied from host to device
#pragma omp target data map(to: rank, values[0:num_values], num_values) use_device_ptr(values)
{
   // Compute on GPU
   #pragma omp target parallel for is_device_ptr(values)
   for (i = 0; i < num_values; ++i) {
       values[i] = values[i] + rank + 1;
       printf("[Within GPU:] values[%d]=%d \n",i,values[i]);
       }

    // Send device buffer to rank 0 without the use of from:values
    MPI_Send(values, num_values, MPI_INT, dest_rank, tag, MPI_COMM_WORLD);
}
```

#### ビルドコマンド

```
$ mpiicc -cc=icx -qopenmp -fopenmp-targets=spir64 test.c -o test
```

ここで、`-cc=icx` は従来の `icc` コンパイラー・ドライバーの代わりに、オフロード対応のコンパイラー・ドライバーを指定します。`-qopenmp` は、LLVM で OpenMP* コンパイルをサポートするミドルエンドを有効にします。そして、`-fopenmp-targets=spir64` は、コンパイラーが GPU デバイスバイナリーに `x86 + SPIR64*` ファットバイナリーを生成することを指示します。

#### 実行コマンド

```
$ export I_MPI_OFFLOAD=1
$ mpiexec.hydra -n 2 ./test
```

`I_MPI_OFFLOAD` は、MPI 関数のデバイスバッファー処理を制御します。値を 1 に設定すると、`libze_loader.so` がロードされている前提でデバイスバッファーの処理が有効にされます。ライブラリーがロードされていない場合 (これは予期されていません)、GPU バッファーのサポートは有効になりません。インテル® MPI ライブラリーが `libze_loader.so` をロードするように強制するには、`I_MPI_OFFLOAD` を 2 に設定します。

## カーネルベースの集合

GPU 対応の MPI モデルとは異なり、このモデルはデバイスによるカーネルの MPI スケールアップ実行を直接サポートします。このアプローチは、ホストで MPI 呼び出しが実行されたホストとデバイス間のデータ転送を抑制するのに役立ちます。インテル® oneAPI コレクティブ・コミュニケーション・ライブラリー (インテル® oneCCL) (英語) などの MPI に類似した実装では、集合呼び出しに似た機能がすでにサポートされています。

## GPU のピニング

ヘテロジニアス (CPU + デバイス) MPI プログラムでは、CPU コアを MPI ランクに割り当てるだけではなく、利用可能な GPU リソースをランクに割り当てる必要もあります。インテル® MPI ライブラリーのコンテキストは、デフォルトでインテル製ディスクリート GPU の基本ワークの単位はスタックであると想定します。ユーザーが必要とするピニングスキームを実現するため、いくつかの環境変数が導入されています。インテル® MPI ライブラリーは、デフォルトでバランスが取れたピニングスキームを適用します。これにより、ノードあたりのランク数がスタック数と等しい場合、スタックごとに 1 つのランクが割り当てられます。オーバーサブスクリプションの場合、ラウンドロビン・スキームにより、最適なバランスの分散が行われます。インテル® MPI ライブラリーは、GPU ピニングを強制するため GPU のトポロジーを必要とします。これは、インテル® oneAPI レベルゼロ (英語) を使用して取得できます。次の環境変数は、トポロジーの検出にレベルゼロを使用することを指定します。

```
$ export I_MPI_OFFLOAD_TOPOLIB=level_zero
```

インテル® MPI ライブラリー 2021 Update6 以降では、`I_MPI_OFFLOAD > 0` の場合 `I_MPI_OFFLOAD_TOPOLIB` は自動的に `level_zero` に設定されます。

デバッグレベル 3 では、インテル® MPI ライブラリーはアプリケーションに適用される GPU ピニングスキームを出力します。

```
$ export I_MPI_DEBUG=3
```

以下は、4 スタックのマシンで、4 つのランクを実行するアプリケーションのデバッグ出力の例です。

```
[0] MPI startup(): ===== GPU pinning on host1 =====
[0] MPI startup(): Rank Pin stack
[0] MPI startup(): 0      {0}
[0] MPI startup(): 1      {1}
[0] MPI startup(): 2      {2}
[0] MPI startup(): 3      {3}
```

デバッグレベル 120 では、配置に関連する詳細が表示されます。

```
$ export I_MPI_DEBUG=120
```

以下は、4 スタックのマシンで、4 つのランクを実行するアプリケーションのデバッグ出力の例です。

```
[0] MPI startup(): ===== GPU Placement on packages =====
[0] MPI startup(): NUMA Id GPU Id Stacks Ranks
[0] MPI startup(): 0       0      (0,1)  0,1
[0] MPI startup(): 1       1      (2,3)  2,3
```

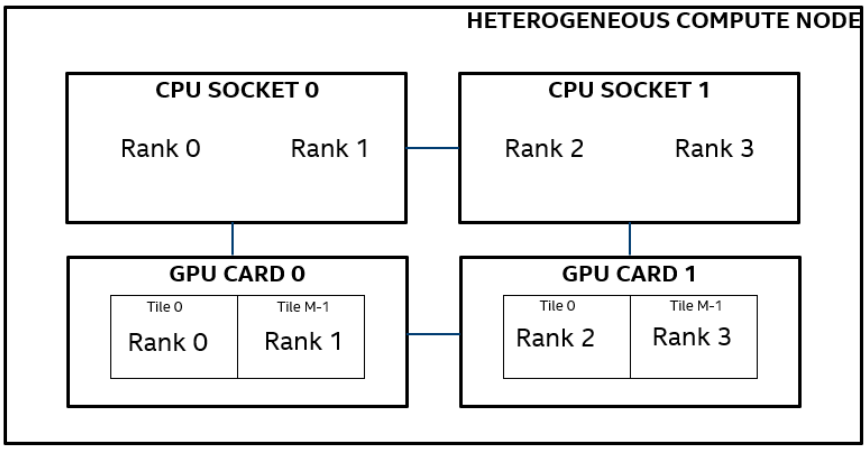
「GPU placement on packages」にある情報は、GPU のピニング情報として扱うべきではありません。これは、ランクが NUMA ノードにどのように配置されるかを示します。ソケットあたり 1 つの NUMA ドメインしかないノードでは、NUMA ノードは CPU ソケットとして扱うことができます。この前提で、上記のデバッグ情報は次のように解釈できます。

- GPU 0 は CPU ソケット 0 に接続される。
- GPU 0 はスタック 0 と 1 を持つ。
- ランク 0 と 1 は CPU ソケット 0 に配置される。

同様に、

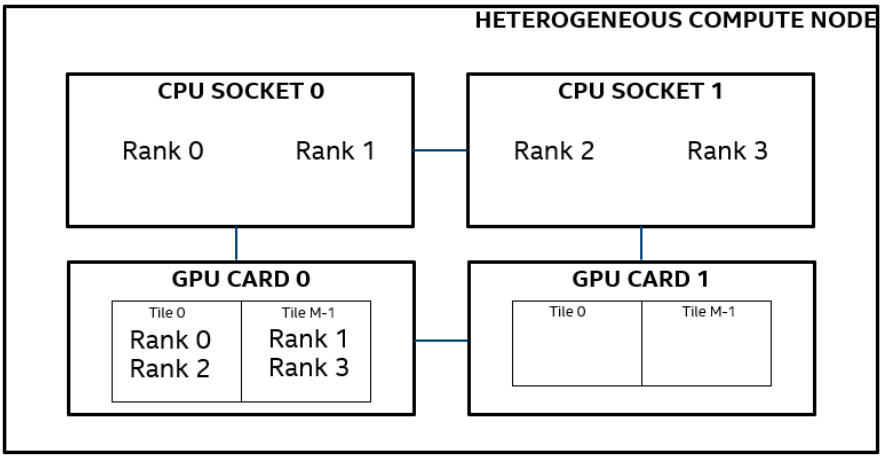
- GPU 1 は CPU ソケット 1 に接続される。
- GPU 1 はスタック 2 と 3 を持つ。
- ランク 2 と 3 は CPU ソケット 1 に配置される。

### デフォルトのピニングスキームの表現図




明確にするため、別の例を考えてみます。上の図で示すノードに対し、実行を GPU 0 のみに限定するため、次の環境変数を設定します。

### ユーザー定義のピニングスキームの表現図




```
$ export I_MPI_OFFLOAD_DEVICES=0
```

次のデバッグレベル 120 の出力にあるように、ランク 0、1 およびランク 2、3 は、それぞれ CPU ソケット 0 と 1 にあります。`I_MPI_OFFLOAD_DEVICES` が 0 に設定されているため、すべてのランクがラウンドロビン方式で GPU 0 のスタックに固定されています。上の図を参照してください。

```
[0] MPI startup(): ===== GPU Placement on packages =====
[0] MPI startup(): NUMA Id GPU Id Stacks Ranks
[0] MPI startup(): 0       0      (0,1)  0,1
[0] MPI startup(): 1       1      (2,3)  2,3
```

```
[0] MPI startup(): ===== GPU pinning on host1 =====
[0] MPI startup(): Rank Pin stack
[0] MPI startup(): 0      {0}
[0] MPI startup(): 1      {1}
[0] MPI startup(): 2      {0}
[0] MPI startup(): 3      {1}
```

CPU 0 へのカスタムピニングのスキームを定義する I_MPI_PIN_PROCESOR_LIST 環境変数と同様に、I_MPI_OFFLOAD_DEVICE_LIST 環境変数も GPU のピニングに使用できます。さらに詳細なピニング制御には、I_MPI_OFFLOAD_CELL、I_MPI_OFFLOAD_DOMAIN_SIZE、I_MPI_OFFLOAD_DOMAIN といった環境変数を使用します。詳細は、「GPU ピニング」(英語) のドキュメントを参照してください。

## マルチスタック、マルチ GPU、およびマルチノードのサポート

最新のヘテロジニアス (CPU + GPU) クラスターには、さまざまな GPU 固有のスケーリングの可能性があります。

1. スタック・スケーリング: 同一 GPU カード内のスタック間でのスケーリング (GPU 内、スタック間)。
2. スケールアップ: 同じノードに接続された複数の GPU カード間でのスケーリング (ノード内、GPU 間)。
3. スケールアウト: 異なるノード間に接続された GPU カード間のスケーリング (GPU 間、ノード間)。

これらのシナリオはすべてインテル® MPI ライブラリーによって透過的に処理されるため、アプリケーション開発者はソースコードを変更する必要はありません。次に、それぞれのシナリオのコマンドラインを示します。

## ベースライン: すべてのランクをスタック 0 で実行

```
$ export I_MPI_OFFLOAD_DEVICE_LIST=0
$ export I_MPI_DEBUG=120
$ export I_MPI_OFFLOAD=1
$ export I_MPI_OFFLOAD_TOPOLIB=level_zero
$ mpiexec.hydra -n 2 ./mpi-binary

[0] MPI startup(): ===== GPU Placement on packages =====
[0] MPI startup(): NUMA Id GPU Id Stacks Ranks
[0] MPI startup(): 0 0 (0,1) 0
[0] MPI startup(): 1 1 (2,3) 1
[0] MPI startup(): ===== GPU pinning on host1 =====
[0] MPI startup(): Rank Pin stack
[0] MPI startup(): 0 {0}
[0] MPI startup(): 1 {0}
```

## スタックのスケーリング: GPU 0 のスタックごとに 1 ランクを実行

```
$ export I_MPI_OFFLOAD_DEVICE_LIST=0,1
$ export I_MPI_DEBUG=120
$ export I_MPI_OFFLOAD=1
$ export I_MPI_OFFLOAD_TOPOLIB=level_zero
$ mpiexec.hydra -n 2 ./mpi-binary

[0] MPI startup(): ===== GPU Placement on packages =====
[0] MPI startup(): NUMA Id GPU Id Stacks Ranks
[0] MPI startup(): 0 0 (0,1) 0
[0] MPI startup(): 1 1 (2,3) 1
[0] MPI startup(): ===== GPU pinning on host1 =====
[0] MPI startup(): Rank Pin stack
[0] MPI startup(): 0 {0}
[0] MPI startup(): 1 {1}
```

## スケールアップ: GPU 0 と GPU 1 のスタックごとに 1 ランクを実行

```
$ export I_MPI_DEBUG=120
$ export I_MPI_OFFLOAD=1
$ export I_MPI_OFFLOAD_TOPOLIB=level_zero
$ mpiexec.hydra -n 4 ./mpi-binary

[0] MPI startup(): ===== GPU Placement on packages =====
[0] MPI startup(): NUMA Id GPU Id Stacks Ranks
[0] MPI startup(): 0 0 (0,1) 0,1
[0] MPI startup(): 1 1 (2,3) 2,3
[0] MPI startup(): ===== GPU pinning on host1 =====
[0] MPI startup(): Rank Pin stack
[0] MPI startup(): 0 {0}
[0] MPI startup(): 1 {1}
[0] MPI startup(): 2 {2}
[0] MPI startup(): 3 {3}
```

## スケールアウト: ホスト 1 の GPU 0 と 1、ホスト 2 の GPU 2 と 3 上のスタックごとに 1 ランクを実行

```
$ export I_MPI_DEBUG=120
$ export I_MPI_OFFLOAD=1
$ export I_MPI_OFFLOAD_TOPOLIB=level_zero
$ mpiexec.hydra -n 8 -ppn 4 -hosts host1,host2 ./mpi-binary


[0] MPI startup(): ===== GPU Placement on packages =====
[0] MPI startup(): NUMA Id GPU Id Stacks Ranks
[0] MPI startup(): 0 0 (0,1) 0,1
[0] MPI startup(): 1 1 (2,3) 2,3
[0] MPI startup(): ===== GPU pinning on host1 =====
[0] MPI startup(): Rank Pin stack
[0] MPI startup(): 0 {0}
[0] MPI startup(): 1 {1}
[0] MPI startup(): 2 {2}
[0] MPI startup(): 3 {3}
```

ここには示しませんが、同様のピニングが `host2` でも発生しました。

## IMB の GPU ベンチマーク

インテル® MPI ベンチマーク (IMB) (英語) は、さまざまなメッセージサイズでポイントツーポイントおよび集合通信操作を行う一連の計測を行います。生成されたベンチマーク・データは、クラスターのパフォーマンス特性と MPI 実装の効率を評価するのに役立ちます。従来、IMB ベンチマークは CPU でのみ実行されていましたが、最近 GPU がサポートされるようになりました。`IMB-MPI1-GPU` と呼ばれるビルド済みバイナリーが、インテル® MPI ライブラリー・パッケージに含まれます (Linux* マシンの `$I_MPI_ROOT/bin` フォルダー)。このバイナリーは、前述の GPU 対応 MPI モデルで実行できます。

以下は、デフォルトの 0 から 4MB のメッセージサイズで `IMB-MPI1-GPU` を実行し、2 つの GPU カードを搭載した単一ノードでメッセージサイズごとに 200 回の反復を行うコマンドラインの例です。ここでは、`MPI_Allreduce` 関数のみが実行されていますが、ほかにも多くの MPI 関数がサポートされています。

```
$ export I_MPI_DEBUG=120
$ export I_MPI_OFFLOAD=1
$ export I_MPI_OFFLOAD_TOPOLIB=level_zero
$ mpiexec.hydra -n 4 -ppn 4 IMB-MPI1-GPU allreduce -mem_alloc_type device -npmin 4 -
iter 200 -iter_policy off
```

`-mem_alloc_type` オプションは、MPI バッファーが存在する場所を制御します。インテル® MPI ライブラリー 2021 Update4 以降では、`-mem_alloc_type` のデフォルトは `device` です。

`-npmin` 4 は、IMB の MPI ランクの最小数を 4 に制限します (指定しない場合、`-n` で指定される MPI ランク数に段階的に到達します)。

`-iter` 200 は、メッセージサイズごとの反復回数を 200 に指定します。

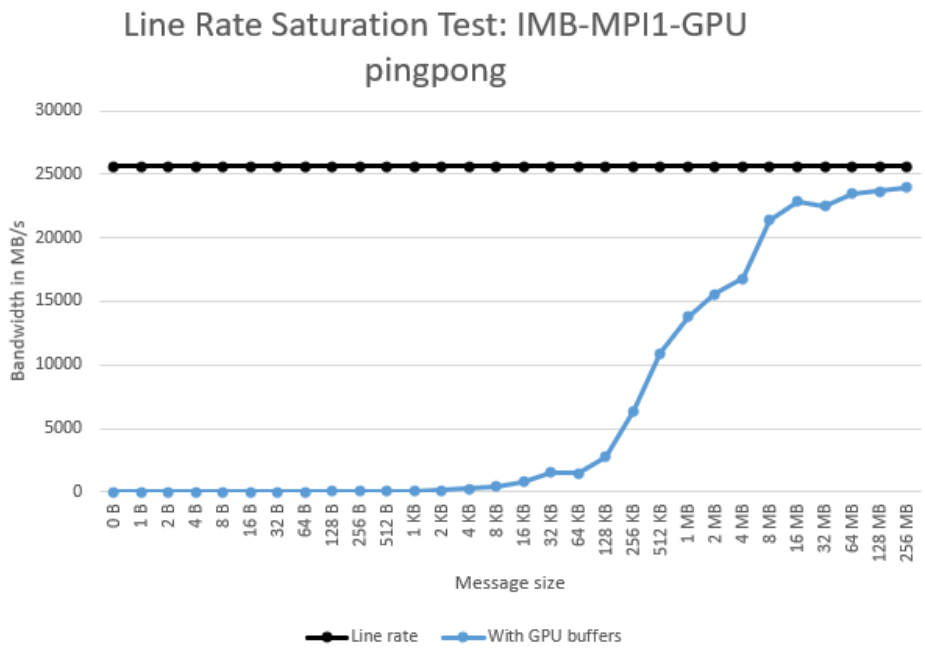
`-iter_policy off` は、大きなメッセージサイズの反復の自動リダクションを無効にします。つまり、`-iter_policy off` は、すべてのメッセージサイズで、`-iter` 200 の要求に従うことを保証します。

## ライン飽和テスト

この項では、GPU バッファーを使用する際にインターコネクトの帯域幅を飽和させるインテル® MPI ライブラリーの機能を説明します。ここでは、デュアルソケットのインテル® Xeon® Platinum 8480+ プロセッサーと 2 つのインテル® データセンター GPU マックス 1550 (Ponte Vecchio†) でテストしています。ノードは、定格 200Gb/秒もしくは 25GB/秒の Mellanox Quantum* HDR インターコネクトで接続されています。次のコマンドに示すように、2 つの MPI ランク (ノードごとに 1 ランク) が起動され、IMB-MPI1-GPU バイナリーのピンポンテストが選択されています。

```
$ I_MPI_DEBUG=5 I_MPI_OFFLOAD=2
$ mpiexec.hydra -n 2 -ppn 1 -hosts host1,host2 IMB-MPI1-GPU pingpong -iter 100 -
iter_policy off -msglog 0:28
```

**GPU バッファーを使用した IMB ベースのライン比率の飽和テスト。多くの要因がパフォーマンスに影響します。パフォーマンスは、ハードウェアとソフトウェアの構成によって異なります。測定されるパフォーマンスは、ここで示す測定値とは異なる場合があります。**




ここでのライン飽和のテストには、インテル® MPI ライブラリー 2021 Update8 が使用されています。23,948.98Mb/秒のピーク帯域幅が計測されました。これは、93.6% のライン比率に相当します。

## GPU で MPI アプリケーションをプロファイル

インテル® VTune™ プロファイラー、インテル® Advisor、そしてインテル® Inspector などのツールは、インテル® GPU で動作するように拡張されています。CPU のプロファイルと同様に、これらのツールは、GPU コンテキストの MPI アプリケーションで引き続き利用できます。ここでは、それらの機能の一部を MPI アプリケーションのコンテキストで紹介します。

また、前の節で紹介した IMB-MPI1-GPU ベンチマークについて検討します。MPI バッファーが GPU メモリーに割り当てられているかどうかを確認するには、PCIe* トラフィックをチェックします。GPU は PCIe* バスを介して CPU に接続されるため (`-mem_alloc_type` CPU と比較して) トラフィックが増加します。インテル® VTune™ プロファイラーの入出力解析では、いくつかの PCIe* メトリックをキャプチャーして表示できます。

リモートマシン上のインテル® VTune™ プロファイラーを操作するモードがサポートされています。ここで示すプロファイル・データは、リモートマシンのコマンドライン・インターフェイスから収集されたものです。リモートマシンの結果フォルダーは、ローカル GUI で表示するようローカルマシンに移動されます。次のコマンドラインは、GPU ノードを持つリモートマシンで実行されます。

**CPU のバッファー割り当て**

```
$ export I_MPI_OFFLOAD=1
$ export I_MPI_OFFLOAD_TOPOLIB=level_zero
$ mpiexec.hydra -n 4 -ppn 4 -f hostfile -gtool "vtune -collect io -r ./vtune_data1:0"
IMB-MPI1-GPU allreduce -npmin 4 -iter 200 -iter_policy off -mem_alloc_type CPU
```

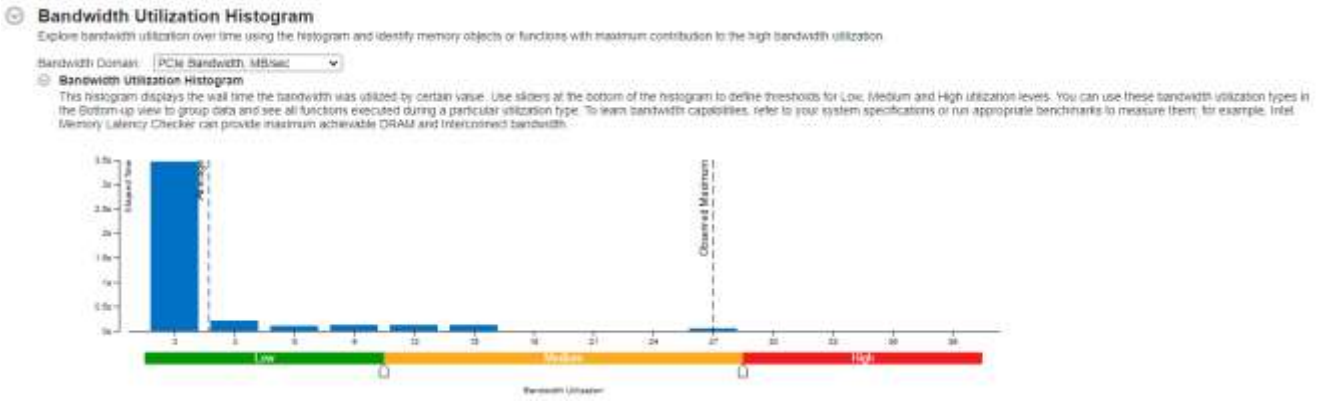
**GPU のバッファー割り当て**

```
$ export I_MPI_OFFLOAD=1
$ export I_MPI_OFFLOAD_TOPOLIB=level_zero
$ mpiexec.hydra -n 4 -ppn 4 -f hostfile -gtool "vtune -collect io -r ./vtune_data2:0"
IMB-MPI1-GPU allreduce -npmin 4 -iter 200 -iter_policy off
```

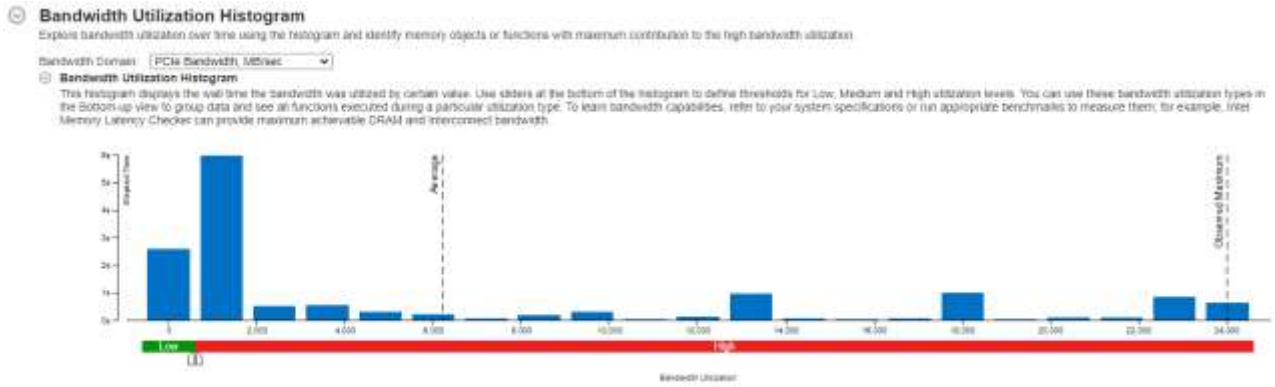
ここでは、`-gtool` オプションを使用して、ランク 0 のみでインテル® VTune™ プロファイラーを起動しています (`-gtool` の最後の引数 :0 でランクを指定)。

入出力解析の **[Summary (サマリー)]** タブの **[Bandwidth Utilization Histogram (帯域幅利用率ヒストグラム)]** (下図) に示すように、MPI バッファーがデバイスに割り当てられていると、PCIe* トラフィックが大幅に増加していることが分かります。`-mem_alloc_type device` で取得された最大帯域幅は 24,020MB/秒であるのに対し、CPU バッファーでは 27MB/秒でした。

### CPU バッファー割り当てによる PCIe* 帯域幅の低い利用率



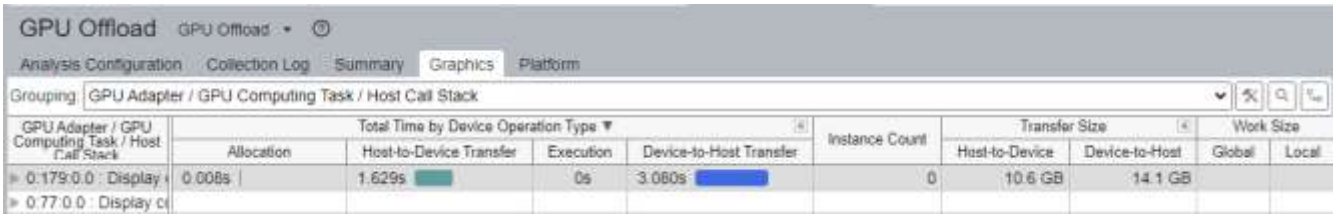
### GPU バッファー割り当てによる PCIe* 帯域幅の高い利用率




ホストとデバイス間で転送されるデータ量を確認するには、インテル® VTune™ プロファイラーの GPU オフロード解析を使用します。次の図に示すように、**[Graphics (グラフィックス)]** タブにホストからデバイス、およびデバイスからホストへの転送時間と転送量が表示されます。GPU オフロード解析を行うには、次のコマンドラインを使用します。

```
$ export I_MPI_DEBUG=120 I_MPI_OFFLOAD=1
$ export I_MPI_OFFLOAD_TOPOLIB=level_zero
$ mpiexec.hydra -n 4 -ppn 4 -f hostfile -gtool "vtune -collect gpu-offload -knob
collect-host-gpu-bandwidth=true -r ./vtune_data3:0" IMB-MPI1-GPU allreduce -npmin 4 -
iter 200 -iter_policy off
```

### GPU オフロード解析




この項では、ヘテロジニアス変数で動作する MPI アプリケーションのコンテキストで、解析ツールが使用できることを実証します。インテル® VTune™ プロファイラーの GPU 固有の各種機能の詳細は、『インテル® VTune™ プロファイラー・ユーザーガイド』を参照してください。前述したように、`-gtool` オプションを使用すると、インテル® Advisor やインテル® Inspector などほかの解析ツールも MPI アプリケーションで利用できます。詳細については、『インテル® Advisor ユーザーガイド』、『インテル® Inspector ユーザーガイド (Linux* 版)』(英語) を参照してください。

## 推奨事項

1. MPI 通信では、可能な限り同じ GPU バッファーを再利用することを推奨します。インテル® MPI ライブラリーのバッファー作成はコストの高い操作であるため、必要のないバッファーの割り当てと解除は避けるべきです。
2. アプリケーションが、非ブロッキングのポイントツーポイント操作に複数の GPU バッファーを使用する場合、インテル® MPI ライブラリーの並列バッファー数を増やすことでパフォーマンスを向上できます。これには、環境変数 `I_MPI_OFFLOAD_BUFFER_NUM` を使用します。デフォルト値は 4 であり、インテル® MPI ライブラリーは 4 ユーザーのデバイスバッファーを並列に処理します。この値を増やすとメモリー消費量も多くなります。
3. アプリケーションが大きなデバイスバッファーを使用する場合、インテル® MPI ライブラリーが使用するスクラッチバッファーのサイズを増やすことでパフォーマンスが向上することがあります。これには、環境変数 `I_MPI_OFFLOAD_BUFFER_SIZE` を使用します。デフォルト値は 8MB です。この値を増やすとメモリー消費量も多くなります。

4. GPU 対応モデルでは、インテル® MPI ライブラリーはパフォーマンス向上のためパイプラインを使用します。メッセージサイズが大きな場合に、環境変数 `I_MPI_OFFLOAD_PIPELINE_THRESHOLD` を使用して、パイプラインのアルゴリズムを有効にするタイミングを制御できます。この変数のデフォルト値は 524288 バイトです。ユーザーは、利用環境に最適な値を設定できるようこの制御を最適化できます。
5. 環境変数 `I_MPI_OFFLOAD_CACHE_TOTAL_MAX_SIZE` を使用すると、インテル® MPI ライブラリーで使用するすべての内部キャッシュのサイズを増やすことができます。デフォルト値は 67108864 バイトです。アプリケーションが多くの GPU バッファーを使用する場合、すべてをキャッシュできないため、この変数の値を増やすとパフォーマンスが向上する可能性があります。

## 参考資料

1. インテル® MPI ライブラリー・リリースノート (英語)
2. インテル® グラフィックス・プロセッサーの表 (英語)
3. インテル® oneAPI レベルゼロ (英語)
4. インテル® oneCCL (英語)
5. GPU のピニング (英語)
6. インテル® MPI ベンチマーク (英語)
7. インテル® VTune™ プロファイラー・ユーザーガイド
8. インテル® Advisor ユーザーズガイド
9. インテル® Inspector ユーザーズガイド Linux* (英語)

## 用語

| 用語 | 略語 | 定義 |
|---|---|---|
| ブリッター | BLT | ブロックイメージ転送。 |
| 子スレッド | | 別のスレッドで作成された分岐ノードまたはリーフノードのスレッド。これはメディア固定機能パイプラインに関連付けられたスレッドの一種です。子スレッドは、VE で実行される (親) スレッドから生成され、TS ユニットからスレッド・ディスパッチャーに転送されます。子スレッドは、ブランチノードのスレッドであるか、リーフノードのスレッドであるかにより、子スレッドを持つ場合と持たない場合があります。子スレッドの URB やスクラッチメモリーなど、事前に割り当てられたリソースは親スレッドで管理されます。「親スレッド」を参照してください。 |
| コマンド | | コマンド・ストリーマーによってメモリー内のリングバッファーからフェッチされ、パイプラインにルーティングされるディレクティブ。命令キャッシュ・サブシステムによってフェッチされ、VE で実行される命令とは異なります。 |
| コマンド・ストリーマー | CS または CSI | コマンドをフェッチして解析し、適切なパイプラインにルーティングするグラフィックス処理エンジンの機能ユニット。 |
| コア | | マルチプロセッサー・システムにおける VE の別名。「EU」を参照。 |
| デュアル・サブスライス | DSS | サンプラー、データポート、ピクセルポートなどの共有機能ユニットの共通セットを持つベクトルエンジン (VE)、またはエグゼキューション・ユニット (EU) の集合。 |
| 最後のスレッド | EOT | メッセージを要求するスレッドが終了したことを示す、出力メッセージバス上のメッセージ側波信号。スレッドが正しく終了するには、メッセージ記述フィールドの EOT ビットが指定された SEND 命令が最低 1 つ必要です。 |

| 用語 | 略語 | 定義 |
|---|---|---|
| 例外 | | スレッド命令の VE/EU 実行に対する中断タイプ。例外が発生すると、VE/EU スレッドは例外を処理するシステムルーチンを実行します。 |
| 実行チャンネル | | SIMD オペランドの単一レーン。 |
| 実行サイズ | ExecSize | SIMD 命令で処理されるデータ要素数。これは、命令フィールドの 1 つであり、命令ごとに変更できます。 |
| エグゼキューション・ユニット | EU | マルチプロセッサー・システム内のマルチスレッド・プロセッサー。各 EU は、命令フェッチとデコード、レジスターファイル、ソースオペランドのスウィズル、SIMD ALU などを含む完全な機能を備えたプロセッサーであり、EU はコアとも呼ばれます。 |
| エグゼキューション・ユニット識別子 | EUID | スレッド状態レジスター (SR0) 内の 4 ビットのフィールド。スレッドが配置されている EU の行と列を識別します。スレッドは EUID と TID によって一意に識別できます。 |
| 実行幅 | ExecWidth | 単一の SIMD 命令が処理できるデータ要素の幅。 |
| 汎用レジスターファイル | GRF | オペランドのソースとデスティネーション向けにすべての EU/VE で共有され読み取り/書き込み可能なラージ・レジスター・ファイル。これは、256 ビット・レジスターの配列として編成された、最も汎用的に使用される読み取り/書き込みレジスター空間です。 |
| 汎用状態ベースアドレス | | メモリーに常駐する「状態データ」ブロックのグラフィックス・アドレス、状態ブロック、スクラッチベース、定数バッファー、カーネルプログラムが含まれます。メモリーブロックの内容は、汎用状態ベース・アドレス・レジスターの内容からのオフセットで参照されます。「グラフィックス処理エンジン」を参照してください。 |
| グラフィックス処理エンジン | GPE | サブシステム、3D およびメディア・パイプライン、コマンド・ストリーマーの総称。 |
| メモリーマップド入出力 | MMIO | CPU/GPU と周辺機器との間で入出力を行う方法。 |
| メッセージ | | スレッドから別のスレッド、別の共有変数、または別の固定関数に送信されるデータパッケージ。メッセージパッシングは、インテル® GPU アーキテクチャーの主要な通信メカニズムです。 |
| 親スレッド | | スレッド生成階層におけるルートノード、または分岐ノードに対応するスレッド。親スレッドは、スレッド生成階層内の位置に応じて、ルートスレッドまたは子スレッドになります。 |
| リソース・ストリーマー | RS | グラフィックス処理エンジンの機能ユニット。リングバッファー内のコマンドを調査し、グラフィックス処理に残された、長いレイテンシーの項目の前処理を試みます。 |
| ルートスレッド | | ルートノードのスレッド。スレッド生成階層のルートノードに対応するスレッド。これはメディア固定機能パイプラインに関連付けられたスレッドの一種です。ルートスレッドは、VFE ユニットから生成され、TS ユニットからスレッド・ディスパッチャーに転送されます。ルートスレッドは、子スレッドを持つ場合と持たない場合があります。ルートスレッドは、TS によって管理されるスクラッチメモリーを持つ場合があります。子スレッドを持つルートスレッドの URB のリソースは、VFE によって管理されます。 |
| 単一命令複数データ | SIMD | 命令レベルでのデータ並列処理を利用する並列処理アーキテクチャー。また、このようなアーキテクチャーの命令を記述したり、特定の命令 (SIMD8 など) のデータ並列処理を記述できます。 |
| スポーン | | EU/VE で実行するスレッドを開始します。3D パイプラインの FF ユニットと同様に、スレッドのスポーンで実行されます。 |

| 用語 | 略語 | 定義 |
|---|---|---|
| サブレジスター | | SIMD レジスターのサブフィールド。レジスターファイルまたはレジスタータイプ向けにアラインされた固定サイズのレジスターです。例えば、GRF レジスター r2 は、256 ビット幅で、256 ビットにアラインされたレジスターです。サブレジスター r2.3:d は、GRF レジスター r2 の 4 つ目の dword です。 |
| サブシステム | | 共有機能や EU/VE など、FF ユニットが共有するリソースの名称。 |
| サーフェス | | テクスチャー、バッファー、レンダーターゲットなどのレンダリング・オペランドやデスティネーション。 |
| サーフェス状態 | | レンダーサーフェスに関連付けられた状態。 |
| サーフェス状態ベースポインター | | バインドテーブルおよびサーフェス状態データを参照する際に使用するベースアドレス。 |
| 同期されたルートスレッド | | 「ルートスレッドのディスパッチ」メッセージに応じて、TS からディスパッチされるルートスレッド。 |
| システム IP | SIP | すべてのスレッドに対して 1 つのグローバルシステム IP レジスターがある。スレッドから見ると、これは読み取り専用の仮想レジスターです。例外が発生すると、ハードウェアは何らかの処理を行ってから SIP にジャンプします。 |
| システムルーチン | | 例外を処理する命令シーケンス。SIP はこのルーチンに関連付けられてプログラムされており、例外が発生したすべてのスレッドはこのルーチンを呼び出します。 |
| スレッド | | EU/VE 上のカーネルプログラムで実行されたインスタンス。スレッドのライフサイクルは、スレッド・ディスパッチャーから EU/VE にディスパッチされた後、最初の命令から始まり、最後の命令 (スレッドの終了を示す EOT **送信**命令) の実行まで続きます。システム内のスレッドは、互いに独立することも、メッセージ・ゲートウェイの共有機能により相互に通信することもできます。 |
| スレッド・ディスパッチャー | TD、TDL | 固定機能ユニットからのスレッド開始要求を調停し、EU/VE でスレッドをインスタンス化する機能ユニット。 |
| スレッド識別子 | TID | スレッドが EU/VE 上のいずれのスレッドスロットを占有するか識別する、スレッド状態レジスター (SR0) のフィールド。スレッドは EUID と TID によって一意に識別できます。 |
| スレッドペイロード | | スレッドが実行を開始する前に、データがスレッドの GRF (r0 から) に事前ロードされます。このデータは通常、スポーン・エンティティー (FF ユニット) によって提供される制御情報と URB から読み取られたデータの組み合わせです。 |
| スレッド起動 | TS | メディア・パイプラインの 2 番目で、最後の固定関数ステージ。汎用/メディア処理に変わって新しいスレッドを開始します。 |
| 同期されないルートスレッド | | TS によって自動的にディスパッチされるルートスレッド。 |

# レベルゼロ

レベルゼロは、oneAPI ライブラリーがターゲットデバイスのハードウェアの機能を活用できるようにする、低レベルのインターフェイスです。この節では、oneAPI レベルゼロのインテル® グラフィックス計算ランタイムで採用されているアーキテクチャーの設計について説明します。実装の詳細と最適化のガイドライン、およびサポートされるプラットフォームで利用可能なさまざまな機能について説明します。


- 即時コマンドリスト


## 即時コマンドリスト

### はじめに

即時コマンドは、低レイテンシーの送信モデルを可能にする、レベルゼロ仕様で提供される機能です。このスキームでは、カーネルの起動やメモリーコピーなど、コマンドリストに追加されたコマンドを実行するため直ちにデバイスに送信します。これは、複数のコマンドを結合して同時に送信する通常のコマンドリストとは異なります。

通常のコマンドリストと比較すると、即時コマンドリストには次のような特徴があります。

- 即時コマンドリストは暗黙のコマンドキューであり、コマンドキュー記述子を使用して作成されます。
- 即時コマンドリストに追加されたコマンドは、送信されるとデバイス上ですぐに実行されます。
- 即時コマンドリストは、クローズしたり、リセットする必要はありません。
- 即時コマンドリストに関連付けられたコマンド・キュー・ハンドルがないため、`zeCommandQueueSynchronize` や `zeFenceHostSynchronize` を介して即時コマンドリストを同期することはできません。即時コマンドリストに送信されたコマンドが完了したことを確認するには、イベントを使用することを推奨します。

即時コマンドリストの意図は、デバイスに極めて軽量な送信インターフェイスを提供することであり、小規模または短時間のカーネル起動を反復して複数回実行するようなワークロードに適しています。そのようなワークロードの例には、HPC 環境や ML/DL フレームワークなどがあげられます。

### プログラミング・モデル

次のコードは、即時コマンドリストを作成し、カーネルを送信する方法を示します。同期は、イベント状態を確認することで行われます。

```
ze_command_queue_desc_t cmdQueueDesc = {ZE_STRUCTURE_TYPE_COMMAND_QUEUE_DESC};
cmdQueueDesc.pNext = nullptr;
cmdQueueDesc.flags = 0;
cmdQueueDesc.priority = ZE_COMMAND_QUEUE_PRIORITY_NORMAL;
cmdQueueDesc.ordinal = queueGroupOrdinal;
cmdQueueDesc.index = 0;
cmdQueueDesc.mode = ZE_COMMAND_QUEUE_MODE_ASYNCHRONOUS;
zeCommandListCreateImmediate(context, device, &cmdQueueDesc, &cmdList);

zeCommandListAppendLaunchKernel(cmdList, kernel, &dispatchTraits,
                                events[0], 0, nullptr);
// 非同期モードでは、同期にイベントを使用
zeEventHostSynchronize(events[0], std::numeric_limits<uint64_t>::max() - 1);
```

即時コマンドリストは、インオーダー・キューを実装するのに使用することもできます。この場合、リストに送信されたコマンドは、以下に示すように、イベントを使用してチェーンされます。

```
zeCommandListAppendMemoryCopy(cmdList, deviceBuffer, hostBuffer, allocSize,
                              events[0],
                              0, nullptr);

zeCommandListAppendMemoryCopy(cmdList, stackBuffer, deviceBuffer, allocSize,
                              events[1],
                              1,
                              &events[0]);

zeEventHostSynchronize(events[1], std::numeric_limits<uint64_t>::max() - 1));
```

通常のコマンドリストと同様に、即時コマンドリストも同期できます。この場合、同期は暗黙的に行われます。リストに送信されたそれぞれのコマンドはすぐに送信され、呼び出しから戻った時点で完了することが保証されます。

```
ze_command_queue_desc_t cmdQueueDesc = {ZE_STRUCTURE_TYPE_COMMAND_QUEUE_DESC};
cmdQueueDesc.pNext = nullptr;
cmdQueueDesc.flags = 0;
cmdQueueDesc.priority = ZE_COMMAND_QUEUE_PRIORITY_NORMAL;
cmdQueueDesc.ordinal = queueGroupOrdinal;
cmdQueueDesc.index = 0;
cmdQueueDesc.mode = ZE_COMMAND_QUEUE_MODE_SYNCHRONOUS;
zeCommandListCreateImmediate(context, device, &cmdQueueDesc, &cmdList);

zeCommandListAppendLaunchKernel(cmdList, kernel, &dispatchTraits,
                              nullptr, 0, nullptr);

// この時点でカーネルは実行済み
```

その他のサンプルコードについては、https://github.com/intel/compute-benchmarks (英語) リポジトリーを参照してください。これから始める方には `create_command_list_immediate_l0.cpp` や `execute_command_list_immediate_l0.cpp` などのシナリオが適しています。

# パフォーマンスのプロファイルと解析

システムの動作を理解することは、情報に基づいて最適化を選択する上で不可欠です。プロファイラー、アナライザー、デバッガーなどのツールはそのための十分な機能を備えています。インターバル・タイマー、カーネルタイマー、`print` 文などの機能は軽量ですが、最適化の過程で重要な役割を果たします。

以下の節では、ソフトウェアを最適化するためこれらのツールをどのように使用するかについて説明します。

- タイマーを使用
- インテル® VTune™ プロファイラー
- インテル® Advisor
- OpenCL* アプリケーション向けインテル® インターセプト・レイヤー
- GPU 向けのインテル® プロファイル・ツール・インターフェイス

## タイマーを使用

標準の C++ `chrono` ライブラリーは、SYCL* で各種精度の時間を追跡する際に利用できます。次の例は、`chrono` タイマークラスを使用して、ホストからカーネルの実行時間を計測する方法を示します。

```
1.  #include <CL/sycl.hpp>
2.  #include <iostream>
3.  using sycl;
4.
5.  // この例の配列タイプとデータサイズ
6.  constexpr size_t array_size = (1 << 16);
7.  typedef std::array<int, array_size> IntArray;
8.
9.  double VectorAdd(queue &q, const IntArray &a, const IntArray &b, IntArray &sum) {
10.    range<1> num_items{a.size()};
11.
12.    buffer a_buf(a);
13.    buffer b_buf(b);
14.    buffer sum_buf(sum.data(), num_items);
15.
16.    auto t1 = std::chrono::steady_clock::now();    // 開始時間
17.
18.    q.submit([&](handler &h) {
19.       // 入力アクセサー
20.       auto a_acc = a_buf.get_access<access::mode::read>(h);
21.       auto b_acc = b_buf.get_access<access::mode::read>(h);
22.
23.       // 出力アクセサー
24.       auto sum_acc = sum_buf.get_access<access::mode::write>(h);
25.
26.       h.parallel_for(num_items, [=](id<1> i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
27.    }).wait();
28.
29.    auto t2 = std::chrono::steady_clock::now();    // 終了時間
30.
31.    return(std::chrono::duration_cast<std::chrono::microseconds>(t2 - t1).count());
32. }
33.
34. void InitializeArray(IntArray &a) {
35.    for (size_t i = 0; i < a.size(); i++) a[i] = i;
36. }
37.
38. int main() {
39.   default_selector d_selector;
```

```
40.
41.   IntArray a, b, sum;
42.
43.   InitializeArray(a);
44.   InitializeArray(b);
45.
46.   queue q(d_selector);
47.
48.   std::cout << "Running on device: "
49.             << q.get_device().get_info<info::device::name>() << "\n";
50.   std::cout << "Vector size: " << a.size() << "\n";
51.
52.   double t = VectorAdd(q, a, b, sum);
53.
54.   std::cout << "Vector add successfully completed on device in " << t << "
    microseconds\n";
55.   return 0;
56. }
```

このタイミングは純粋にホスト側のものであることに注意してください。デバイス上のカーネルの実行は、ホストがカーネルを送信した後、かなり遅れて開始されることがあります。SYCL* は、カーネルの実行に消費した時間を追跡するプロファイル機能を提供します。

```
1.  #include <CL/sycl.hpp>
2.  #include <array>
3.  #include <iostream>
4.  using namespace sycl;
5.
6.  // この例の配列タイプとデータサイズ
7.  constexpr size_t array_size = (1 << 16);
8.  typedef std::array<int, array_size> IntArray;
9.
10. double VectorAdd(queue &q, const IntArray &a, const IntArray &b, IntArray &sum) {
11.    range<1> num_items{a.size()};
12.
13.    buffer a_buf(a);
14.    buffer b_buf(b);
15.    buffer sum_buf(sum.data(), num_items);
16.
17.    event e = q.submit([&](handler &h) {
18.       // 入力アクセサー
19.       auto a_acc = a_buf.get_access<access::mode::read>(h);
20.       auto b_acc = b_buf.get_access<access::mode::read>(h);
21.
22.       // 出力アクセサー
23.       auto sum_acc = sum_buf.get_access<access::mode::write>(h);
24.
25.       h.parallel_for(num_items, [=](id<1> i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
26.    });
27.    q.wait();
28.    return(e.template get_profiling_info<info::event_profiling::command_end>() -
29.        e.template get_profiling_info<info::event_profiling::command_start>());
30. }
31.
32. void InitializeArray(IntArray &a) {
33.    for (size_t i = 0; i < a.size(); i++) a[i] = i;
34. }
35.
36. int main() {
37.   default_selector d_selector;
38.
39.   IntArray a, b, sum;
40.
41.   InitializeArray(a);
42.   InitializeArray(b);
43.
```

```
44.   queue q(d_selector, property::queue::enable_profiling{});
45.
46.   std::cout << "Running on device: "
47.             << q.get_device().get_info<info::device::name>() << "\n";
48.   std::cout << "Vector size: " << a.size() << "\n";
49.
50.   double t = VectorAdd(q, a, b, sum);
51.
52.   std::cout << "Vector add successfully completed on device in " << t << "
    nanoseconds\n";
53.   return 0;
54. }
```

この例を実行すると、chrono タイマーを使用する時間が SYCL* プロファイル・クラスを使用する時間よりもかなり長くなる可能性があります。これは、SYCL* プロファイルにはホストとオフロードデバイス間のデータ転送時間が含まれないためです。

## インテル® VTune™ プロファイラー

インテル® VTune™ プロファイラーは、シリアルおよびマルチスレッド化されたアプリケーション向けのパフォーマンス解析ツールです。選択されたアルゴリズムを解析し、アプリケーションが利用可能なハードウェア・リソースからどのように利益を得られるか特定するのに役立ちます。インテル® VTune™ プロファイラーは次の場所を特定できます。

- 利用可能なプロセッサー・リソースを効率良く利用していないコード領域
- シーケンシャルとマルチスレッドの両方のパフォーマンスを最適化するのに最適なコード領域
- アプリケーションのパフォーマンスに影響する同期オブジェクト
- I/O 操作にアプリケーションが時間を費やしているかどうか、およびその場所と原因
- アプリケーションが CPU または GPU に依存するかどうか、およびコードを GPU に効率良くオフロードできるか
- 異なる同期メソッド、異なるスレッド数、または異なるアルゴリズムのパフォーマンスの影響
- スレッドのアクティビティーと移行
- データ共有、キャッシュミス、分岐予測ミスなど、コード内のハードウェアに関連する問題
- GPU で実行される SYCL* アプリケーションのプロファイル

GPU 解析をサポートする新しい機能も追加されています。

- GPU オフロード解析 (テクニカルプレビュー機能)
- GPU 計算/メディア・ホットスポット解析 (テクニカルプレビュー機能)

**GPU オフロード解析 (プレビュー)**

この解析タイプを使用して、プラットフォーム上の各種 CPU と GPU コアのコード実行を調査し、CPU と GPU のアクティビティーを関連付けて、アプリケーションが GPU 依存であるか CPU 依存であるかを特定します。このツールは、システムのすべてのコアクロックを自動的に調整し、同じ時間進行で GPU ベースのワークロードと CPU ベースのワークロードを解析できるようにします。この解析は以下を可能にします。

- アプリケーションが SYCL* または OpenCL* カーネルをどれくらい効率良く使用しているか特定できます。
- インテル® メディア SDK タスクの実行を時間経過で解析します (Linux* ターゲットのみ)。
- GPU の使用を調査して、すべての時点での GPU エンジンのソフトウェア・キューを解析できます。

**GPU 計算/メディア・ホットスポット解析 (プレビュー)**

このツールを使用し、最も時間のかかる GPU カーネルを解析し、GPU ハードウェア・メトリックに基づいて GPU の使用状況を特徴付け、メモリー・レイテンシーや非効率なカーネル・アルゴリズムによって引き起こされるパフォーマンスの問題を特定して、命令タイプごとの GPU 命令の頻度を解析します。GPU 計算/メディア・ホットスポット解析は、以下を可能にします。

- GPU 利用率の高い GPU カーネルを調査し、この利用率の有効性を予測してストールまたは低い占有率の原因を特定します。
- 選択した GPU メトリックで時間経過ごとにアプリケーションのパフォーマンスを調査します。
- 最もホットな SYCL* または OpenCL* カーネルを解析して、カーネルコードのアルゴリズムが不完全であるか、work-item の構成が適切ではないか確認します。
- SYCL* アプリケーションで GPU オフロード解析を実行します。

## インテル® VTune™ プロファイラーの GPU アプリケーション解析

1. インテル® VTune™ プロファイラー起動し、**[Welcome (ようこそ)]** ページで **[New Project (新規プロジェクト)]** をクリックします。**[Create a Project (プロジェクトを作成)]** ダイアログボックスが表示されます。
2. プロジェクトのプロジェクト名と場所を指定して、**[Create Project (プロジェクトを作成)]** ボタンをクリックします。**[Configure Analysis (解析の設定)]** ウィンドウが表示されます。
3. **[WHERE (どこを)]** ペインでローカルホストが選択されていることを確認します。
4. **[WHAT (何を)]** ペインで、起動するアプリケーションのターゲットが選択されていることを確認し、プロファイルするアプリケーションとして `matrix_multiply` バイナリーを指定します。
5. **[HOW (どのように)]** ペインの **[Accelerators (アクセラレーター)]** グループから **[GPU Offload analysis (GPU オフロード解析)]** を選択します。
6. **[Start (開始)]** ボタンをクリックして解析を実行します。

これは、インテル® VTune™ プロファイラーでサポートされるインテル® グラフィックスとサードパーティー製の GPU を搭載したプラットフォーム上で実行されるアプリケーションを解析する最も直感的な方法です。

## コマンドラインから解析を実行

コマンドラインから解析を実行するには、次の操作を行います。

Linux* :

- インテル® VTune™ プロファイラーの環境変数を設定するスクリプトを `source` します。

```
$ source <*install_dir*>/env/vars.sh
```

- 解析コマンドを実行します。

```
$ vtune -collect gpu-offload -- ./matrix.dpcpp
```

Windows*:

- インテル® VTune™ プロファイラーの環境変数を設定するバッチファイルを実行します。

```
$ <*install_dir*>\env\vars.bat
```

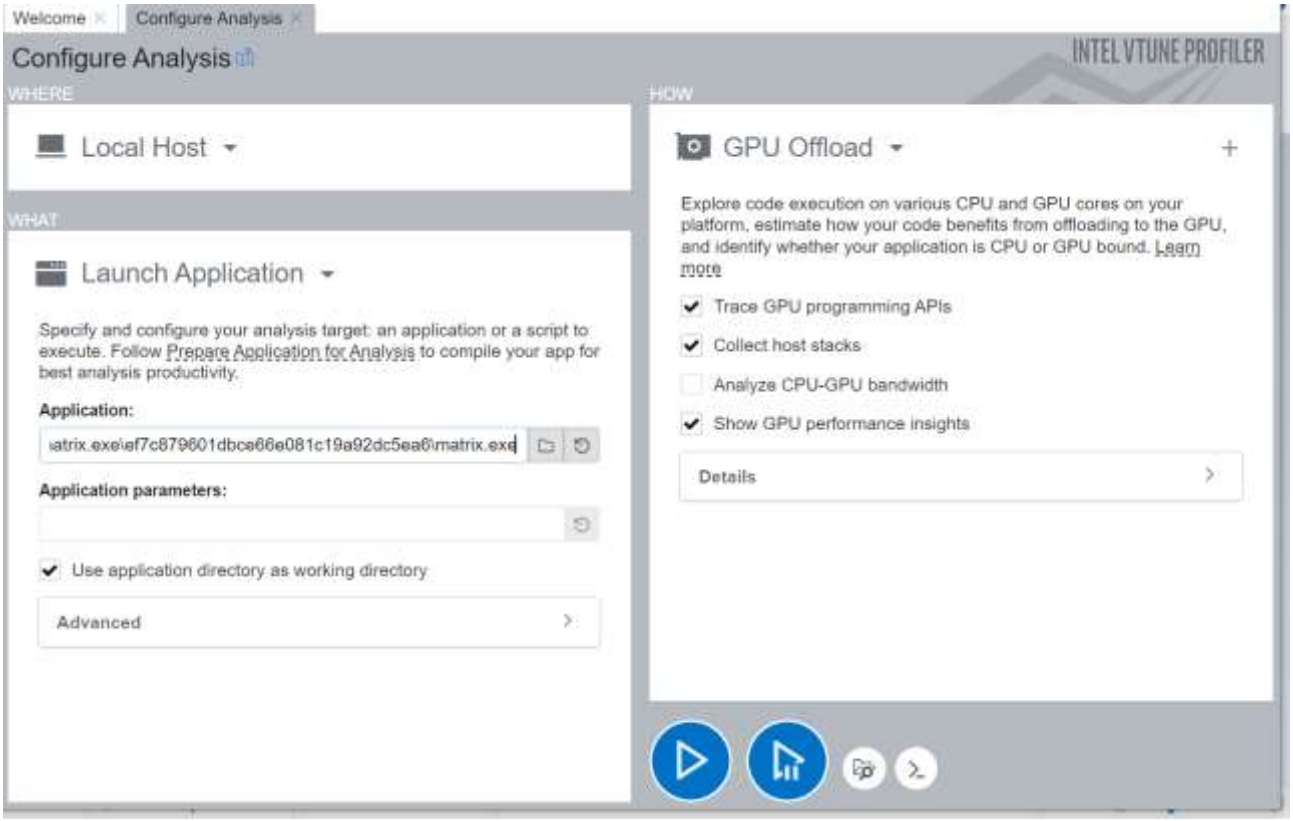
- 解析コマンドを実行します。

```
$ vtune.exe -collect gpu-offload -- matrix_multiply.exe
```

## 収集したデータを解析

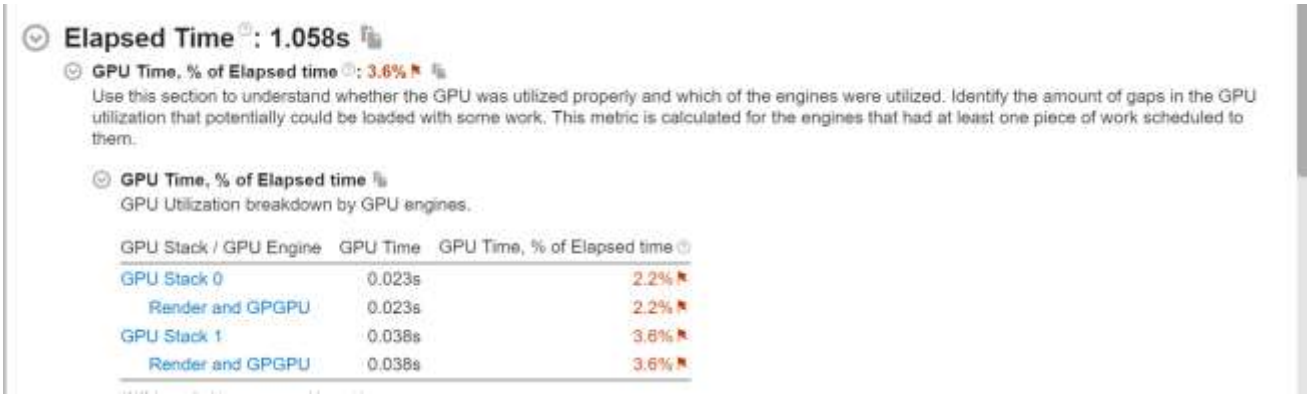
GPU オフロードの観点から解析を開始します。[Summary (サマリー)] ウィンドウで、CPU および GPU リソースの使用統計を確認して、アプリケーションが GPU 依存であるか、CPU 依存であるか、またはシステムの計算能力を効率良く利用していないことを判断できます。この例では、アプリケーションは計算集約型であり、GPU を使用する必要があります。ただし、サマリー情報は GPU の利用率が低いことを示しています。

### GPU オフロードビューポイント




**[Platform (プラットフォーム)]** ウィンドウに切り替えます。ここでは、ソフトウェア・キューの GPU 使用状況の解析に役立つ基本 CPU および GPU メトリックを確認できます。このデータはタイムライン上の CPU 使用状況に関連しています。**[Platform (プラットフォーム)]** ウィンドウの情報は、いくつかの推測に役立ちます。

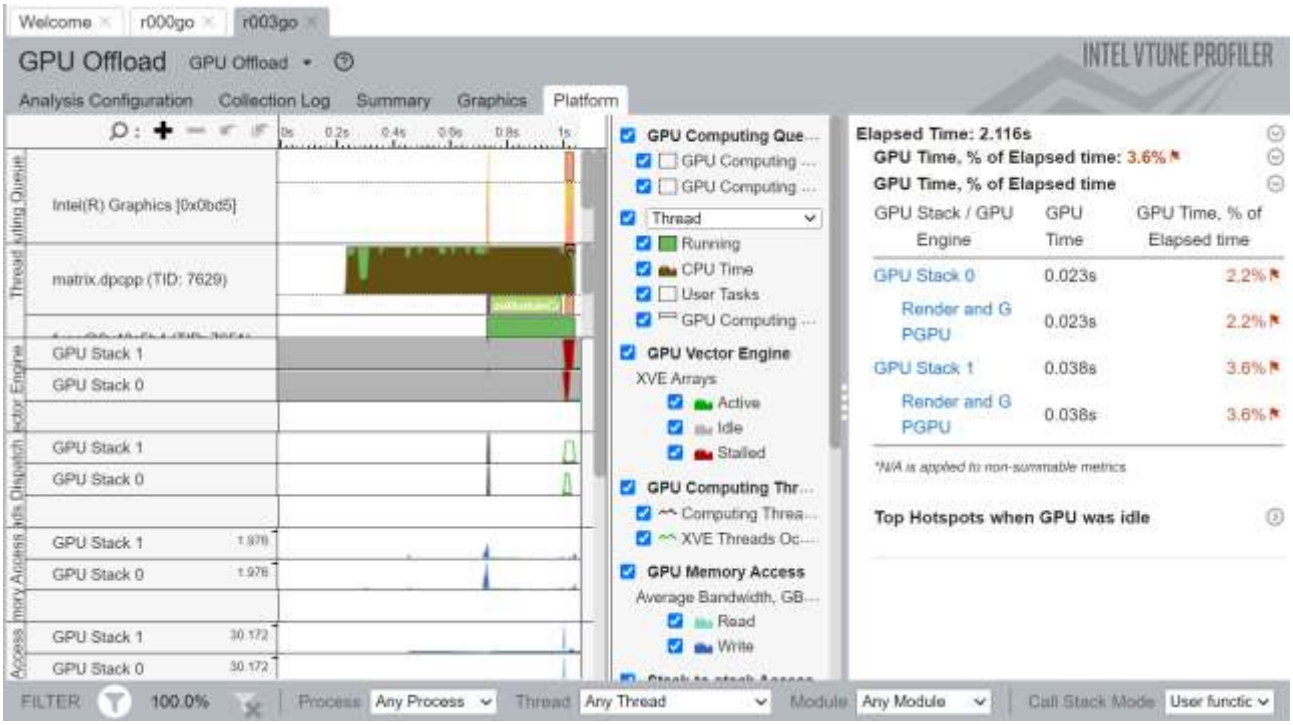
### GPU 利用率




ほとんどのアプリケーションは、上記のような状況を示さない可能性があります。すべての依存関係を理解するには、詳しい解析が必要です。例えば、ビデオ処理とレンダリングを実行する GPU エンジンは順番にロードされます。この場合、エンジンは連続して使用されます。アプリケーション・コードが CPU で実行されると、GPU では非効率なスケジュールが行われる可能性があります。これにより、アプリケーションが GPU 依存であると誤認識されることがあります。

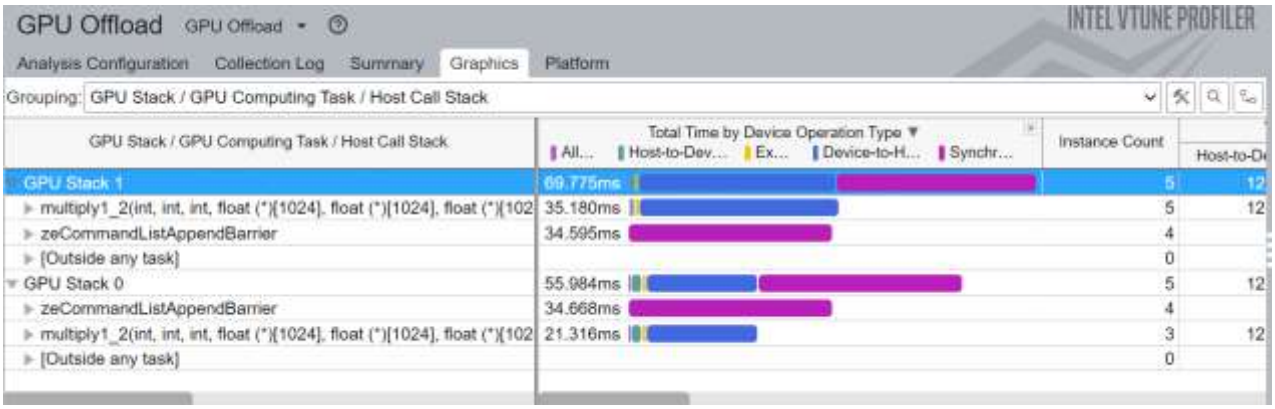
計算タスク・リファレンスと GPU 利用率メトリックから GPU 実行フェーズを特定し、タスクを生成しキューに配置するオーバーヘッドを調査します。

### GPU オフロード・プラットフォーム・ウィンドウ




計算タスクを調査するには、**[Graphics (グラフィックス)]** ウィンドウに切り替えて、スレッドごとに GPU で実行されるワークのタイプ (レンダリングまたは計算) を調べます。**[Computing Task (計算タスク)]** のグループ化を選択し、タスクのパフォーマンス特性を調査するためリストを参照します。計算タスクをさらに詳しく解析するには、**[GPU Compute/Media Hotspots analysis (GPU 計算/メディア・ホットスポット解析)]** タイプを実行します。

### 計算タスクのグループ化

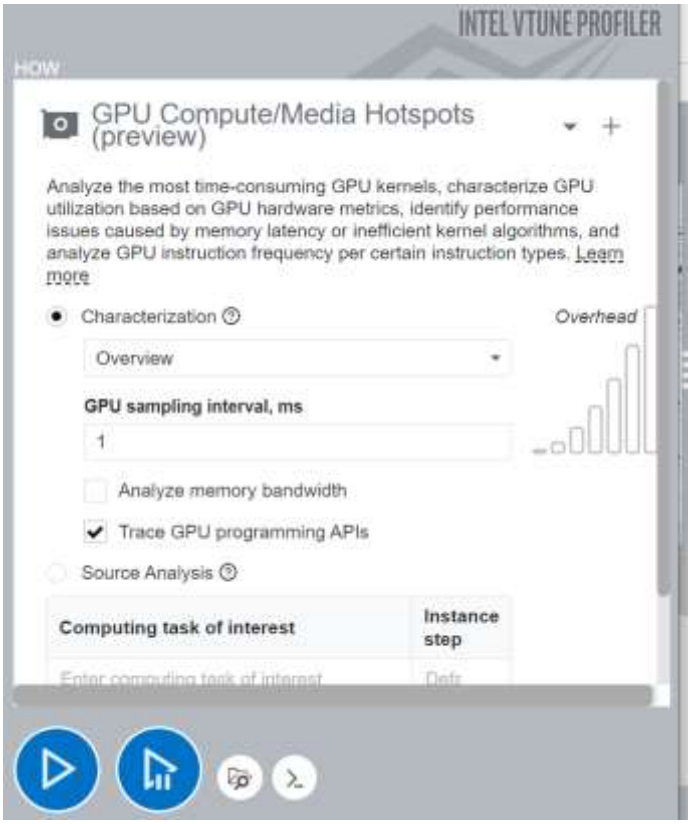



## GPU 計算/メディア・ホットスポット解析を実行

解析を実行するには、次の操作を行います。

1. **[Accelerators (アクセラレーター)]** グループで、**[GPU Compute/Media Hotspots analysis (GPU 計算/メディア・ホットスポット解析)]** タイプを選択します。
2. 前のセクションで説明したように、解析オプションを設定します。
3. **[Start (開始)]** ボタンをクリックして解析を実行します。

### GPU 計算/メディア・ホットスポット解析



コマンドラインから解析を実行します。

Linux*:

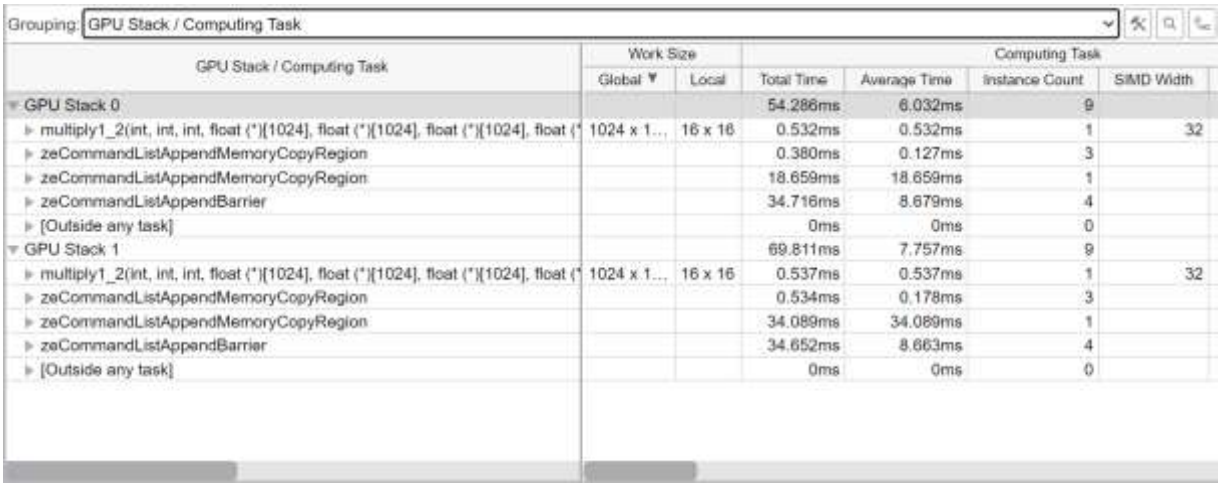
```
$ vtune -collect gpu-hotspots -- ./matrix.dpcpp
```

Windows*:

```
$ vtune.exe -collect gpu-hotspots -- matrix_multiply.exe
```

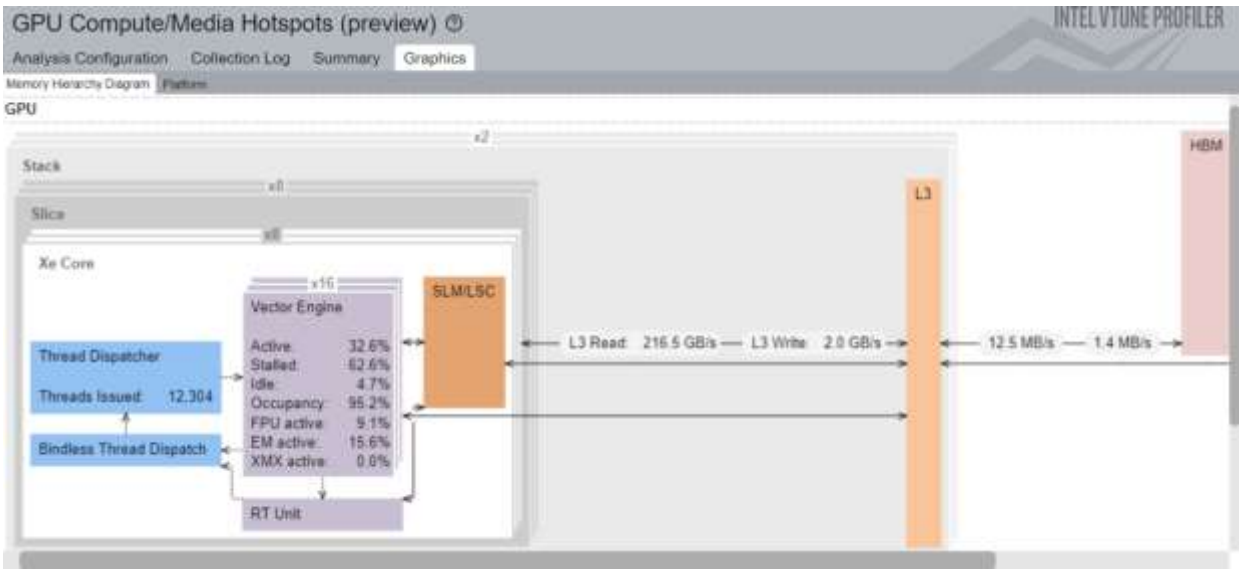
## 計算タスクを解析

### 特性化プロファイル



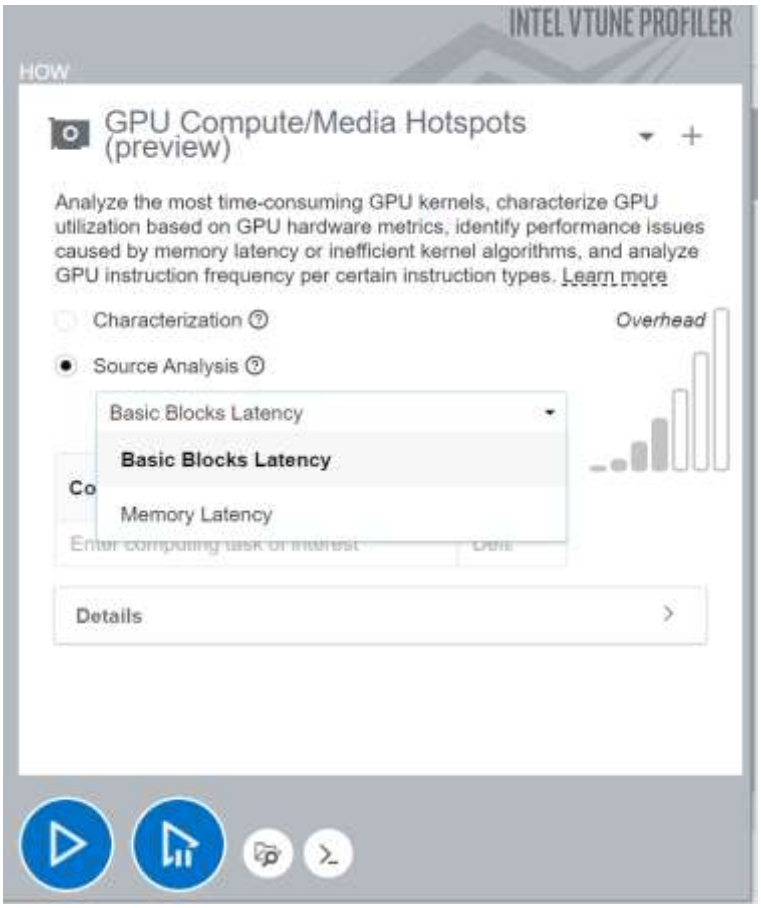

デフォルトの解析設定は、概要メトリックのセットを使用して特性プロファイルを起動します。GPU オフロード解析により利用可能な個々の計算タスクの特性に加えて、インテル® VTune™ プロファイラーは、GPU メモリー階層のそれぞれのレベルに分類されるメモリー帯域幅メトリックを示します。

### インテル® VTune™ プロファイラーのメモリー帯域幅メトリック



ソースコード・レベルで計算タスクを解析することもできます。例えば、特定のタスクに費やされた、またはメモリー待機時間で費やされた GPU クロックサイクルをカウントするには、**[Source Analysis (ソース解析)]** オプションを使用します。

### GPU 計算/メディア・ホットスポット解析、ソース解析




行列計算のコードを SYCL* に移植しました。

```
// 基本行列乗算
void multiply1(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM],
               TYPE c[][NUM], TYPE t[][NUM]) {
  int i, j, k;

  // deviceQueue を宣言
  sycl::queue q(sycl::default_selector_v, exception_handler);
  cout << "Running on " << q.get_device().get_info<sycl::info::device::name>()
       << "\n";
  // 2 次元のレンジを宣言
  sycl::range<2> matrix_range{NUM, NUM};

  // 3 つのバッファーを宣言して初期化
  sycl::buffer<TYPE, 2> bufferA((TYPE *)a, matrix_range);
  sycl::buffer<TYPE, 2> bufferB((TYPE *)b, matrix_range);
  sycl::buffer<TYPE, 2> bufferC((TYPE *)c, matrix_range);
  // ジョブをキューへ送信
  q.submit([&](auto &h) {
     // バッファーに対応する 3 つのアクセサーを宣言します。最初の 2 つは read で残りは
     // read_write です
     sycl::accessor accessorA(bufferA, h, sycl::read_only);
     sycl::accessor accessorB(bufferB, h, sycl::read_only);
```

```
     sycl::accessor accessorC(bufferC, h);

     // matrix_range で並列に行列乗算を実行します
     // ind はこの範囲へのインデックス
     h.parallel_for(matrix_range, [=](sycl::id<2> ind) {
       int k;
       for (k = 0; k < NUM; k++) {
         // 計算を実行。ind[0] は行、ind[1] は列
         accessorC[ind[0]][ind[1]] +=
             accessorA[ind[0]][k] * accessorB[k][ind[1]];
       }
     });
   }).wait_and_throw();
} // multiply1
```

コマンドラインから GPU オフロードレポートを解析すると、アプリケーションを最適化する方法を示す詳しい推奨事項を取得できます。

```
GPU Time, % of Elapsed time: 6.8%
 | GPU utilization is low. Consider offloading more work to the GPU to
 | increase overall application performance.
 |

    Top Hotspots when GPU was idle
    Function                    Module           CPU Time
    --------------------------  ---------------  --------
    [Outside any known module]  [Unknown]          0.402s
    [Skipped stack frame(s)]    [Unknown]          0.228s
    operator new                libc++abi.so       0.136s
    memmove                     libc-dynamic.so    0.078s
    memcmp                      libc-dynamic.so    0.058s
    [Others]                    N/A                0.918s

Hottest Host Tasks
Host Task                             Task Time  % of Elapsed Time(%)  Task Count
-----------------------------------  ---------  --------------------  ----------
zeModuleCreate                          0.233s                 22.4%           1
zeEventHostSynchronize                  0.068s                  6.5%           6
zeCommandListCreateImmediate            0.003s                  0.3%           1
zeCommandListAppendMemoryCopyRegion     0.002s                  0.2%           4
zeMemAllocDevice                        0.002s                  0.2%           3
[Others]                                0.003s                  0.3%          14

Hottest GPU Computing Tasks
GPU Adapter                                                    Computing Task
Total Time  Execution Time  % of Total Time(%)  SIMD Width     XVE Threads
Occupancy(%)  SIMD Utilization(%)
------------------------------------------------------------  ----------------------
-------------------------------------------------------------------------------     ------
-------------------------------------------------------------------------------------------
------------  ----------  --------------  ------------------  -------     ---  -------
-----------------  -------------------
0:41:0.0 : Display controller: Intel Corporation Device 0x0bdb  multiply1_2(int, int,
int, float (*)[1024], float (*)[1024], float (*)[1024], float (*)[1024])::
{lambda()#1}::operator()<sycl::_V1::handler>(, signed char)
const::{lambda(sycl::_V1::nd_item<(int)2>)#1}      0.071s          0.003s
4.5%             32                      97.0%              100.0%
0:41:0.0 : Display controller: Intel Corporation Device 0x0bdb
zeCommandListAppendBarrier
0.065s              0s                0.0%
Collection and Platform Info
    Application Command Line: ./matrix.dpcpp
    Operating System: 5.15.47+prerelease6379.25 DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=20.04 DISTRIB_CODENAME=focal DISTRIB_DESCRIPTION="Ubuntu 20.04.2 LTS"
    Computer Name: ortce-pvc1
    Result Size: 49.2 MB
```

```
    Collection start time: 04:55:43 16/03/2023 UTC
    Collection stop time: 04:55:44 16/03/2023 UTC
    Collector Type: Event-based sampling driver,Driverless Perf system-wide
sampling,User-mode sampling and tracing
    CPU
        Name: Intel(R) Xeon(R) Processor code named Sapphirerapids
        Frequency: 2.000 GHz
        Logical CPU Count: 224
        LLC size: 110.1 MB
    GPU
        0:41:0.0: Display controller: Intel Corporation Device 0x0bdb:
            XVE Count: 448
            Max XVE Thread Count: 8
            Max Core Frequency: 1.550 GHz
        0:58:0.0: Display controller: Intel Corporation Device 0x0bdb:
            XVE Count: 448
            Max XVE Thread Count: 8
            Max Core Frequency: 1.550 GHz

Recommendations:
    GPU Time, % of Elapsed time: 6.8%
     | GPU utilization is low. Switch to the for in-depth analysis of host
     | activity. Poor GPU utilization can prevent the application from
     | offloading effectively.
    XVE Array Stalled/Idle: 97.0% of Elapsed time with GPU busy
     | GPU metrics detect some kernel issues. Use GPU Compute/Media Hotspots
     | (preview) to understand how well your application runs on the specified
     | hardware.
    Execution % of Total Time: 2.4%
     | Execution time on the device is less than memory transfer time. Make sure
     | your offload schema is optimal. Use Intel Advisor tool to get an insight
     | into possible causes for inefficient offload.
```

また、GPU ホットスポットを使用して、GPU カーネルがどの程度効率良く実行されているか調査することもできます。エグゼキューション・ユニットがストールする頻度は、GPU のパフォーマンスを示す指標になります。もう 1 つの重要なメトリックは、L3 帯域幅依存です。

```
Elapsed Time: 1.682s
GPU Time: 0.071s
Display controller: Intel Corporation Device 0x0bdb Device Group
XVE Array Stalled/Idle: 98.5% of Elapsed time with GPU busy
 | The percentage of time when the XVEs were stalled or idle is high, which
 | has a negative impact on compute-bound applications.
 |

    This section shows the XVE metrics per stack and per adapter for all the devices
in this group.
    GPU Stack  GPU Adapter                                                      XVE
Array Active(%)  XVE Array Stalled(%)  XVE Array Idle(%)
    ---------  ----------------------------------------------------------  -------
------------  --------------------  -----------------
    0          0:58:0.0 : Display controller: Intel Corporation Device 0x0bdb
0.0%                  0.0%             100.0%
    0          0:41:0.0 : Display controller: Intel Corporation Device 0x0bdb
0.1%                  3.8%              96.0%
GPU L3 Bandwidth Bound: 0.2% of peak value
```

## マルチスタック GPU でインテル® VTune™ プロファイラーを実行

インテル® データセンター GPU マックス・シリーズは、マルチスタック・アーキテクチャーを備えています。このアーキテクチャーには、パフォーマンスに関連する新しい考慮事項があります。古いバージョンのインテル® VTune™ プロファイラー (2021.7 以前) を使用する場合、すべてのスタックから GPU ハードウェア・メトリックを収集するには、次の環境変数を設定する必要があります。

```
$ export AMPLXE_EXPERIMENTAL=gpu-multi-tile-metrics
```

マルチスタック GPU でアプリケーションを実行するには、暗黙のスケーリングと明示的なスケーリングの 2 つの方法があります。暗黙のスケーリングでは、コードを変更する必要がなく、GPU ドライバーはデバイスカーネルのスレッドグループを両方のスタックに分散します。

暗黙のスケーリングを使用するには、次の環境変数を設定します。

```
$ export EnableImplicitScaling=1
```

明示的なスケーリングを使用するにはコードを変更する必要があり、カーネルをどのスタックで実行するか指定します。明示的なスケーリングを使用すると、追加の制御が可能になるため、パフォーマンスが向上することがあります。明示的なスケーリング・モードでは、GPU ハードウェアのメトリックと計算カーネルのスタックごとの属性を含む、完全なプロファイルのサポートが得られます。

## アプリケーションのスケーリング

アプリケーションが 2 つのスタックでスケーリングしているかテストします。まず、環境変数 `ZE_AFFINITY_MASK` を使用して単一スタックで実行します。

```
$ export ZE_AFFINITY_MASK=0.0
$ export ZE_AFFINITY_MASK=0.1
```

次に、暗黙のスケーリングを使用して、単一スタックと 2 つのスタックでアプリケーションがどのように実行されているか比較します。

NUMA (やその他の問題) により、チューニングなしでは 2 つのスタックに拡張できない可能性があります。アーキテクチャーで考慮すべきいくつかの点があります。

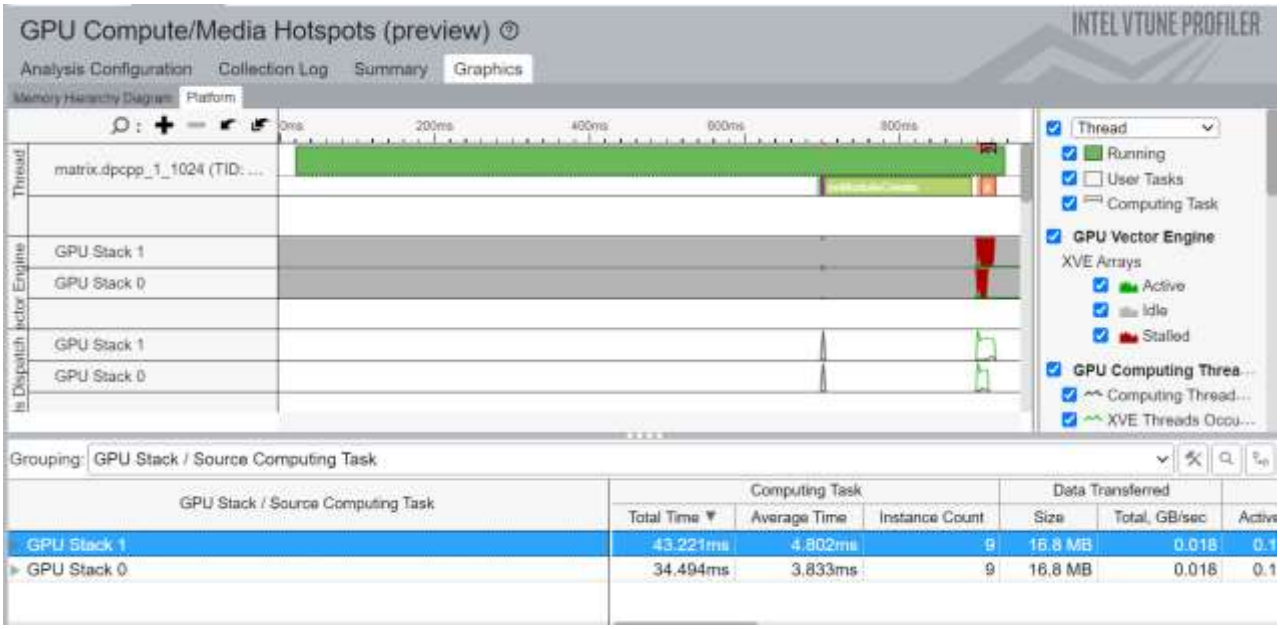
- インテル® データセンター GPU マックス・シリーズの各スタックには、個別の L3 キャッシュがあります。
- これらのキャッシュはコヒーレントではありません。
- USM 割り当ては GPU ドライバーによって管理されます。
- デバイスは、別のデバイスに割り当てられたメモリーにアクセスできません。

カーネルのパフォーマンスを理解するには、次のことに注意してください。

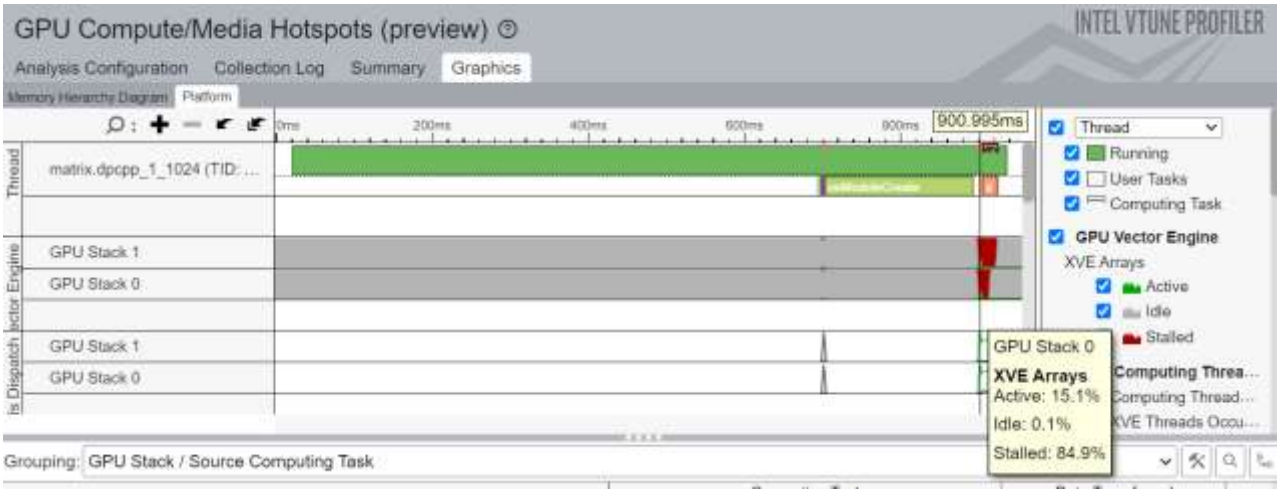
- カーネルが異なるスタックの実行リソースをどのように利用するか。
- 割り当てが異なるスタックのメモリーにどのように分散されるか。
- 割り当てが位置する場所と、割り当てがシステム内でどのように移動するか。

## マルチスタック解析

インテル® VTune™ プロファイラーの GUI では、以下のように、すべての GPU エグゼキューション・ユニットをスタックごとに表示できます。




また、スタック上のアクティビティーを関連付けて、アクティビティーのアイドル状態、またはストール状態を確認できます。




インテル® VTune™ プロファイラーは、カーネルのタイムライン全体で各スタックの読み取り/書き込みメモリー帯域幅も表示できます。

また、L3 キャッシュとキャッシュ帯域幅の状況を確認できます。

GPU 周波数は、パフォーマンスに影響する要素の 1 つです。特にスタックが異なる周波数で動作する場合、周波数の変動に注意する必要があります。これには、追加の調査が必要になります。

## 占有率解析

**占有率** - スレッドスロットにスレッドがスケジュールされている場合のすべてのサイクルの合計。

カーネルのパラメーターに基づいてピーク占有率を推測できます。

制限要因:

- グローバルサイズとローカルサイズ
- 要求される SLM サイズ
- バリアの利用 (進行中)
- 小さな/大きなカーネルのスケジュールの問題 (解析中)

## マルチ GPU

インテル® VTune™ プロファイラーは、複数のスタックを解析できるだけでなく、複数の GPU も解析できます。マルチ GPU とは、システムに複数の GPU カードが接続されている状態です。古いバージョンのインテル® VTune™ プロファイラー (2021.7 以前) を使用する場合、マルチ GPU を解析するには、次の環境変数を設定する必要があります。

```
$ export AMPLXE_EXPERIMENTAL=gpu-multi-adapter-metrics,gpu-multi-tile-metrics
```

特定の GPU で収集を行うには、各 GPU デバイスを識別して、Linux* 上で `lspci` コマンドを実行し、そのアダプターのバス/デバイス/機能 (BDF) を検出する必要があります。

収集で GPU を指定するには、**10 進**形式で `-knob target-gpu=BDF` を設定します。BFD は [B:D:F] フォーマットです。

例えば、次のコマンドでは、インテル® VTune™ プロファイラーでプロファイルできるよう、BDF=**'0000:4d:00.0'** を指定しています。

```
$ vtune -c gpu-hotspots -knob target-gpu=0:77:0.0 -- <your application>
```

GUI で **[Target GPU (ターゲット GPU)]** プルダウンメニューを使用して、プロファイルするデバイスを指定します。インテル® VTune™ プロファイラーがシステムで複数の GPU を検出した場合にのみ、**[Target GPU (ターゲット GPU)]** プルダウンメニューが表示されます。メニューには利用可能な GPU 名とアダプターの BDF が示されます。ユーザーが GPU を選択していない場合、インテル® VTune™ プロファイラーはデフォルトで最も新しいデバイスファミリーを選択します。

インテル® VTune™ プロファイラーを使用して GPU のパフォーマンスを最適化するその他の方法については、『インテル® VTune™ プロファイラー・パフォーマンス解析クックブック』の「インテル® GPU 向けのソフトウェア最適化」および「インテル® VTune™ プロファイラーを使用してインテル® GPU 向けにアプリケーションを最適化」を参照してください。

## ハードウェア・アシストのストール・サンプリング

ハードウェア・アシストのストール・サンプリングは、インテル® データセンター GPU マックス・シリーズに実装される新しいパフォーマンス監視機能です。

インテル® VTune™ プロファイラーでは、GPU オフロード解析を使用してオフロードの効率を調査でき、GPU 計算/メディア・ホットスポット解析では、GPU の実行効率をカバーします。GPU 実行効率の特性評価は、前の節のインテル® VTune™ プロファイラーの GPU 計算/メディア・ホットスポット解析の特徴付けモードで部分的に説明されています。**[Dynamic Instruction Count (動的命令カウント)]** オプションを除いて、特徴付けモードはハードウェア・カウンター・ベースであり、その粒度は計算カーネルです。カーネルレベル (アクティブ/ストール/アイドルメトリックなど) の実行効率を推測することはできますが、特徴付けモードでは、ソース/アセンブリーのレベルでパフォーマンスに関する情報が提供されます。その結果、中規模もしくは大規模なカーネルのパフォーマンス最適化は、カーネルやアプリケーション・レベルで見られる非効率な動作に対処するため、ソースレベルで修正を繰り返す必要があり、仮定と推測に基づいた時間を要する作業になることがあります。GPU 計算/メディア・ホットスポット解析のソース解析モードは、時間のかかるカーネルを詳しく調査し、基本ブロックとメモリーのレイテンシー解析やハードウェア・アシストのストール・サンプリングなど、時間を要するソース行や命令を特定するため、特徴付けモードの次のステップとして推奨されます。

HW アシストのストール・サンプリングは、インテル® データセンター GPU マックス・シリーズに実装された新しいパフォーマンス監視機能であり、Xe ベクトル (XVE) ストールイベントを実行された命令に統計的に関連付け、ストールの原因ごとにストールイベントを分類します。このサンプリング・メカニズムでは、ストールがチェックされた固定構成が可能なサイクル数に基づいて XVE がサンプリングされます。XVE は、少なくとも 1 つのスレッドがロードされているか、サンプリングされたサイクルで実行できるスレッドがない場合に限って、ストールしていると見なされます。XVE ストールが発生すると、独自のヒューリスティックに基づいてハードウェア・スレッドからスレッドを選択し、そのスレッドの命令ポインターをストールの原因とともに記録します。XVE は、命令フェッチ、バリアの送信などが原因でストールすることがあります。インテル® データセンター GPU マックス・シリーズでは、XVE ストール・サンプリングには、表に示す 8 つの理由によるストールをカウントする 8 つのカウンターが備わっています。最も粒度が細かい間隔では、HW アシストのストール・サンプリングでは、カーネル/アプリケーション実行の最大 10% のオーバーヘッドが発生する可能性があります。

### インテル® GPU 計算スループット率 (Ops/clock/EU)

| HW ストールの原因 | 説明 |
|---|---|
| ACTIVE | 少なくとも 1 つのパイプラインでアクティブに実行されている |
| INST_FETCH | 命令フェッチ操作が原因でストール |
| SYNC | 同期操作によりストール |
| SCOREBOARD ID | メモリー依存関係または内部 XVE パイプラインの依存関係によるストール |
| DIST または ACC | 内部パイプラインの依存関係によるストール |
| SEND | メモリー依存関係または内部パイプラインの送信依存関係によるストール |
| PIPESTALL | XVE パイプラインによるストール |

| HW ストールの原因 | 説明 |
|---|---|
| CONTROL | 分岐によるストール |
| OTHER | そのほかの理由によるストール |

## GUI での設定と実行

1. 環境変数を設定します (`AMPLXE_EXPERIMENTAL=gpu-stall-sampling`)。
2. インテル® VTune™ プロファイラーを起動し、**[Welcome (ようこそ)]** ページで **[New Project... (新規プロジェクト...)]** をクリックします。
3. **[Create a Project (プロジェクトを作成)]** ダイアログボックスが表示されます。
4. プロジェクトのプロジェクト名と場所を指定して、**[Create Project (プロジェクトを作成)]** ボタンをクリックします。
5. **[Configure Analysis (解析の設定)]** ウィンドウが表示されます。
6. **[WHERE (どこを)]** ペインでローカルホストが選択されていることを確認します。
7. **[WHAT (何を)]** ペインで、起動するアプリケーションのターゲットが選択されていることを確認し、プロファイルするアプリケーションとして `matrix_multiply` バイナリーを指定します。
8. **[HOW (どのように)]** ペインの **[Accelerators (アクセラレーター)]** グループから **[GPU Compute/Media Hotspots analysis (GPU 計算/メディア・ホットスポット解析)]** を選択します。
9. **[GPU Compute/Media Hotspots analysis (GPU 計算/メディア・ホットスポット解析)]** で **[Source Analysis (ソース解析)]** モードが選択されていることを確認します。
10. **[Source Analysis (ソース解析)]** モードのドロップボックスから、**[Stall Sampling (ストール・サンプリング)]** オプションを選択します。

## CLI での設定と実行

- 環境変数を設定するスクリプトを `source` します。

```
$ source <vtune_install_dir>/env/vars.sh
```

- ストール・サンプルリングはまだ実験的な機能であるため、実行するには次の環境変数を設定します。

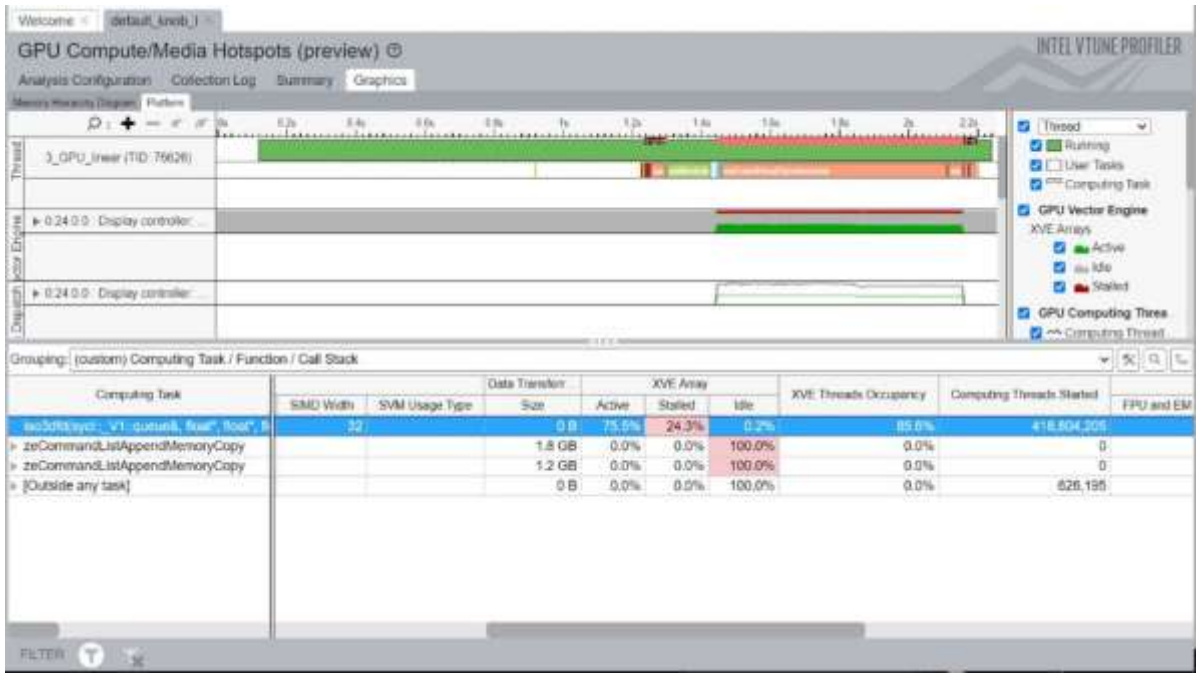
```
$ export AMPLXE_EXPERIMENTAL=gpu-stall-sampling
```

- 解析コマンドを実行します。

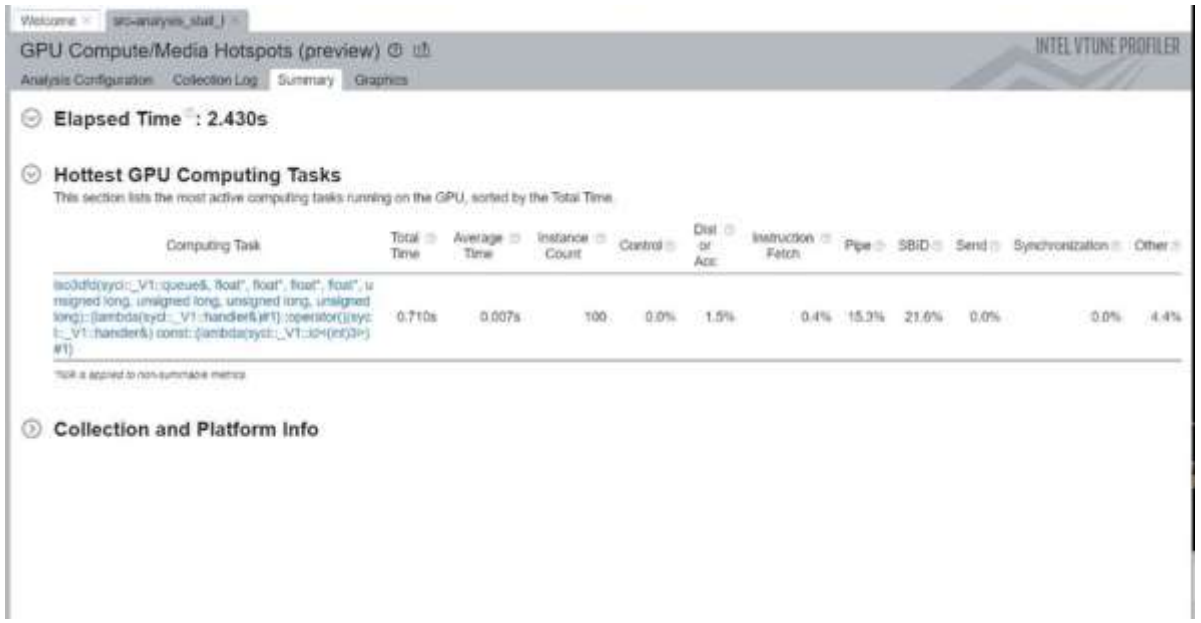
```
$ vtune -collect gpu-hotspots -knob profiling-mode=source-analysis -knob source-
analysis=stall-sampling -- <app> [parameters]
```

## データの可視化

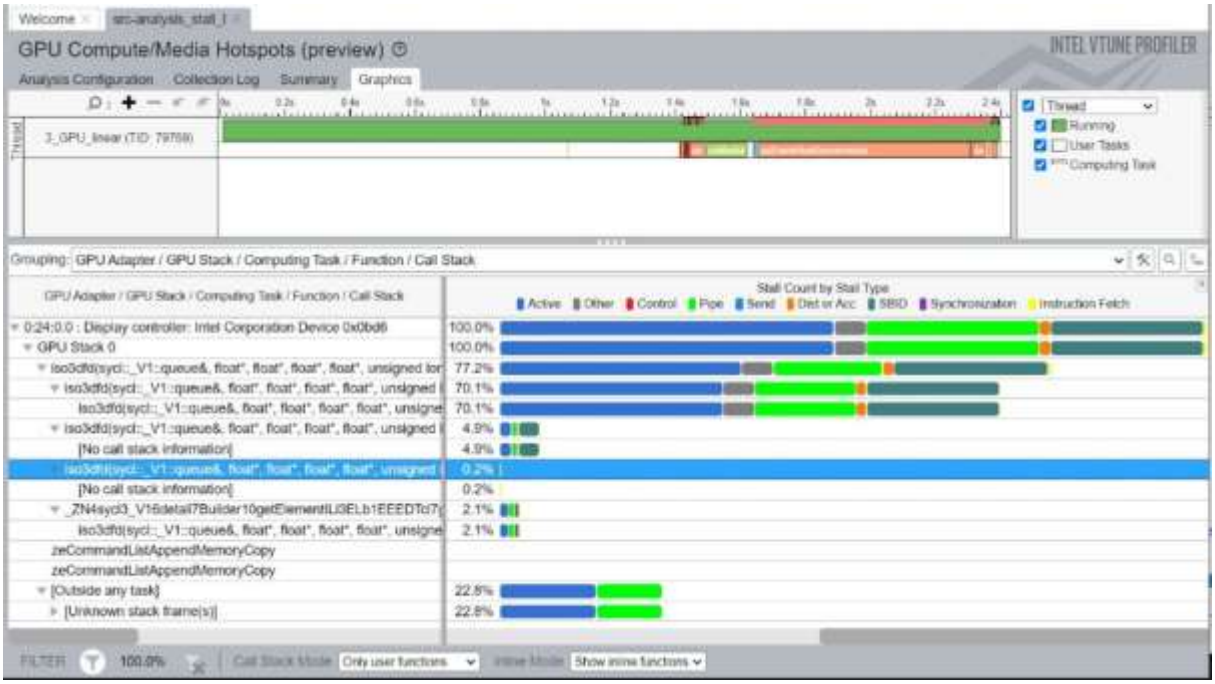
https://github.com/oneapi-src/oneAPI-samples/tree/master/DirectProgramming/C%2B%2BSYCL/StructuredGrids/guided_iso3dfd_GPUOptimization (英語) にある oneAPI サンプルの iso3dfd (線形インデックス・バージョン) のユースケースを使用して、HW アシストのストール・サンプリング解析からのデータを可視化する方法を示します。この例では、GPU 計算/メディア・ホットスポット解析の **[Characterization (特徴付け)]** モードで、アプリケーション実行中の GPU 時間の 25% ほどがストールであることが分かります。ストール・サンプリングから、アプリケーションがストールした場所と原因を特定し、コードの動作を命令レベルで把握します。



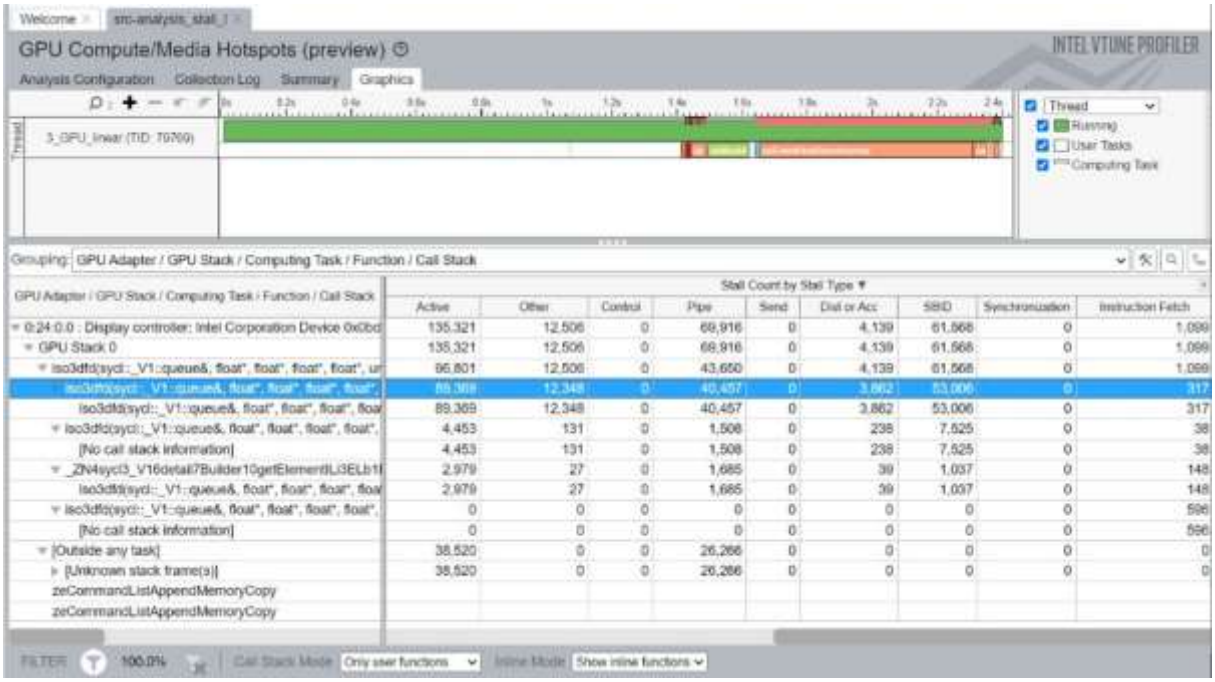

HW ストール・サンプリングで利用される主なビューは、**[Summary (サマリー)]** タブと **[Graphics (グラフィックス)]** タブです。**[Summary (サマリー)]** タブには、最も時間を要した GPU タスクが、関連するストールの理由のパーセンテージとともに表示されます。この例では、最もホットな計算タスクがストールする主な原因は、パイプストールとスコアボード ID です。



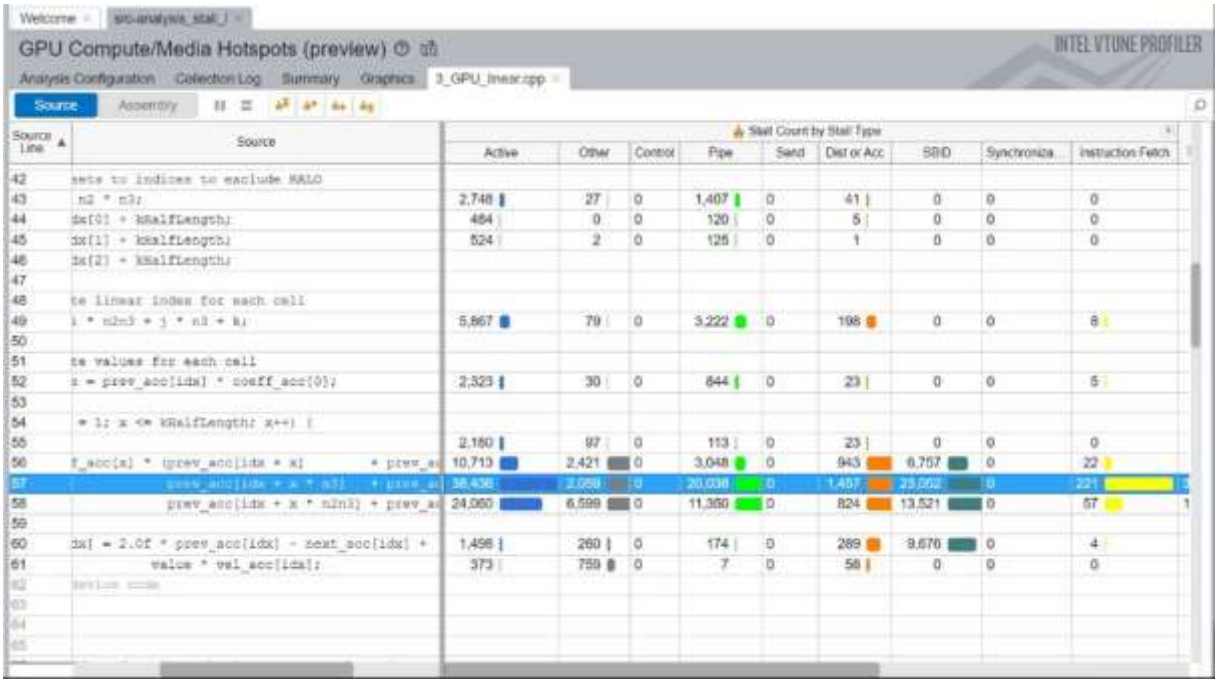

**[Graphics (グラフィックス)]** タブには、計算タスク (カーネル) を含むグリッドが表示され、すべてのインスタンスで集計された各計算カーネルのサンプル数や、サンプルの比率によってアクティブおよびストールしていることが分かります。**[GPU Adapter/GPU Stack/Compute Task/Function/Call Stack (GPU アダプター/GPU スタック/計算タスク/関数/コールスタック)]** グリッドグループはカーネル行を展開して、GPU スタックとカードレベルの関数および GPU コールスタックの統計を表示します。



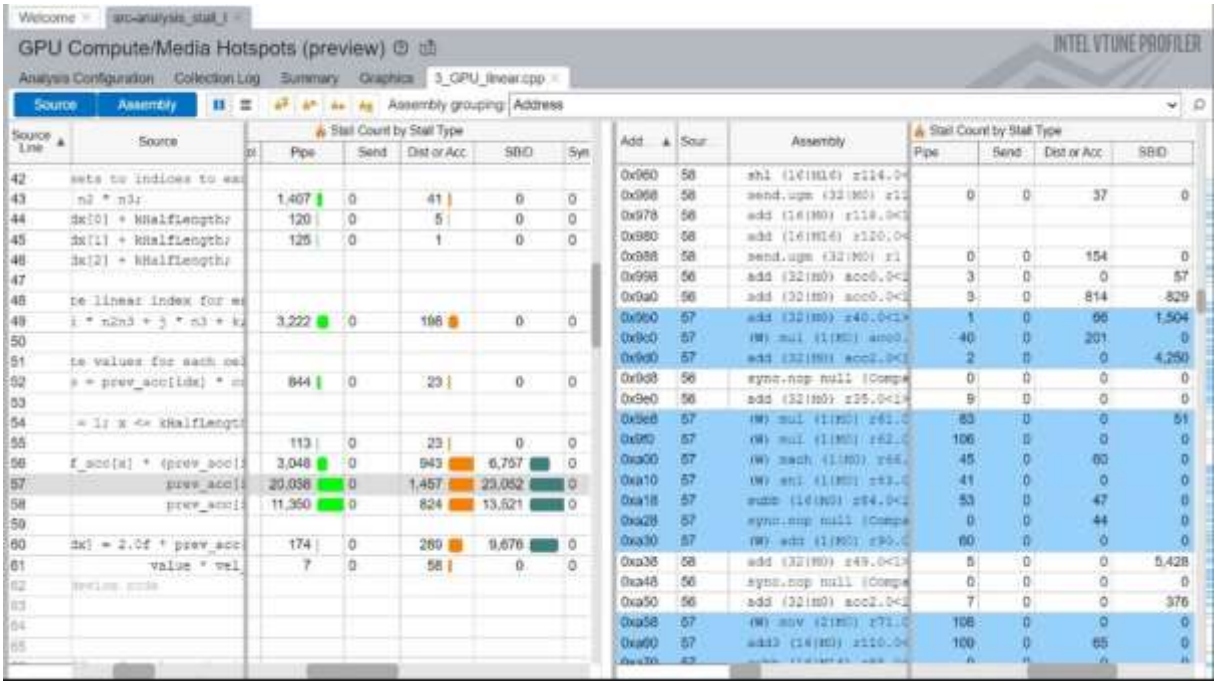

**[Stall Count by Stall Type (タイプ別のストールカウント)]** のカラムを展開して、ストールタイプ別に内訳を表示し、正確なカウントから特定の原因でソートすることもできます。



ソース/アセンブリーごとに分散されたアクティブまたはストールしたサンプルを表示するには、ターゲットの計算タスク、または関数をダブルクリックして **[Source(ソース)]** ビューに移動します。




**[Assembly (アセンブリー)]** または **[Source(ソース)]/[Assembly (アセンブリー)]** ビューを表示するには、**[Assembly (アセンブリー)]** トグルボタンを使用します。**[Source(ソース)]** ペインと **[Assembly (アセンブリー)]** ペインは、行レベルで同期されることに注意してください。



# インテル® Advisor

インテル® Advisor には、GPU で実行されるアプリケーションのパフォーマンス解析に役立つ 2 つの機能があります。

- オフロードのモデル化を使用して、CPU ベースのコード内のカーネルを識別して、GPU で実行した場合のパフォーマンスを推測します。また、使用する GPU 以外に各種 GPU の構成を調査することもできます。
- GPU ルーフラインの調査機能を使用して、GPU 性能の上限に対するアプリケーションのパフォーマンスを確認します。

**必要条件:** インテル® Advisor を使用するには、次の環境変数を設定します。

- Linux*: `source <install-dir>/advisor-vars.sh`
- Windows*: `<advisor-install-dir>\advisor-vars.bat`

この章の残りのセクションでは、上記の 2 つの機能と GPU ルーフラインの調査を使用して、メモリー依存のアプリケーションを解析および最適化する方法について詳しく説明します。

- オフロードのモデル化を使用して GPU にオフロードする領域を特定
- GPU ルーフライン解析を実行
- GPU ルーフラインでメモリー依存のアプリケーションを最適化

## オフロードのモデル化を使用して GPU にオフロードする領域を特定

インテル® Advisor の機能の 1 つであるオフロードのモデル化は、次の用途に利用できます。

- GPU にオフロードされる可能性があるコード領域を特定します。
- GPU で実行した場合のコードのパフォーマンスを推測します。
- アクセラレーターの設定パラメーターの組み合わせを試します。

オフロードのモデル化は、上限とボトルネックのパフォーマンス・モデルを使用して、速度向上の上限を示します。測定された CPU メトリックとアプリケーション特性を入力として受け取り、解析モデルを適用することでターゲット GPU 上の実行時間と特性を推測します。

インテル® Advisor の GUI から `advisor` コマンドライン・インターフェイスを使用するか、インテル® Advisor に付属する専用の Python* スクリプトを使用して、オフロードのモデル化パースペクティブを実行できます。ここでは、スクリプトを使用してオフロードのモデル化を行う方法について説明します。パースペクティブを実行するほかの方法については、『インテル® Advisor ユーザーズガイド』(英語) を参照してください。

GPU-to-GPU パフォーマンス・モデル化を実行して、SYCL*、OpenMP*、および OpenCL* アプリケーションを解析することを推奨します。正確な見積もりが得られます。GPU-to-GPU モデル化は、GPU 計算カーネルのみを解析し、CPU で実行されるアプリケーション領域を無視します。以下は、インテル® HD グラフィックス 630 GPU で動作する C++ Matrix Multiply ターゲット・アプリケーションのモデル化パフォーマンスを、異なる GPU であるインテル® データセンター GPU マックス・シリーズで比較した例です。

Linux* で C++ 行列乗算アプリケーションのオフロードモデル化を実行するには、以下の操作を行います。

1. `collect.py` スクリプトでアプリケーションのパフォーマンス・メトリックを収集します。

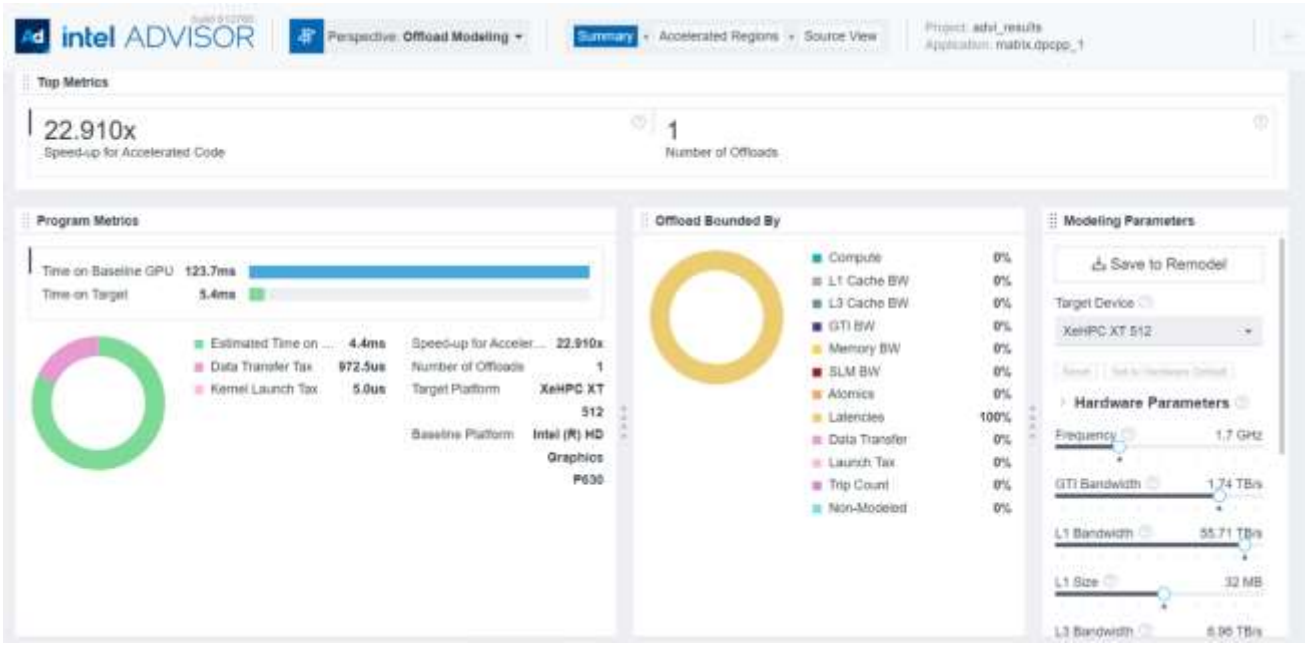
```
$ advisor-python $APM/collect.py ./advi_results --gpu --config=pvc_xt_512xve --
 ./matrix.dpcpp
```

2. `analyze.py` で GPU 上のアプリケーションのパフォーマンスをモデル化します。

```
$ advisor-python $APM/analyze.py ./advi_results --gpu --config=pvc_xt_512xve
```

パフォーマンスのモデル化を実行し、インテル® Advisor GUI で結果を開くか、`advisor_project/e000/pp000/data.0` に生成された CSV メトリックレポートとインタラクティブな HTML レポートを開きます。

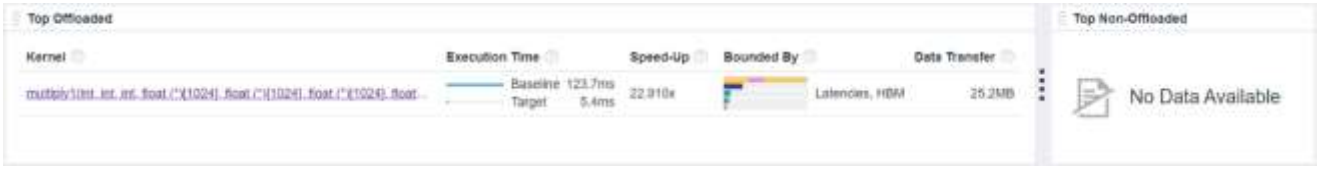
### インテル® Advisor GUI のオフロードのモデル化




例えば、レポートの **[Summary (サマリー)]** セクションを次の点に注意して閲覧します。

- ベースラインの元の実行時間、GPU アクセラレーターでの予測実行時間、オフロード領域の数、および **[Program metrics (プログラムメトリック)]** ペインの推測されるスピードアップ。`Matrix Multiply` では、インテル® Advisor は 22.9 倍の潜在的なスピードアップをレポートしています。
- オフロードを制限する要因。このペインは、アプリケーションがターゲットデバイスの帯域幅でより良いパフォーマンスを達成するのを妨げる主な制限要因をレポートします。
- GPU へのオフロードから恩恵を得られる **[Top Offloaded (上位のオフロード領域)]** のソースコード行と、それぞれのコード領域で推測されるパフォーマンスを示します。Matrix Multiply では、オフロードに推奨されるコード領域が 1 つ示されています。
- オフロードに適さない **[Top Non-Offloaded (上位の非オフロード領域)]** のソース行とその理由を示します。

### 上位の領域




**[Offloaded Regions (オフロード領域)]** タブに移動して、オフロードに推奨されるコード領域の詳細な推測および推測メトリックを表示します。また、コード領域で転送されたデータの推定量とそれに対するコストを報告します。

```
// 基本行列乗算
void multiply1(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM],
```

```
                TYPE c[][NUM], TYPE t[][NUM]) {
  int i, j, k;

  // deviceQueue を宣言
  sycl::queue q(sycl::default_selector_v, exception_handler);
  cout << "Running on " << q.get_device().get_info<sycl::info::device::name>()
       << "\n";
  // 2 次元のレンジを宣言
  sycl::range<2> matrix_range{NUM, NUM};

  // 3 つのバッファーを宣言して初期化
  sycl::buffer<TYPE, 2> bufferA((TYPE *)a, matrix_range);
  sycl::buffer<TYPE, 2> bufferB((TYPE *)b, matrix_range);
  sycl::buffer<TYPE, 2> bufferC((TYPE *)c, matrix_range);
  // ジョブをキューへ送信
  q.submit([&](auto &h) {
     // バッファーに対応する 3 つのアクセサーを宣言最初の 2 つは read で残りは
     // read_write です
     sycl::accessor accessorA(bufferA, h, sycl::read_only);
     sycl::accessor accessorB(bufferB, h, sycl::read_only);
     sycl::accessor accessorC(bufferC, h);

     // matrix_range で並列に行列乗算を実行します
     // ind はこの範囲へのインデックス
     h.parallel_for(matrix_range, [=](sycl::id<2> ind) {
       int k;
       for (k = 0; k < NUM; k++) {
           // 計算を実行。ind[0] は行、ind[1] は列
         accessorC[ind[0]][ind[1]] +=
             accessorA[ind[0]][k] * accessorB[k][ind[1]];
       }
     });
   }).wait_and_throw();
} // multiply1
```

## GPU ルーフライン解析を実行

GPU で実行されるアプリケーションのハードウェアの上限に対するパフォーマンスを見積もるには、GPU ルーフライン解析を利用できます。インテル® Advisor は、インテル® GPU で実行されるカーネルのルーフライン・モデルを生成できます。GPU ルーフライン・モデルは、カーネルを特徴付けて、理想的なパフォーマンスの上限からどれくらい離れているか、視覚的なグラフを示す非常に優れた機能を提供します。GPU ルーフラインの詳細については、『インテル® Advisor ユーザーズガイド』(英語) をご覧ください。

**要件:** Linux* では root 権限で、また Windows* では管理者として GPU ルーフライン解析を実行することを推奨します。

### Linux* ユーザー

Linux* で root 権限がない場合、root 以外のユーザーが GPU メトリックを収集できるようシステムを構成します。

1. ユーザー名を video グループに追加します。以下のコマンドを実行して、ユーザー名がすでに video グループに含まれるか確認します。

   ```
   $ groups | grep video
   ```

   ユーザー名が含まれていない場合、次のコマンドでユーザー名を追加します。

   ```
   $ sudo usermod -a -G video <username>
   ```

2. `dev.i915.perf_stream_paranoid` オプションの値を 0 に設定します。

```
$ sysctl -w dev.i915.perf_stream_paranoid=0
```

3. OpenCL* カーネルを長時間実行する場合、時間制限を無効にします。

```
$ sudo sh -c "echo N> /sys/module/i915/parameters/enable_hangcheck"
```

**すべてのユーザー**

1. SYCL* コードが GPU 上で適切に実行されていることを確認します。どのハードウェアでコードが実行されているか確認するには、SYCL* コードに次の行を追加してください。

```
sycl::default_selector selector;
sycl::queue queue(delector);
auto d = queue.get_device();
std::cout<<Running on :<<d.get_info<cl::sycl::info::device::name>()<<std::endl;
```

2. Linux* でインテル® Advisor の環境を設定します。

```
$ source <advisor_install_dir>/env/vars.sh
```

インテル® Advisor CLI で GPU ルーフライン解析を行うには、次の操作を行います。

1. `profile-gpu` オプションでサーベイ解析を実行します。

```
$ advisor -collect=survey --profile-gpu --project-dir=./advisor-project --search-dir
src:r=./ matrix_multiply -- matrix_multiply
```

2. `profile-gpu` オプションでトリップカウント & FLOP 解析を実行します。

```
$ advisor --collect=tripcounts --stacks --flop --profile-gpu --project-dir=./advisor-
project --search-dir src:r=./matrix_multiply -- matrix_multiply
```

3. インテル® Advisor GUI で生成された GPU ルーフライン・レポートを開きます。SYCL* Matrix Multiply アプリケーションの次のメトリックを確認します。

   - **[Summary (サマリー)]** タブで、**[Top Hotspots (上位のホットスポット)]** ペインに上位ホットスポットとメモリーレイアウトを表示します。

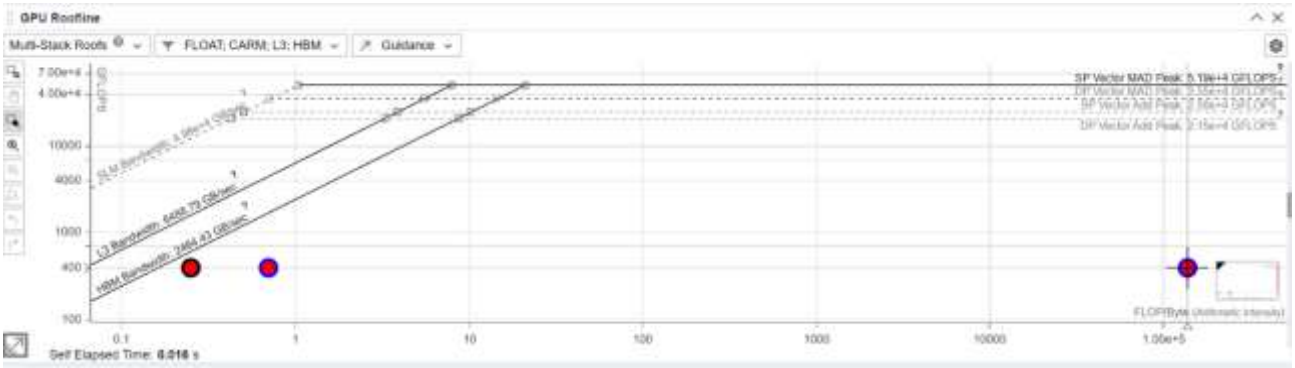
**上位のホットスポット・ペイン**

**[GPU Kernels (GPU カーネル)]** ペインで、アプリケーションが各 GPU カーネルのエグゼキューション・ユニットをどの程度効率良く使用しているか確認します。**[Top Hotspot (上位のホットスポット)]** で上位ホットスポットをクリックすると、**[GPU Kernels (GPU カーネル)]** ペインに XVE の使用状況が表示されます。

### GPU カーネルペイン

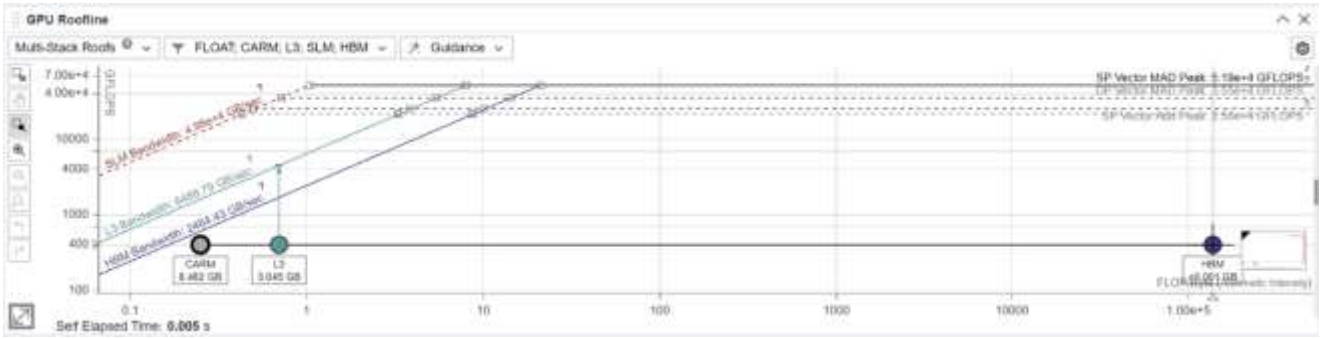
**[GPU Roofline (GPU ルーフライン)]** タブで、GPU ルーフライン・グラフとパフォーマンス・メトリックを参照します。

### GPU ルーフライン・グラフとパフォーマンス・メトリック




- `MatrixMultiply` アプリケーションは、396.97GFLOPS のパフォーマンスを達成しています。グローバルメモリーを使用しており、アプリケーションがグローバルアクセサーを使用するため、ローカル (SLM) メモリー向けには最適化されていません。
- このルーフライン・ビューでは、1 つのループに対し 3 つの点が表示されています。各点は、同じパフォーマンス (垂直位置) と異なる演算強度 (水平位置) を示します。
- 緑色の点で示されるように、アプリケーションは L3 帯域幅によって制限されています。
- 左の灰色の点は CARM を示し、レジスターと L1 間のメモリー転送を考慮しています。
- 右の紫の点は HBM を示し、L3 と HBM 間のメモリー転送を考慮しています。

### GPU ルーフライン・グラフとパフォーマンス・メトリック



GPU ルーフライン・グラフが示しているように、メモリー帯域幅をより有効に利用することを可能にする次の最適化手法があります。

- ローカルメモリーを使用 (SLM)
- SLM を効率良く利用するキャッシュのブロック化を使用

次に最適化されたバージョンの `MatrixMultiply` アプリケーションを示します。このバージョンでは、2 つのスタックを宣言して、それらを `sycl::access::target:local` として定義します。いくつかの内部ループでこれらのスタックを処理するようにカーネルを変更します。

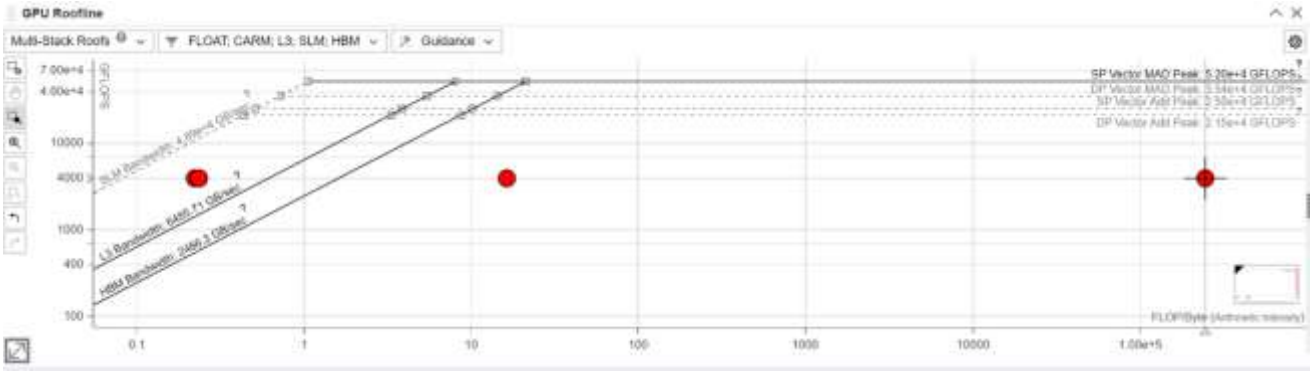
```
1.  // accessorC 参照をローカル変数に置き換えます
2.  void multiply1_1(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM],
3.                   TYPE c[][NUM], TYPE t[][NUM]) {
4.
5.    int i, j, k;
6.
7.    // deviceQueue を宣言
8.    sycl::queue q(sycl::default_selector_v, exception_handler);
9.    cout << "Running on " << q.get_device().get_info<sycl::info::device::name>()
10.        << "\n";
11.
12.   // 2 次元のレンジを宣言
13.   sycl::range<2> matrix_range{NUM, NUM};
14.
15.   // 3 つのバッファーを宣言して初期化
16.   sycl::buffer<TYPE, 2> bufferA((TYPE *)a, matrix_range);
17.   sycl::buffer<TYPE, 2> bufferB((TYPE *)b, matrix_range);
18.   sycl::buffer<TYPE, 2> bufferC((TYPE *)c, matrix_range);
19.
20.   // ジョブをキューへ送信
21.   q.submit([&](auto &h) {
22.      // バッファーに対応する 3 つのアクセサーを宣言最初の 2 つは read で残りは
23.      // read_write です
24.      sycl::accessor accessorA(bufferA, h, sycl::read_only);
25.      sycl::accessor accessorB(bufferB, h, sycl::read_only);
26.      sycl::accessor accessorC(bufferC, h);
27.
28.      // matrix_range で並列に行列乗算を実行します
29.      // ind はこの範囲へのインデックス
30.      h.parallel_for(matrix_range, [=](sycl::id<2> ind) {
31.        int k;
32.        TYPE acc = 0.0;
33.        for (k = 0; k < NUM; k++) {
34.          // 計算を実行。ind[0] は行、ind[1] は列
35.          acc += accessorA[ind[0]][k] * accessorB[k][ind[1]];
36.        }
37.        accessorC[ind[0]][ind[1]] = acc;
38.      });
39.    }).wait_and_throw();
40. }
41.
42. // accessorC 参照をローカル変数に置き換えて行列をタイル化します
43. void multiply1_2(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM],
44.                   TYPE c[][NUM], TYPE t[][NUM]) {
45.   int i, j, k;
46.
47.   // deviceQueue を宣言
48.   sycl::queue q(sycl::default_selector_v, exception_handler);
49.   cout << "Running on " << q.get_device().get_info<sycl::info::device::name>()
```

```
50.         << "\n";
51.
52.   // 2 次元のレンジを宣言
53.   sycl::range<2> matrix_range{NUM, NUM};
54.   sycl::range<2> tile_range{MATRIXTILESIZE, MATRIXTILESIZE};
55.
56.   // 3 つのバッファーを宣言して初期化
57.   sycl::buffer<TYPE, 2> bufferA((TYPE *)a, matrix_range);
58.   sycl::buffer<TYPE, 2> bufferB((TYPE *)b, matrix_range);
59.   sycl::buffer<TYPE, 2> bufferC((TYPE *)c, matrix_range);
60.
61.   // ジョブをキューへ送信
62.   q.submit([&](auto &h) {
63.      // バッファーに対応する 3 つのアクセサーを宣言最初の 2 つは read で残りは
64.      // read_write です
65.      sycl::accessor accessorA(bufferA, h, sycl::read_only);
66.      sycl::accessor accessorB(bufferB, h, sycl::read_only);
67.      sycl::accessor accessorC(bufferC, h);
68.
69.      // 行列タイルを作成
70.      sycl::local_accessor<TYPE, 2> aTile(
71.          sycl::range<2>(MATRIXTILESIZE, MATRIXTILESIZE), h);
72.      sycl::local_accessor<TYPE, 2> bTile(
73.          sycl::range<2>(MATRIXTILESIZE, MATRIXTILESIZE), h);
74.      // matrix_range で並列に行列乗算を実行します
75.      // ind はこの範囲へのインデックス
76.      h.parallel_for(sycl::nd_range<2>(matrix_range, tile_range),
77.                     [=](cl::sycl::nd_item<2> it) {
78.                       int k;
79.                       const int numTiles = NUM / MATRIXTILESIZE;
80.                       const int row = it.get_local_id(0);
81.                       const int col = it.get_local_id(1);
82.                       const int globalRow =
83.                           MATRIXTILESIZE * it.get_group(0) + row;
84.                       const int globalCol =
85.                           MATRIXTILESIZE * it.get_group(1) + col;
86.                       TYPE acc = 0.0;
87.                       for (int t = 0; t < numTiles; t++) {
88.                         const int tiledRow = MATRIXTILESIZE * t + row;
89.                         const int tiledCol = MATRIXTILESIZE * t + col;
90.                         aTile[row][col] = accessorA[globalRow][tiledCol];
91.                         bTile[row][col] = accessorB[tiledRow][globalCol];
92.                         it.barrier(sycl::access::fence_space::local_space);
93.                         for (k = 0; k < MATRIXTILESIZE; k++) {
94.                           // 計算を実行。ind[0] は行、ind[1] は列
95.                           acc += aTile[row][k] * bTile[k][col];
96.                         }
97.                         it.barrier(sycl::access::fence_space::local_space);
98.                       }
99.                       accessorC[globalRow][globalCol] = acc;
100.                    });
101.   }).wait_and_throw();
102.} // multiply1_2
```

最適化されたバージョンを `multiply_1_2` として保存し、GPU ルーフラインを再度実行し、新しいグラフを表示します。

- 最適化されたアプリケーションは、3994.98 GFLOPS を達成しています。
- アプリケーションは、16 x 16 スタックを格納するグローバルメモリーと SLM メモリーを使用します。これによりメモリー帯域幅が増加します。

### GPU ルーフラインの新しいグラフ




## GPU ルーフラインでメモリー依存のアプリケーションを最適化

ここでは、インテル® Advisor の GPU ルーフラインの調査パースペクティブを使用して、GPU アプリケーションのパフォーマンスの問題を特定して、その理由を理解する方法について説明します。

SYCL* または OpenMP* プログラミング・モデルを使用して GPU アプリケーションを開発する場合、カーネルの並列実行を維持することが重要です。通常、大量で不均一なメモリーアクセスは、GPU のパフォーマンスを制限する主な問題となります。グローバルデータが配置されるメモリーレベルと GPU エグゼキューション・ユニット間のパスが複雑である場合、パス上にある異なるステージで帯域幅の制限を受け、GPU がストールしてデータを待機している可能性があります。それらのステージを特定して、パス上のデータフローを測定するのは、重要なパフォーマンス最適化手法です。GPU ルーフラインの調査パースペクティブは、カーネルごとのボトルネックを解析し、問題の原因となるデータパスのステージを迅速に特定するのに役立ちます。GPU の実行アルゴリズムを最適化する手法として、データアクセスの元を再考することが考えられます。

## インテル® GPU マイクロアーキテクチャーのメモリーパス

インテル® プロセッサー・グラフィックスの計算アーキテクチャーは、GPU の世代によって、CPU と同じ DRAM を共有する統合計算デバイスメモリーとしてのシステムメモリーと、ディスクリート GPU カードに搭載される専用の VRAM を使い分けます。

CPU と GPU で DRAM を共有する統合 GPU では、グローバルデータはシステム DRAM から LLC (ラスト・レベル・キャッシュ) を経由して GPU 上の GTI (グラフィックス・テクノロジー・インターフェイス) に移動します。データを効率良く再利用することで、データは GPU の L2 キャッシュに留まり高速に $X^e$ ベクトルエンジン (XVE) レジスターファイルにフェッチできます。

高帯域幅で低レイテンシーのシステム上でデータにアクセスする最速の方法がレジスターにアクセスすることであると仮定すると、インテル® Advisor のキャッシュを考慮したルーフライン・モデル (CARM) は、アルゴリズムが消費する真の (つまり純粋な) データ量で最も効果的にアクセスできるものとして扱います。これをアルゴリズム的データと呼ぶことにします。例えば、行列乗算アルゴリズムの純粋な実装において、理論的にそれぞれの行列を読み取って計算に使用するデータ量は次のように考えることができます。

**D * M3**

説明:

- D はバイト単位の行列要素のサイズです
- M は正方行列のサイズです

GRF (汎用レジスターファイル) が理論的にすべての行列データを格納できれば、DRAM から GPU へのデータ転送は 1 回のみで済みます。それ以外では、データは DRAM から L2 キャッシュに転送され、そしてアルゴリズムで定義されたメモリーアドレスのアクセス順に分割して転送されます。計算でデータを再利用することで、データの一部が長くキャッシュに留まりアクセスが早くなります。アルゴリズムが使用するデータは、L2 キャッシュに収まるのが理想的です。実際には、L2 キャッシュにあるデータを可能な限り再利用して、以降の部分データを再利用できるよう退避させるのが現実的です。

L2 キャッシュからデータが退避され、次の計算で必要になる場合もあります。これは、冗長なキャッシュ・トラフィックを発生させ、データパスの帯域幅の負荷が増加することになります。そのような状況を判断できるメトリックに L2 キャッシュミスがあります。

DRAM から L2 キャッシュへのデータフェッチは 64 バイトのキャッシュライン単位で行われるため、キャッシュライン・サイズより小さなデータ・オブジェクトやキャッシュラインにまたがるデータ・オブジェクトにアクセスすると、DRAM から不要なデータもフェッチしなければならないため、過度なキャッシュ・トラフィックが発生することがあります。最悪のシナリオは、グローバルメモリー空間にランダムに格納されるバイトサイズのオブジェクトにアクセスすることです。この場合、アクセスごとに 63 バイトの使用しないデータが L2 キャッシュに取り込まれ、データパスでは (アルゴリズム的なデータの代わりに) データ転送オーバーヘッドが増加します。

L2 キャッシュのほかに、work-group のローカル・メモリー・アドレス空間をサポートする L2 キャッシュ外の専用構造である共有ローカルメモリー (SLM) があります。SLM は、Gen9 では物理的には L2 の一部ですが、機能的には分離しています。SLM は GPU パフォーマンスを大幅に向上させますが、サイズに制限があるため、パフォーマンスを確実に向上させるには work-group のサイズを慎重に選択しなければなりません。

各ステージには帯域幅の制限があります。通常、VE から遠くなるほど帯域幅は低くなります (CPU のメモリー・アーキテクチャーに似ています)。アルゴリズムが実装する特定のデータ・アクセス・パターンによっては、いくつかのステージがデータフローのボトルネックになることがあります。複雑なアルゴリズムでは、データアクセスにさまざまなパターンが組み合わされるため、ボトルネックが複数存在する可能性があります。

インテル® Advisor の GPU ルーフラインの調査パースペクティブは、データ転送のさまざまなステージでボトルネックを特定し、ソースコードに対応付けることでパフォーマンスの問題と最適化に焦点を当てることができます。GPU ルーフラインの調査が提供するデータだけではなく、その他のツールを使用してボトルネックの原因となるデータを特定できます。

## GPU ルーフラインの方法論

インテル® Advisor のルーフライン解析は、2 つの視点から実装されています。「CPU/メモリー・ルーフラインの調査」では CPU アプリケーションのパフォーマンスを解析し、「GPU ルーフラインの調査」では GPU ルーフラインのパフォーマンスを解析します。CPU/メモリーのルーフラインに焦点を当てたルーフライン・モデルの一般的な方法論は、「インテル® Advisor ユーザー向けのルーフライン・リソース」で詳しく説明されています。ルーフラインの基本を理解してから最適化を進めることを強く推奨します。この記事では、インテル® GPU 上でパフォーマンスを解析する GPU ルーフラインの調査パースペクティブの機能を説明しています。

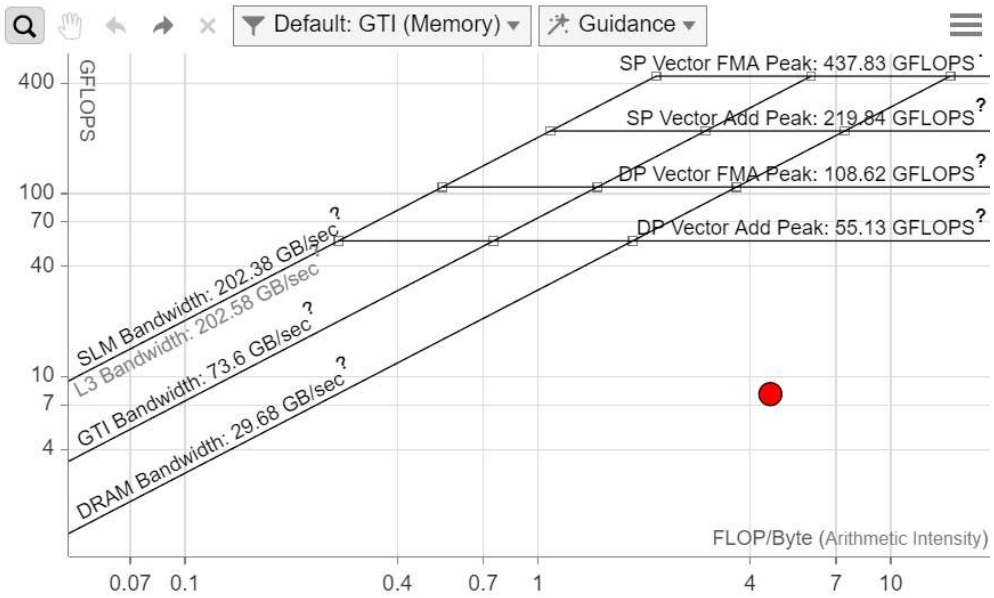
### ルーフライン解析の概要

GPU ルーフラインの調査を使用した GPU パフォーマンスの測定は非常に簡単です (「GPU ルーフラインを使用」(英語) を参照)。

1. インテル® Advisor GUI (英語) またはインテル® Advisor のコマンドライン・ツール (英語) から GPU ルーフラインの調査を行います。

2. 解析結果を開いて、GPU ルーフライン・グラフのレポートを調査します。グラフは、マシンが達成可能な最大パフォーマンスに対するアプリケーションのパフォーマンスと演算強度を表示します。

   例えば、行列乗算アプリケーションの場合、GTI (メモリー) レベルでフィルター処理された GPU ルーフライン・グラフには、GPU カーネルを示す 1 つの赤い点が描画されています。




グラフでは、各点は測定されたデータとパフォーマンス特性をもとにプロットされたカーネルを示します。演算強度 (X 軸) とパフォーマンス (Y 軸) の 2 次元座標は、解析の中心となります。この座標の点の位置は、カーネルのパフォーマンスとアルゴリズムのデータ消費量、そして GPU の最大演算性能とデータフロー帯域幅などのハードウェアによる制限との関係を示します。グラフ上で、ハードウェアの制限は対角線またはルーフとして示されます。グラフ上のカーネルの位置は、2 つの主要なことを理解するのに役立ちます。

- 現在の GPU でカーネルのパフォーマンスをスピードアップする改善の余地があるか
- カーネルが、計算、キャッシュ、メモリー帯域幅のいずれに依存するか、およびその制限を超えてパフォーマンスを向上させるためアルゴリズムの実装で変更できるものは何であるか

ここでは、メモリーやキャッシュに依存するアプリケーションの例を説明しています。

**カーネル位置の計算**

カーネルの点がグラフのどの位置にあるか、その理由を知ることは次のことから重要です。

カーネルはアルゴリズムの実装であり、一定量のデータに対して一定数の計算操作 (add、mul、mad など) を行います。例えば、行列乗算の自然な実装で、データをメモリーから直接読み取ると仮定すると、アルゴリズムの演算強度 (AI) は次のように計算できます。

**AI = M3/ 3 * M2**

説明:

- M は正方行列のサイズです
- M3 は操作数です
- 3 * M2 はリード/ライトの量です

アルゴリズムのパフォーマンス (P) は次のように計算できます。

**P = M3 / T**

説明:

- T は操作が完了するまでの時間です
- M3 は操作数です

AI と P の値により、グラフ上のカーネルの点の位置が決定されます。

理論的な計算から実際の例に切り替えて考えてみます。インテル® Advisor は、実行時にデータを計測するため、アルゴリズムが必要とする理論上の操作数やデータ量を必要としません。各カーネルは、内部インストルメント・ツールによって分離され、API トレースによって計測されます。計測パフォーマンス (P') は次のように計算できます。

**P' = I' / T'**

説明:

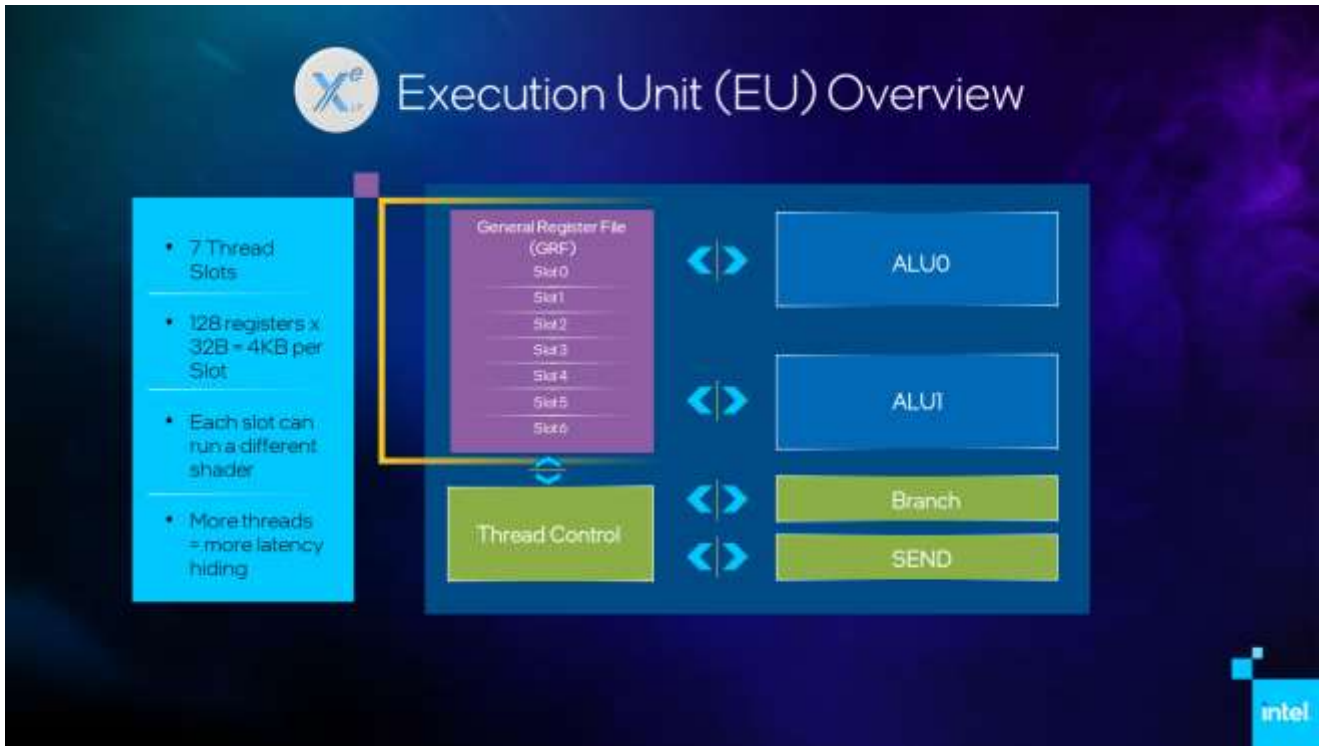
- I' は計測された計算命令の実行数です
- T' は計測時間です

計算命令には、カーネルが使用するバイト数の計算に役立つ特定のデータ参照構文があり、アルゴリズムが使用するデータは VE 数が GRF からデータを取得するステージでのみ容易に計測できます。しかし、このデータは GPU マイクロアーキテクチャー内のメモリー階層の異なる場所から取得される可能性があり、異なるステージを通過するデータ量も異なることがあります。

ルーフライン・グラフでは、カーネルの点がメモリーレベルごとの複数のドットに分割されることがあります。以降では、各メモリーレベルの詳細と、インテル® Advisor のルーフライン・グラフへのプロット方法について説明します。

どのメモリーレベルがアルゴリズムの実行を制限しているか、知る方法はあるでしょうか? アルゴリズムのパフォーマンスは、時間 T' の間に実行された命令数 I' として測定され、各メモリーステージでデータ・トラフィック $D_{XX}$ が必要です。アルゴリズムがメモリー依存であると仮定すると、一部のレベルではデータフローがハードウェアの上限帯域幅に近くなりますが、ほかのレベルでは制限が減少する可能性があります。アルゴリズムの実装における最も重要なボトルネックを特定するには、対応するメモリーレベルのルーフラインに最も近い点を見つける必要があります。データフローには複数のボトルネックが存在する可能性があり、その場合、点とそれに対応するルーフラインとの距離は同じである必要がある、ということに注意してください。

インテル® Advisor のルーフライン・グラフで点をダブルクリックして、点までの距離が最も短いルーフを見つけ、ボトルネックを特定します。このツールはメモリー依存に関する追加のヒントも提供しますが、それについては後で説明します。




## GPU ルーフラインを使用したパフォーマンス最適化のシナリオ

ルーフライン・グラフは、GPU で高速に動作させるためコードの変更が必要であることを直接示すわけではありませんが (ヒントやガイドは提供します)、アルゴリズムの実装で最も影響の高いメモリー局所性のパターン (複数可能) を示します。カーネルのドットがメモリーレベルに対してグラフ上のどの位置にあるかを調べることで、データフローに対しボトルネックになっているメモリーステージを特定できます。この情報を基に、アルゴリズムのデータパターンを修正します。例えば、データをブロック化してキャッシュを再利用し、不要なデータの多重リードを回避するなどの応用が可能です。

経験を積むほどデータのパターンを容易に理解できるようになりますが、初心者でも基本的なケースを検証することはできます。しかし、実際のアプリケーションは次のような影響を受けるため、通常、特定のルーフに単純に依存するような極端な動作を示すことはありません。

- work-group 項目の計算インデックス、メモリーアドレス計算、ループ・インデックスなど、メモリーと VE 間で転送される補助データ
- 補助データの影響を受けることで複雑化するデータキャッシュ
- データ管理に影響する VE スレッドのスケジュール

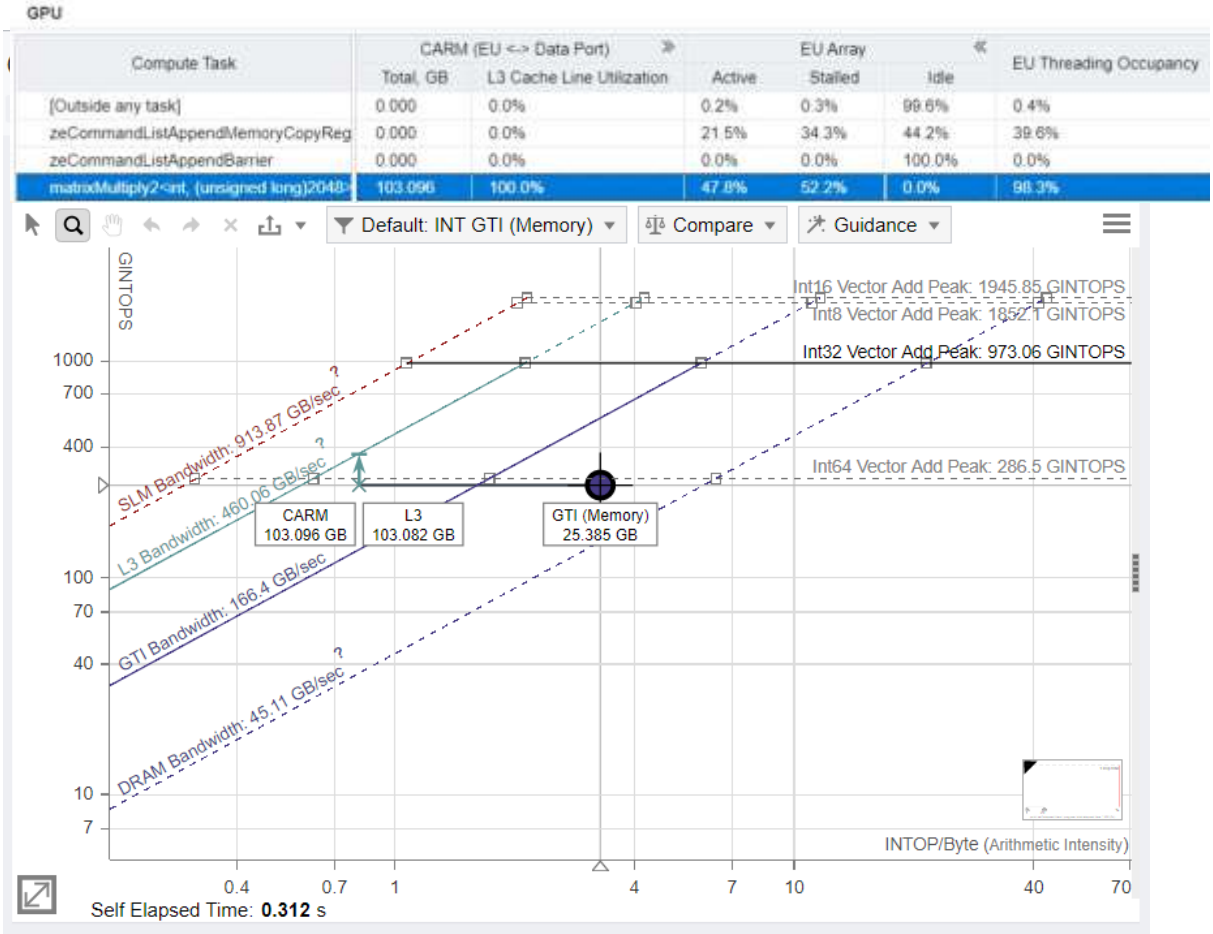
前述した理論上のケースと同様に、異なるアプリケーションの実装例をいくつか考えてみましょう。

**SLM を使用しない行列乗算アルゴリズムのデータブロック最適化**

この実装は、データのブロック化を行わない自然な行列乗算であり、最適化されたカーネルとデータフロー最適化のケースと同様です。

ソースコードでデータがブロック単位で整理されていなくても、コンパイラーはそのパターンを認識して行列配列へのアクセスを最適化します。その結果、キャッシュのデータ再利用率が高まり、カーネルのパフォーマンスは L2 キャッシュ依存に変化しました。ルーフライン・グラフは、アプリケーションの 1 つのカーネルに対応する 1 つの点を示しています。次のようにグラフ上の位置に基づきます。

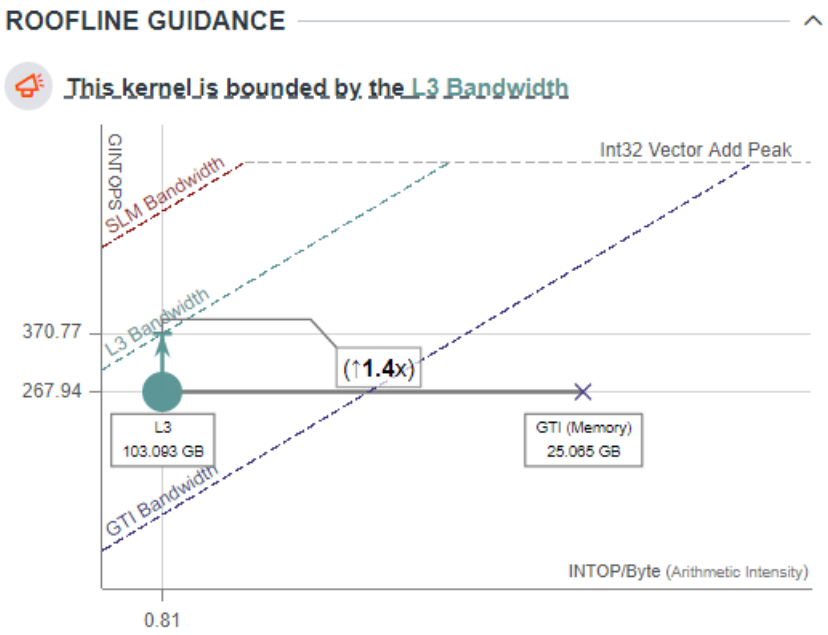
- カーネルはメモリー依存であり、対応する点は L2 帯域幅のリーフに近接しています。
- GTI のデータ・トラフィックは L2 キャッシュのデータ・トラフィックの 1/4 であり、データの再利用率が高いことが分かります。
- CARM と L2 のトラフィックはほぼ同じです。これは、キャッシュラインがアルゴリズムのデータで完全に満たされているため、間接的なキャッシュラインの使用率が 100% であることを示しています。



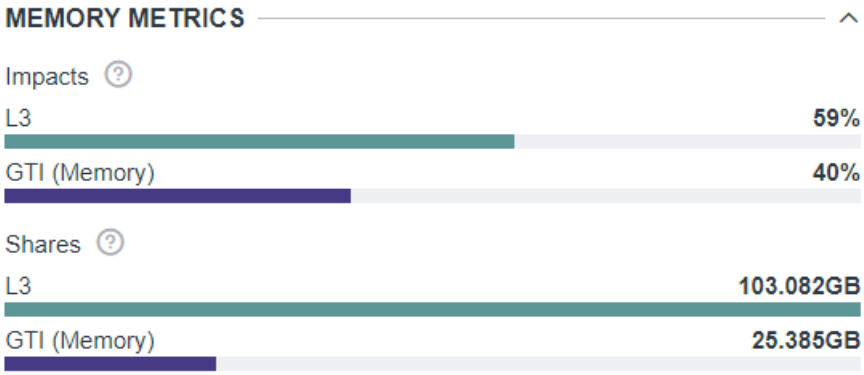

キャッシュラインの使用率が 100% であることを確認するため、**[GPU]** ペイングリッドの L2 キャッシュライン使用率メトリックを見ると 100% になっていることが分かります。グリッドはまた、VE スレッドの占有率が 98.1% で、スレッド間のスケジュールとデータ分散が適切であることを示しています。

以降のカーネルコードの最適化における制限を理解するため、インテル® Advisor がレポートする以下のデータを確認します。

- **[Roofline Guidance (ルーフライン・ガイド)]** ペインには、カーネルの制限のサマリーと最適化後のパフォーマンス向上に関する推定値が表示されます。行列乗算カーネルでは、L2 キャッシュの帯域幅が主な制限となります。L2 キャッシュをより効果的に使用し、同じ演算強度でも優れたデータ・アクセス・パターンで最大帯域幅を達成できれば、カーネルは 1.4 倍高速に動作します。

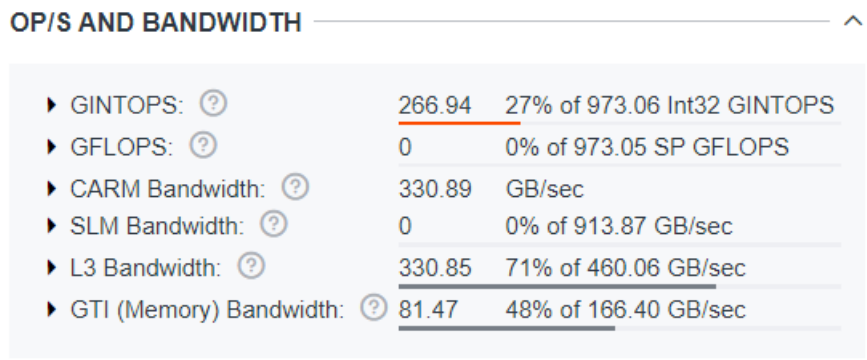



- メモリー・メトリック・ペインは、メモリーレベルによる影響 (異なるメモリーレベルからの要求に費やされる時間が合計時間に占める割合) を理解するのに役立ちます。このカーネルでは、GTI の影響は L2 キャッシュほど高くありませんが、カーネルの実行時間全体に占める割合はまだ高く、SLM などで L2 キャッシュの制限が解消されるとボトルネックになる可能性があります。共有メトリックでは、さまざまなメモリーレベルで処理されたデータ部分を視覚的に推測します。このケースでは、L2 キャッシュは GTI の 4 倍以上のデータを持ちます。




- OP/S と帯域幅ペインには、帯域幅の制限に関連する 1 秒あたりに測定された操作数とデータ・トラフィックが表示されます。このカーネルでは、サマリーに次のデータがレポートされます。

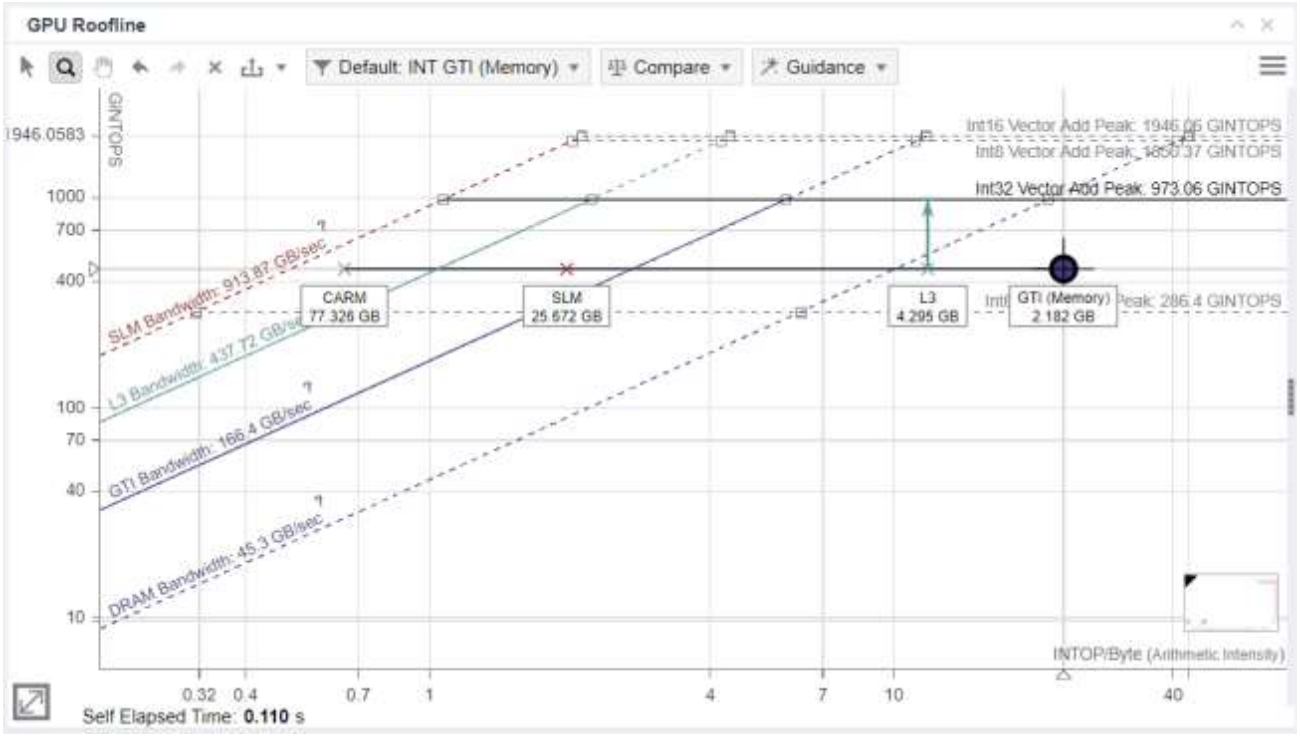
  - 理論上の SLM 帯域幅は L2 キャッシュのほぼ 3 倍ですが、この実装では SLM を使用しません。行列配列をブロック化してローカル共有データをして使用すると、L2 キャッシュの帯域幅の制限を緩和できます。
  - カーネルのパフォーマンスは、Int32 データの理論的な最大ピーク・パフォーマンスの 27% にしかすぎません。より適切なメモリーアクセスを実装することで、このカーネルのパフォーマンスを 3 倍以上向上できる可能性があります。




**SLM を使用した行列乗算アルゴリズムのデータブロック最適化**

前述のインテル® Advisor からの推奨事項にしたがって、行列乗算のデータブロック化を実装するため行列配列を小さなブロックに分割し、データブロックを SLM に配置することで $X^e$ コアレベルでの高速なデータ再利用を実現しました。

このデータブロック化による最適化の実装により、ルーフライン・グラフは次のようになります。




前回の結果とはデータの分布が変化しています。実行がメモリー依存ではなく計算依存であることが、全体のパフォーマンスやさらなる最適化には適していると考えられます。

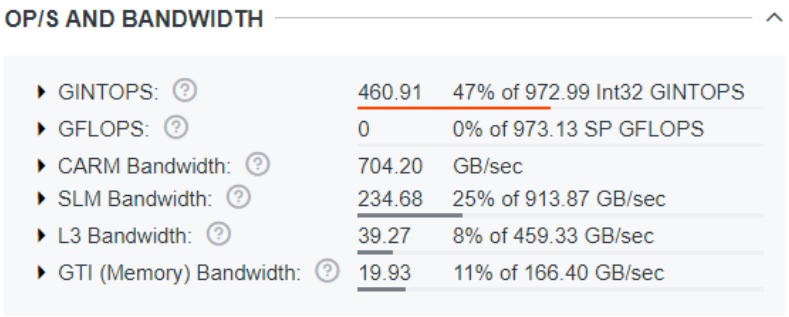
メモリーレベルの点には、いくつか注意すべきことがあります。

- SLM のトラフィックが L2 のトラフィックよりはるかに大きくなっています。L2 トラフィックはゼロではありません。これは、データブロックを L2 キャッシュに取り込んでから SLM にコピーして再利用するため予想されることです。

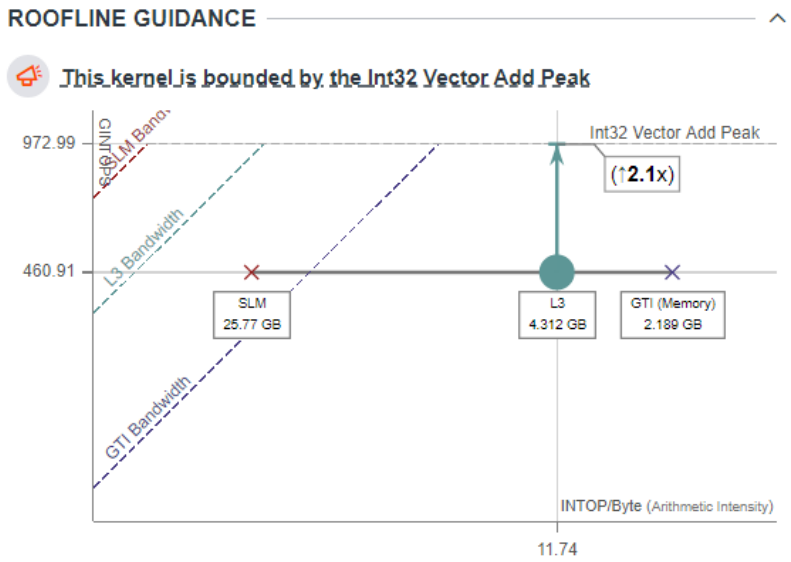
- CARM データのトラフィックは、SLM トラフィックの 3 倍です。この原因は明確ではありませんが、VE データポートが SLM から取得したデータをバッファーして逐次アクセスするために発生すると考えられます。この効果は悪いものではなく、VE に最も近いメモリーレベルのデータを再利用していることを意味します。

このアルゴリズムにおける実装のパフォーマンスの変化を理解するため、**[GPU Detail (GPU 詳細)]** ペインでデータを確認してみます。

- **[OP/S and Bandwidth (OP/S と帯域幅)]** ペインに示されるように、L2 と SLM の帯域幅は上限から離れています。カーネルのパフォーマンスは、1 秒あたりの整数操作 (INTOPS) の理論上限値の 47% まで向上しています。




- **[Roofline Guidance (ルーフライン・ガイド)]** グラフが示すように、カーネルのパフォーマンスは `Int32` ベクトル操作によって制限されています。これは、コンパイラーがコードを実装する際に選択した操作です。このグラフは、カーネルを最適化することで 2.1 倍高速化できることを示しています。



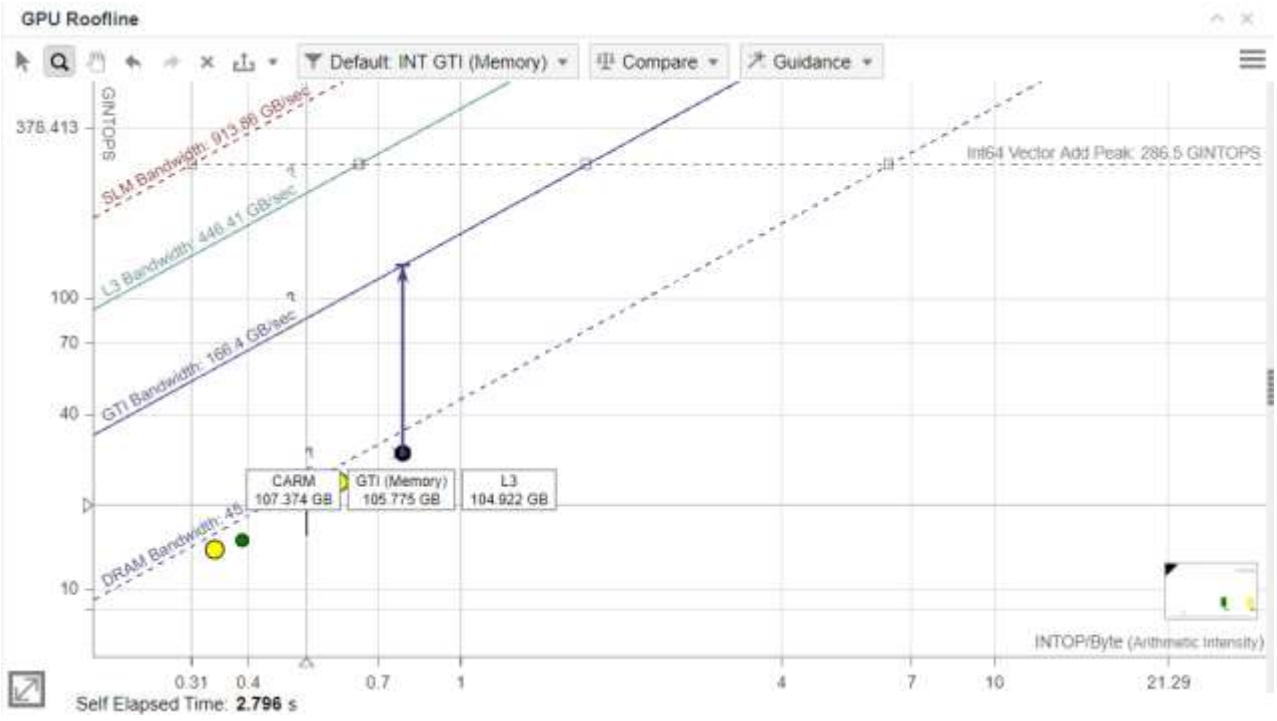

- **[Performance Characteristics (パフォーマンス特性)]** ペインに示されるように、VE は実行サイクルの 43.6% でストールしています。アルゴリズムは完全にベクトル化されているため、VE がストールする理由はほかにあるはずです。VE のパフォーマンスを最適化することで、**[Roofline Guidance (ルーフライン・ガイド)]** ペインに示されるように 2.1 倍のパフォーマンス向上を達成できる可能性があります。

インテル® VTune™ プロファイラーの GPU ホットスポット解析から GPU 計算/メディア・ホットスポット解析を実行して、VE がストールする原因を調査できます。

**STREAM ベンチマーク向けのデータ再利用可能なビッグデータ配列**

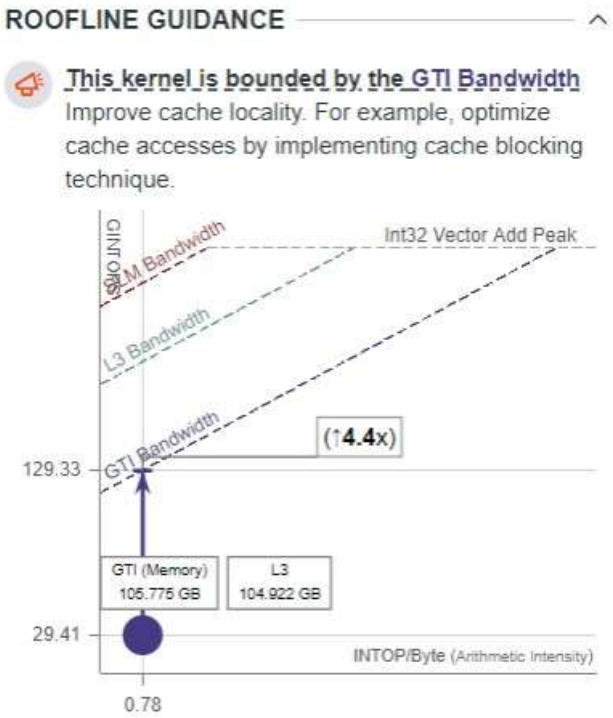
STREAM ベンチマークは、メモリーから大きなデータチャンクを取得して、基本的な計算カーネルを実行する小規模なアプリケーションです。Copy、Scalar、Add、および Triad を実行します。カーネルの計算操作はゼロかそれに近いため、カーネルはメモリー依存であることが予想されます。そのため、システムにおけるデータ帯域幅の上限を定義するために使用されます。

インテル® プロセッサー・グラフィックス (Tiger Lake†) の GPU ルーフラインの調査でベンチマークを解析した結果、ルーフライン・グラフにはベンチマーク・カーネルに対応する 4 つの点が表示されました。ドットは、DRAM 帯域幅のルーフ下にあり、左のメモリー依存側に配置されています。

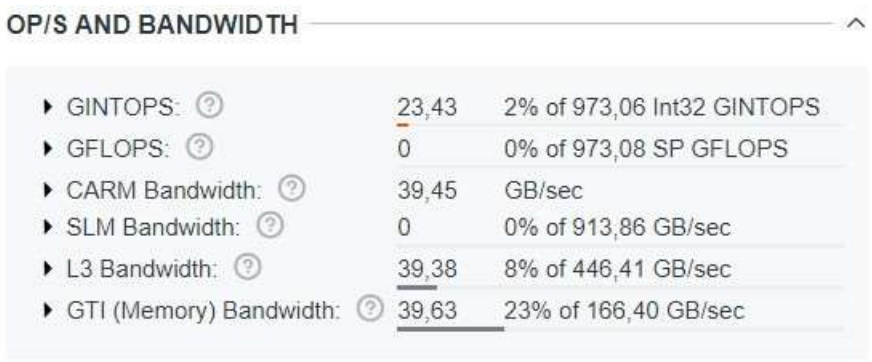



**[Roofline Guidance (ルーフライン・ガイド)]** グラフは、カーネルが GTI 帯域幅依存であることを示しており、メインのルーフライン・グラフが示すように DRAM 依存ではないことが分かります。この理由は、インテル® Advisor は、ハードウェアの制約によって統合 GPU の DRAM と XVE 間で転送されるデータ帯域幅を測定できないためです。

**[Roofline Guidance (ルーフライン・ガイド)]** では、キャッシュの局所性を改善することで、パフォーマンスを最適化して適切にデータを再利用できることが提案されています。このアドバイスは、データの帯域幅をテストする計算パフォーマンスが最適化の対象ではないほかのケースにも当てはまります。




**[OP/S and Bandwidth (OP/S と帯域幅)]** ペインで、達成されたメモリー帯域幅の具体的な数値を確認します。CARM、L2、および GTI の各ステージは、同程度の帯域幅を達成しているため、このベンチマークのボトルネックは最も遠いメモリー・インターフェイスであることに注意してください。



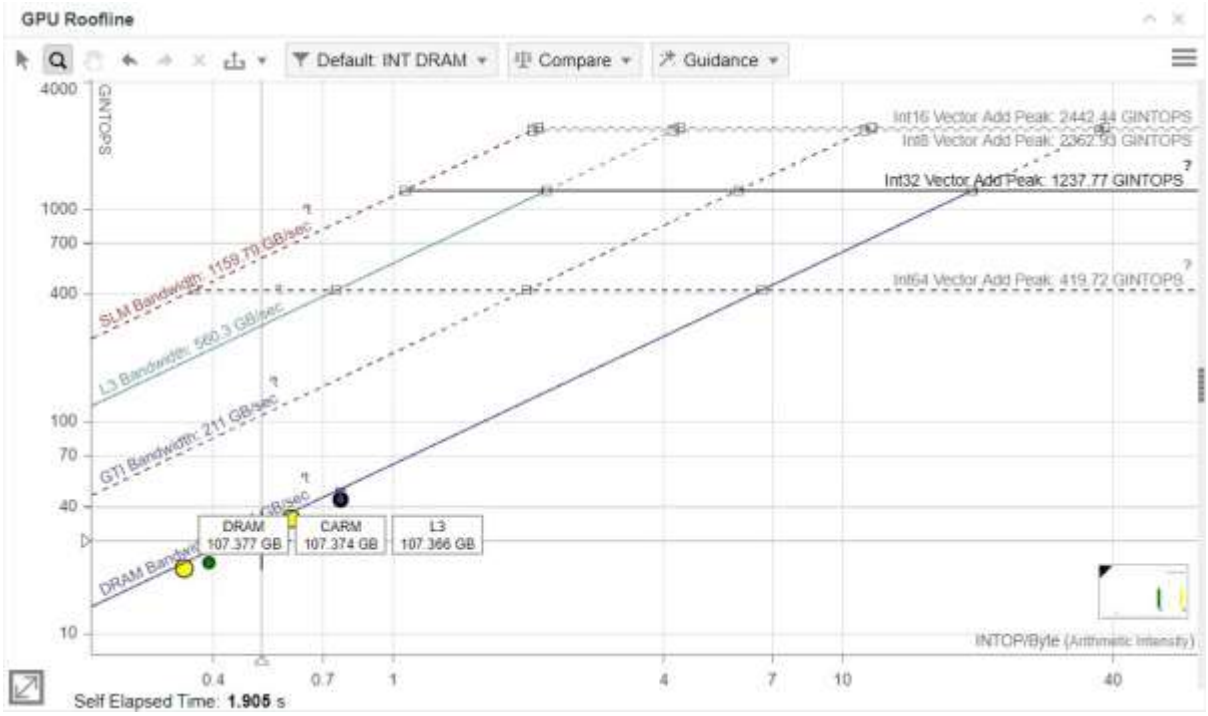

ここで、CARM、L2、および GTI のすべてのステージが同等の実効帯域幅を持ち、3 つのメモリー・コンポーネントは、特定のカーネルに対して相互にほぼ同じであることに注意してください。同一のルーフライン・コンポーネントを持つことは、キャッシュやレジスターファイルが再利用されず、データを取得するたびに外部メモリーにアクセスする必要があることを意味します。等価な CARM、L2、外部メモリーのルーフライン・コンポーネントは、一般的に「ストリーミング」パターンを示します。

これは、ベンチマークが最も離れたメモリー・インターフェイスがボトルネックであることも示しています。カーネルの帯域幅にわずかな差が見られますが、これは Copy/Scale カーネルのリード/ライトが等しく、Add/Triad カーネルではリードがライトの 2 倍で、リード帯域幅が高いことに影響を受けています。

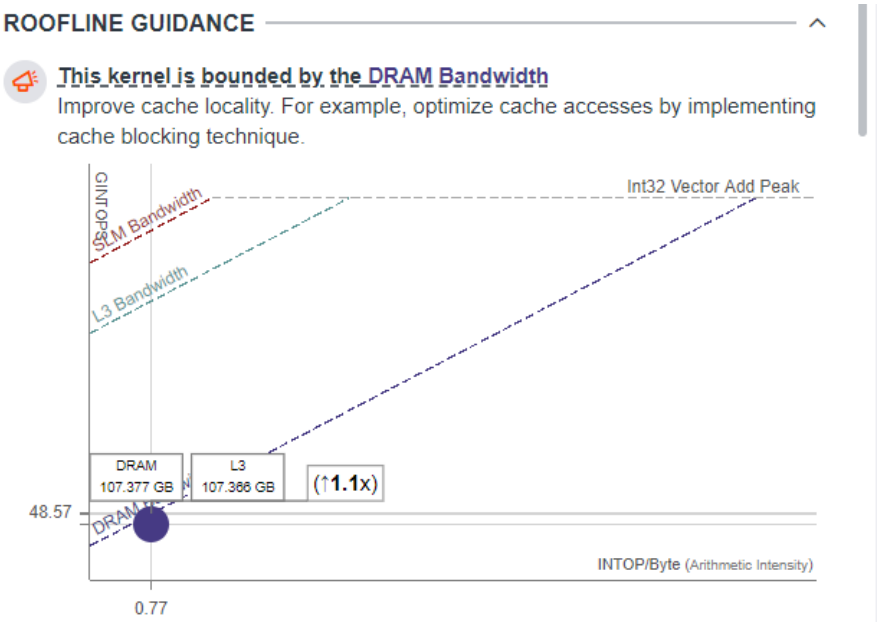
GPU

| Compute Task | GPU Memory | | | | | |
|---|---|---|---|---|---|---|
| | Total, GB | Read, GB | Write, GB | Bandwidth, GB/sec | Read, GB/sec | Write, GB/sec |
| [Outside any task] | 4.984 | 2.892 | 2.093 | 2.300 | 1.334 | 0.966 |
| zeCommandListAppendMemoryC | 6.463 | 3.247 | 3.217 | 27.502 | 13.814 | 13.687 |
| _ZTSZ17tuned_STREAM_Copyv | 107.904 | 54.376 | 53.528 | 37.885 | 19.091 | 18.793 |
| _ZTSZ18tuned_STREAM_Scalei | 105.775 | 53.313 | 52.462 | 37.837 | 19.071 | 18.766 |
| _ZTSZ16tuned_STREAM_AddvE | 160.174 | 107.105 | 53.069 | 39.589 | 26.472 | 13.117 |
| _ZTSZ18tuned_STREAM_TriadiE | 161.783 | 108.173 | 53.610 | 39.630 | 26.498 | 13.132 |

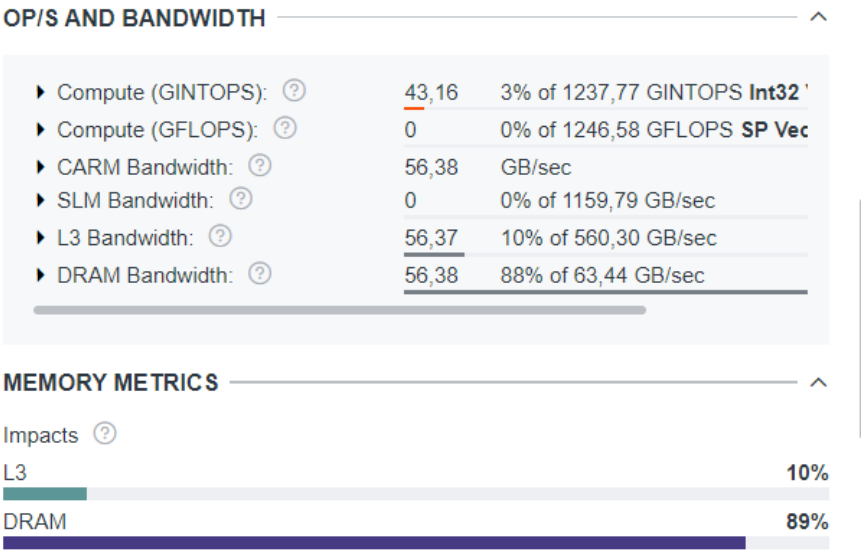
インテル® プロセッサー・グラフィックス (Tiger Lake†) のハードウェアの制限により、インテル® Advisor が DRAM と VE 間の帯域幅を測定できないことに対処するため、ディスクリートのインテル® Iris® Xᵉ MAX グラフィックスでベンチマークを解析してみます。その結果、ルーフライン・グラフには、DRAM 帯域幅のルーフの下に 4 つのカーネルの点が表示されました。




**[OP/S and Bandwidth (OP/S と帯域幅)]** ペインで、インテル® Advisor が DRAM を最も高いレベルのボトルネックとして識別できるようになりました。




**[OP/S and Bandwidth (OP/S と帯域幅)]** ペインおよび **[Memory Metrics (メモリーメトリック)]** ペインで示されるように、DRAM データ・トラフィックは理論上の上限に近く、stream ベンチマークは実際のデータフローの制限を測定します。

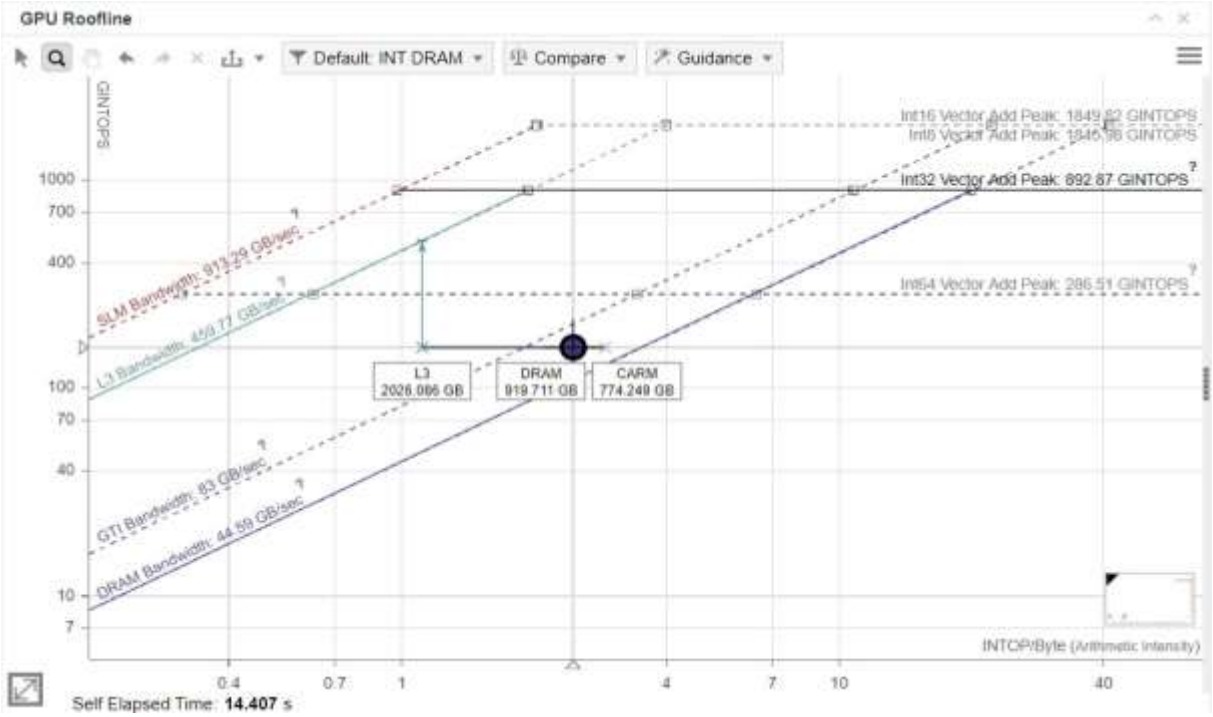



**ステンシル・データ・パターン・ベンチマークにおける部分的なデータアクセスの効率**

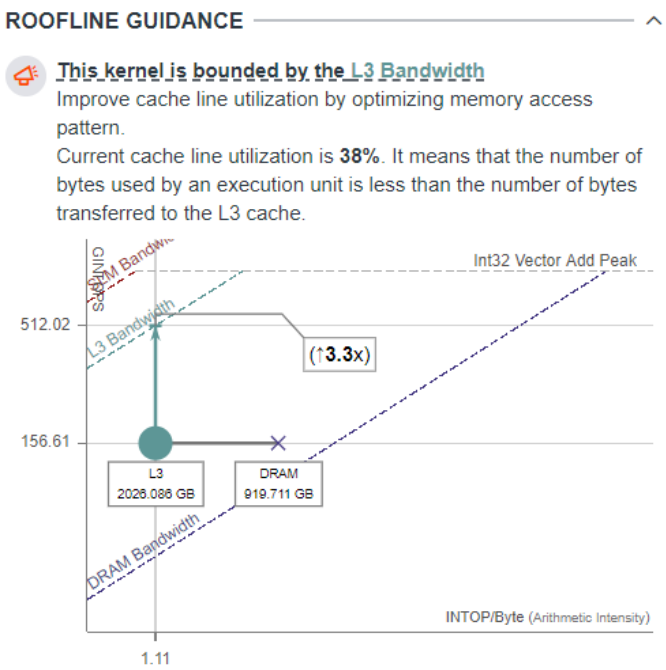
最も興味深いケースは、データアクセスがコンパクトであってローカルの範囲が限られていても、グローバルではアクセスがスパースである場合です。例えば、ステンシルベースのカーネルの計算では、X と Y の 2 軸のデータはメモリー空間でシーケンシャルにアクセスされますが、Z 軸のデータは大きなストライド単位でアクセスされるなど、実際のアプリケーションでは良くあることです。

第 12 世代のインテル® プロセッサー・グラフィックスで動作する SPEC ACCEL ベンチマーク・セットに含まれる 504.polbm アプリケーションを解析してみます。このベンチマーク・アプリケーションは C 言語で記述されており、OpenMP* によって GPU へオフロードされています。倍精度データタイプで動作しますが、Gen12 アーキテクチャーのインテル® プロセッサー・グラフィックスは、整数タイプのデータでしか計算をシミュレートできません。そのため、整数操作のルーフライン・グラフを調査します。

GPU ルーフライン・グラフは、ベンチマークのカーネルに対し 1 つの点を示しています。点は、メモリールーフと計算ルーフの間に位置するため、GPU のパラメーターを変更すると (メモリー帯域幅の大きいハードウェアで解析を実行するなど)、カーネルの点がメモリー依存から計算依存にグラフ上で移動することがあります。




**[Roofline Guidance (ルーフライン・ガイド)]** ペインで示されるように、カーネルは L2 キャッシュの帯域幅に依存しています。また、インテル® Advisor は、ステンシルベースのカーネルで予測される、低いキャッシュライン利用率を検出します。




一般に、ステンシルベースのカーネルでデータアクセスを最適化するには、データの局所性を高めるのに SLM を使用し、データ軸ごとに SIMD 並列化を行うよう、データ配置を変更する手法を導入する必要があります。ただし、ベンチマークではデータ配置を変更できないため、最適化はすべてグラフィックス向けのコンパイラーによって行われます。

### まとめ

インテル® Advisor による GPU ルーフラインの調査パースペクティブは、インテル® GPU にオフロードされたカーネルのパフォーマンスを調べるのに役立つツールです。ルーフラインの結果は、理論上の極端なケースを想定したほうが理解しやすいと考えられます。このようなケースではハードウェア上の制限しかないため、パフォーマンス最適化の方針が明確になります。しかし、実際のアプリケーションでは、複数の制限要因が存在する可能性があります。ルーフラインは、最も影響のある要因を特定することで、パフォーマンスの問題に対処します。1 つの制限を排除すると、解析では次に対処すべき要因が示され、パフォーマンスが理論上のハードウェアの上限に近づき、最適化によって改善の余地がなくなるまで作業を繰り返します。

## OpenCL* アプリケーション向けインテル® インターセプト・レイヤー

OpenCL* アプリケーションのインターセプト・レイヤーは、デバッグやパフォーマンス解析で OpenCL* 呼び出しをインターセプトして変更できるツールです。OpenCL* アプリケーションのインターセプト・レイヤーを使用する場合、アプリケーションやドライバーを変更する必要はありません。

OpenCL* アプリケーションのインターセプト・レイヤーは、OpenCL* ICD ローダー (通常) または OpenCL* 実装 (まれに) になりすまし、アプリケーションが実際の OpenCL* ICD ローダーをロードしようとするとロードされます。OpenCL* アプリケーションのインターセプト・レイヤーの初期化の一環として、実際の OpenCL* ICD ローダーをロードし、OpenCL* エントリーポイントへの関数ポインターを取得します。次に、アプリケーションが OpenCL* API を呼び出すたびにインターセプトされ、変更の有無にかかわらず OpenCL* に渡されます。

OpenCL* インターセプト・レイヤーのリポジトリーへアクセスします。

```
$ git clone https://github.com/intel/opencl-intercept-layer
```

コントロールのすべては、https://github.com/intel/opencl-intercept-layer/blob/main/docs/controls.md (英語) で文書化されています。

コントロールの詳細については、インターセプトのドキュメント (英語) を参照してください。

実行するには、次の設定を行います。

```
$ export
CLI_OpenCLFileName=/opt/intel/inteloneapi/compiler/latest/linux/lib/libOpenCL.so.1
$ export LD_LIBRARY_PATH=/home/opencl-intercept-layer/build/intercept:$LD_LIBRARY_PATH

$ export SYCL_BE=PI_OPENCL

$ CLI_ReportToStderr=0 CLI_ReportToFile=1 CLI_HostPerformanceTiming=1
CLI_DevicePerformanceTiming=1 CLI_DumpDir=. ./matrix.dpcpp
```

これにより `cliintercept_report.txt` ファイルが生成されます。このファイルには、以下に示すデータと表が含まれます。

- Total Enqueues: 2
- Total Time (ns): 1604325652

### ホストのパフォーマンス・タイミングの結果

| 関数名 | 呼び出し | 時間 (ns) | 時間 (%) | 平均 (ns) | 最小 (ns) | 最大 (ns) |
|---|---|---|---|---|---|---|
| clBuildProgram | 1 | 337069812 | 21.01% | 337069812 | 337069812 | 337069812 |
| clCreateBuffer | 3 | 3393909 | 0.21% | 1131303 | 140325 | 2036170 |
| clCreateCommandQueue (プロパティーあり) | 1 | 5221 | 0.00% | 5221 | 5221 | 5221 |
| clCreateContext | 1 | 33639 | 0.00% | 33639 | 33639 | 33639 |
| clCreateKernel | 1 | 11713 | 0.00% | 11713 | 11713 | 11713 |
| clCreateProgramWithIL | 1 | 153337 | 0.00% | 153337 | 153337 | 153337 |
| clEnqueueNDRangeKernel ( _ZTS9Matrix1_2IfE ) | 3 | 3102488 | 0.19% | 3102488 | 3102488 | 3102488 |
| clEnqueueReadBufferRect | 1 | 1099684 | 0.07% | 1099684 | 1099684 | 1099684 |
| clGetContextInfo | 8 | 4720 | 0.00% | 590 | 160 | 1997 |
| clGetDeviceIDs | 12 | 53004 | 0.00% | 4417 | 504 | 14853 |
| clGetDeviceInfo | 30 | 85695 | 0.01% | 2856 | 133 | 19920 |
| clGetExtensionFunctionAddressForPlatform | 3 | 6446 | 0.00% | 2148 | 1317 | 3687 |
| clGetKernelInfo | 2 | 716 | 0.00% | 358 | 169 | 547 |
| clGetPlatformIDs | 2 | 1198290216 | 74.69% | 599145108 | 715 | 1198289501 |
| clGetPlatformInfo | 12 | 22538 | 0.00% | 1878 | 404 | 7326 |
| clReleaseCommandQueue | 1 | 1744 | 0.00% | 1744 | 1744 | 1744 |
| clReleaseContext | 1 | 331 | 0.00% | 331 | 331 | 331 |
| clReleaseDevice | 6 | 6365 | 0.00% | 1060 | 491 | 1352 |
| clReleaseEvent | 2 | 2398 | 0.00% | 1199 | 992 | 1406 |
| clReleaseKernel | 1 | 2733 | 0.00% | 2733 | 2733 | 2733 |
| clReleaseMemObject | 3 | 45464 | 0.00% | 15154 | 10828 | 22428 |
| clReleaseProgram | 1 | 51380 | 0.00% | 51380 | 51380 | 51380 |
| clRetainDevice | 6 | 8680 | 0.00% | 1446 | 832 | 2131 |
| clSetKernelArg | 20 | 6976 | 0.00% | 348 | 180 | 1484 |
| clSetKernelExecInfo | 3 | 1588 | 0.00% | 529 | 183 | 1149 |
| clWaitForEvents | 6 | 60864855 | 3.79% | 10144142 | 928 | 60855555 |

### デバイスのパフォーマンス・タイミングの結果

| 関数名 | 呼び出し | 時間 (ns) | 時間 (%) | 平均 (ns) | 最小 (ns) | 最大 (ns) |
|---|---|---|---|---|---|---|
| _ZTS9Matrix1_2IfE | 1 | 58691515 | 99.98% | 58691515 | 58691515 | 58691515 |
| clEnqueueReadBufferRect | 1 | 13390 | 0.02% | 13390 | 13390 | 13390 |

レポートには、ホストとデバイスの両方に関する詳細なタイミングデータが含まれます。

## GPU 向けのインテル® プロファイル・ツール・インターフェイス

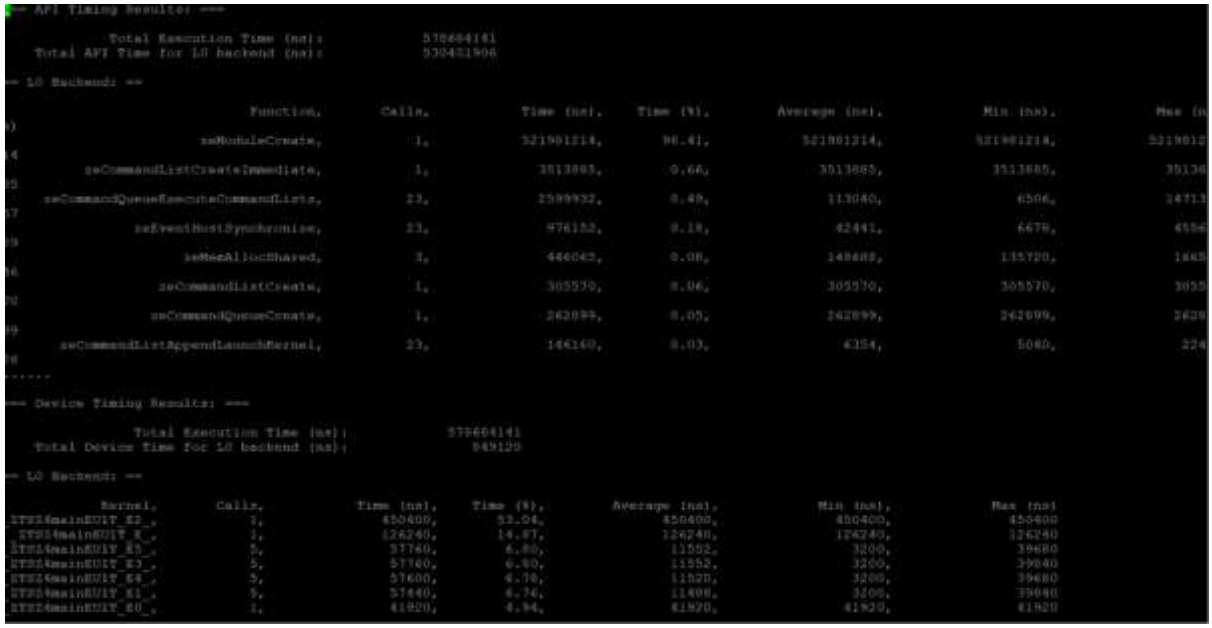
OpenCL* アプリケーションのインテル® インターセプト・レイヤーは OpenCL* バックエンド専用です。レベルゼロと OpenCL* バックエンドを使用してアプリケーションをトレースするには、GPU 向けのインテル® プロファイル・ツール・インターフェイス (英語) が必要です。

GPU 向けのインテル® プロファイル・ツール・インターフェイスでは、独自のツール開発を行う開発者向けにサンプルコードとドキュメントを提供します。また、軽量のコマンドライン・パフォーマンス・ツールを使用する開発者向けのツールセットも用意されています (onetrace (英語) はその 1 つです)。

OpenCL* アプリケーションのインテル® インターセプト・レイヤーと同様に、**onetrace** は、アプリケーションの実行中にホストとデバイスの両方でアクティビティーをトレースします。

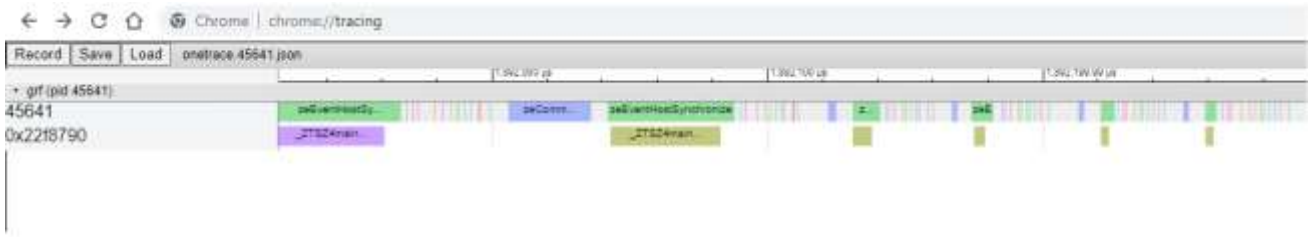
デフォルトでは、**onetrace** はデバイス上で実行されるカーネルやコマンド、およびホスト上のレベルゼロや OpenCL* 呼び出しをトレースします。

```
$ onetrace <application>
```




**onetrace** を使用してホスト/デバイス/カーネルのタイムライン・データを収集し、**chrome://tracing** を使用して表示できます。

```
$ onetrace --chrome-call-logging --chrome-device-timeline <application>
```




ソースコードを含む **onetrace** とその他のツールの詳細については、インテル® プロファイル・ツール・インターフェイス (英語) のリポジトリーを参照してください。

# GPU デバイスの構成

## はじめに

アプリケーションから最高のエンドツーエンドのパフォーマンスを引き出すには、システムレベルでデバイス構成を変更することが必要な場合もあります。これらの変更は、`sudo` アクセスが必要かどうかに応じて、システム管理者が行うことが求められることがあります。

この節では、Linux* 環境でパフォーマンスを得るため、特定のデバイス向けの構成を変更するガイドラインと手順を説明します。

## GPU ドライバーまたはプラグインのインストール (オプション)

インテル、AMD、または NVIDIA の GPU で動作する、C++ および SYCL* を使用した oneAPI アプリケーションを開発できます。

特定の GPU 向けのアプリケーションを開発および実行するには、対応するドライバーまたはプラグインをインストールする必要があります。

- インテル® GPU を使用する場合は、最新のインテル® GPU ドライバーをインストール (英語) します。
- インテル® oneAPI DPC++ コンパイラーを使用して AMD* GPU で動作するアプリケーションを開発する場合は、Codeplay が提供している oneAPI for AMD* GPU プラグインをインストールします。
- インテル® oneAPI DPC++ コンパイラーを使用して NVIDIA* GPU で動作するアプリケーションを開発する場合は、Codeplay が提供している oneAPI for NVIDIA* GPU プラグインをインストールします。

## GPU 周波数を固定することのパフォーマンスへの影響

アプリケーションが計算のほとんどに GPU を使用する場合、最適な周波数に GPU を固定することを推奨します。このようなアプリケーションの例として、デバイスにオフロードされたアルゴリズムが計算集約型のコードを持つハイパフォーマンス・コンピューティングのワークロードがあります。

カーネルの実行時間が短くても、データ交換に多くの時間を費やすメモリー依存型のアプリケーションでは、GPU を高い周波数に固定する効果があまり得られないばかりか、有害である場合があります。

理論上の HW 最大周波数である融合 GPU 最大周波数は、以下のように `sysfs` ハンドルを使用して取得できます。

```
for (( i=1; i<$num_cards; i++ ))
do
  for (( j=0; j<$num_tiles; j++ ))
  do
    cat /sys/class/drm/card$i/gt/gt$j/rps_RP0_freq_mhz;
  done
done
```

ユーザーは、`rps_max_freq_mhz` を使用して、動的に変更可能な現在の最大ソフトウェア周波数も取得できます。

```
for (( i=1; i<$num_cards; i++ ))
do
  for (( j=0; j<$num_tiles; j++ ))
  do
    cat /sys/class/drm/card$i/gt/gt$j/rps_max_freq_mhz;
  done
done
```

サーバー用の Linux* カーネル・モード・ドライバー (i915) のデフォルトポリシーは、周波数要求範囲を次のように設定します。

```
rps_min_freq_mhz = rps_max_freq_mhz = rps_boost_freq_mhz = rps_RP0_freq_mhz
```

そのため、多くのワークロードは、このポリシーですぐに最高のパフォーマンスを得られる可能性があります。

ユーザーは、DRM Linux* カーネルドライバーで提供される以下の `sysfs` ハンドルを使用して周波数を設定できます。

```
for (( i=1; i<$num_cards; i++ ))
do
  for (( j=0; j<$num_tiles; j++ ))
  do
    echo $desired_freq > /sys/class/drm/card$i/gt/gt$j/rps_min_freq_mhz;
    echo $desired_freq > /sys/class/drm/card$i/gt/gt$j/rps_max_freq_mhz;
    echo $desired_freq > /sys/class/drm/card$i/gt/gt$j/rps_boost_freq_mhz;
  done
done
```

周波数固定の注意事項:

- ファームウェアは、許可された周波数を最終的に決定します。
- サーマル/電力バジェットによってスロットリングが発生する場合があります。
- 周波数が固定されると動的スケーリングは行われません。サーバー・プラットフォームでは、i915 ポリシーは周波数をブート時に `rps_max_freq_mhz` に固定します。

## インテル® Xe リンクのオン/オフの切り替え

インテル® Xe リンクは、高速接続ファブリック・ハードウェアであり、スケールアップとスケールアウト操作に必要な高速データ転送機能を提供します。この機能は、クラスター規模で展開された HPC アプリケーションにおいて、GPU 間およびノード間のデータ転送に使用されます。この機能を使用しないアプリケーションでは、ハードウェア・リソースの電源をオフにして、GPU 計算エンジンを低周波数で動作させることが有益な場合もあります。

次のコマンドラインは、インテル® Xe リンク電源をオフにする方法を説明しています。

```
modprobe -r iaf;
for i in {0..$num_cards}; do
  cat /sys/class/drm/card$i/iaf_power_enable;
done;

for i in {0..$num_cards}; do
  echo 0 >  /sys/class/drm/card$i/iaf_power_enable;
done;

for i in {0..$numcards}; do
  cat /sys/class/drm/card$i/iaf_power_enable;
done
```

`$num_cards` は、`/sys/class/drm/` ディレクトリーの一覧を表示し、`card*` サブ・ディレクトリーがいくつ存在するか確認できます。

# GPU 上でのメディア・グラフィックス計算

ここまでは、GPGPU の最適化手法とツールに焦点を当ててきました。メディア・グラフィックス・コンピューティングでは、ハードウェア、特にメディアエンジンなどの専用ユニットの利用率を最大化するため、新しい最適化手法とツールセットも重要です。

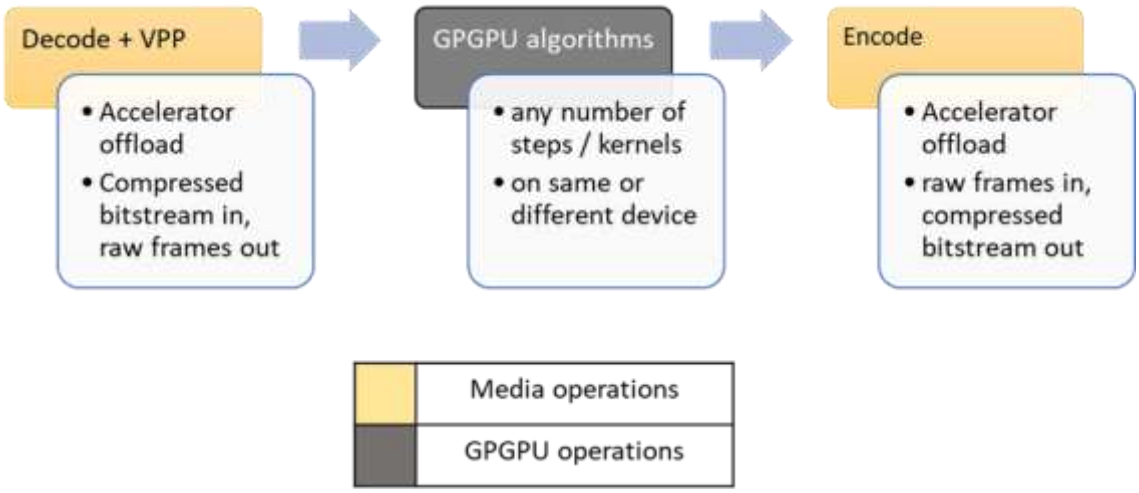
- メディア・パイプラインの最適化
- インテル® グラフィックス・パフォーマンス・アナライザーによるパフォーマンス解析

## メディア・パイプラインの最適化

メディア処理は、入力と出力が明確に定義された比較的大きなアルゴリズムであるため、ハードウェア・アクセラレーションの理想的な候補であると言えます。ビデオ処理のハードウェア機能は、業界標準のフレームワークであるインテル® ビデオ・プロセシング・ライブラリー (インテル® VPL) や、Linux* 向けのビデオ・アクセラレーション API (VA-API) や Windows* 向けの Microsoft DirectX* などの低レベル/オペレーティング・システム固有のアプローチでアクセスできます。どの方法を選択するかは、多くの要因に影響されます。しかし、マルチストリームによる並列化やデータの局所性の最大化といった基本原則は、すべてに当てはまります。

ビデオ処理と GPGPU ワークの主な違いは、すべてのアクセラレーター API オプションに当てはまります。典型的な GPGPU 最適化の多くは、大きなグリッドのワークを複数の処理ユニットに分割することに最適化の重点を置いています。ハードウェア・アクセラレーションによるメディア処理は、シリコン上に実装されています。これらはフレーム単位で動作し、通常、フレームストリームによってワークが分割されます。

メディア最適化のステップは、他のセクションで説明する GPGPU のワークロードと異なります。それらは GPGPU のワークの前後に追加することができます。メディアステップは、GPGPU ステップに入力データを供給、または GPGPU ステップから出力データを取得します。例えば、次のようになります。

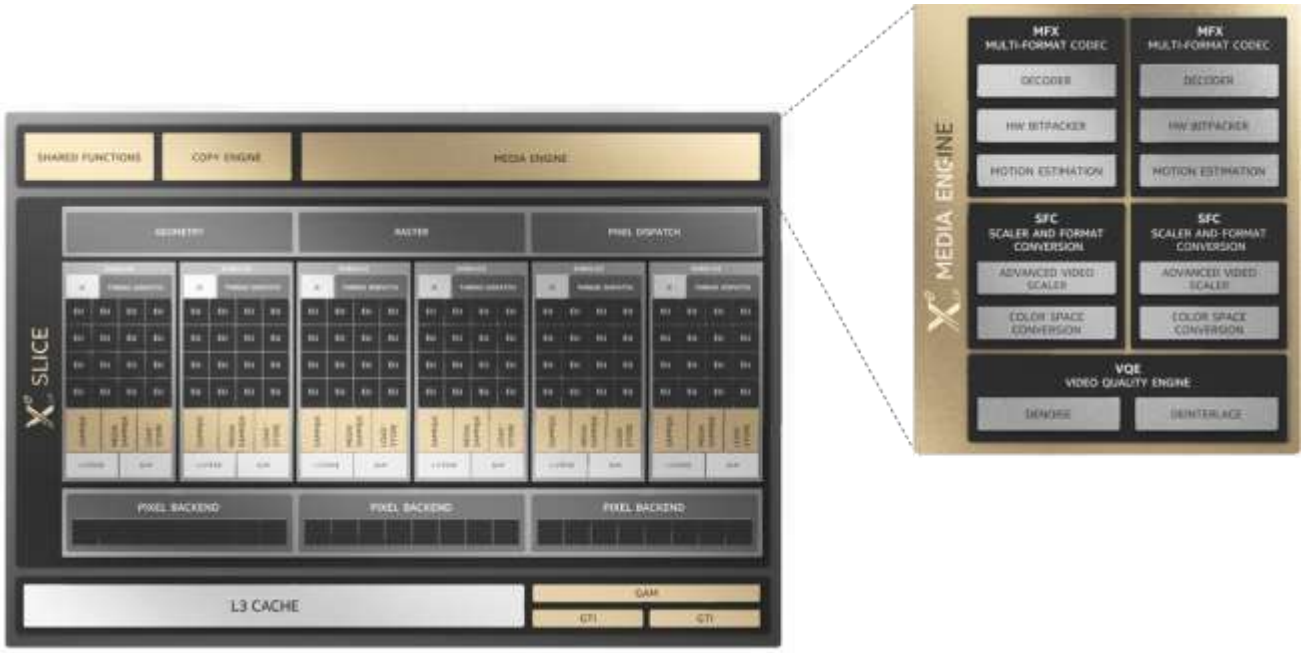



- メディア・エンジン・ハードウェア
- ハードウェア・アクセラレーション向けのメディア API オプション
- メディア・パイプラインの並列処理
- メディア・パイプラインの相互運用とメモリー共有
- SYCL* によるぼかし (Blur) の例

職場で会議をストリーミングしたり、自宅で動画を視聴したり、今日の世界では、ビデオ・ストリーミングが普及しており、高い品質が期待されます。新しいメディア・エンジン・ハードウェアを利用することで、ストリーミングを高速化したり、高品質でストリーミングしたり、低電力でストリーミングすることができます。このハードウェア・ソリューションは、ビデオデータを扱うパイプラインでは、エンドツーエンドのパフォーマンスを考慮する上で重要なものです。

## メディア・エンジン・ハードウェア

「インテル® Xe GPU アーキテクチャー」の章で説明したように、Xe インテル® データセンター GPU フレックス・シリーズといくつかのインテル® GPU は、完全にアクセラレートされたビデオデコード、エンコードおよび処理能力を備えたビデオエンジンを搭載しています。これは、インテル® クイック・シンク・ビデオと呼ばれることもあります。メディアエンジンは、計算エンジン (ベクトルまたは行列エンジン) から完全に独立して動作します。




アプリケーションはいくつかのコンポーネントを利用できます。

- マルチフォーマット・コーデック (MFX): ハードウェア・デコードとエンコード、一部の構成では、1) 動き推測 + ビット・パッキング、2) 完全固定機能/低電力、の 2 つのエンコードを含みます。
- スケーラーとフォーマット変換 (SFC): リサイズ (主にダウン・スケーリング向け)、NV12 や BGRA などのカラー・フォーマット変換。
- ビデオ品質エンジン: ノイズ除去やインターレースなど、複数のフレーム処理操作を実行。

このハードウェアは、独自の命令キューとクロックを持っているため、低消費電力で動作するように構成すれば、完全な固定機能のワークを低消費電力で実行できます。GPU スライスの機能をほかのワークに利用することもできます。

## サポートされるコーデック

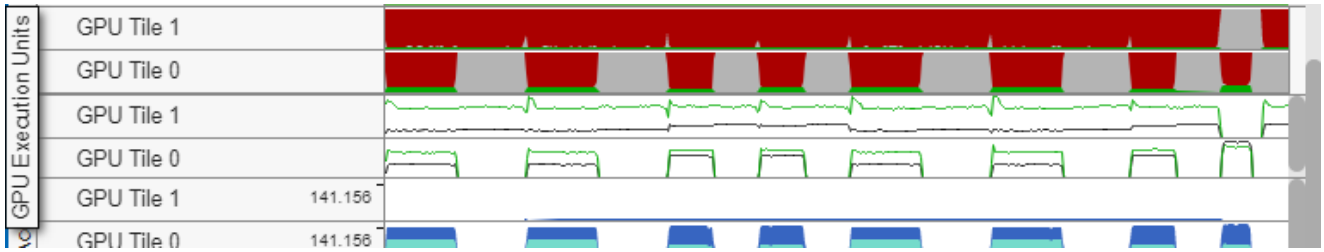
GPU ハードウェアの世代が新しくなるたびに、新しいコーデックの機能が追加されています。

| | | AVC | MPEG 2 | JPEG | VP8 | HEVC 8-bit | HEVC 8-bit 422 | HEVC 8-bit 444 | HEVC 10-bit | HEVC 10-bit 422 | HEVC 10-bit 444 | HEVC 12-bit | HEVC 12-bit 422, 444 | VP9 8-bit | VP9 8-bit 444 | VP9 10-bit | VP9 10-bit 444 | VP9 12-bit, 12-bit 444 | AV1 8-bit, 10-bit |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CPU | | D/E* | D | D/E | | D/E | | | D/E | | | | | | | | | | D/E |
| Media SDK GPU | 5th Generation Intel® Core (BDW) | D/Es | D/Es | D | D | | | | | | | | | | | | | | |
| | 6th Generation Intel® Core (SKL) | D/E/Es | D/Es | D/E | D | D/Es | | | | | | | | | | | | | |
| | Intel Atom® Processor E3900 series (APL) | D/E/Es | D | D/E | D | D/Es | | | D | | | | | D | | | | | |
| | 7th Generation Intel® Core (KBL-x) | D/E/Es | D/Es | D/E | D/Es | D/Es | | | D/Es | | | | | D | | D | | | |
| | 10th Generation Intel® Core (ICL) | D/E/Es | D/Es | D/E | D/Es | D/E/Es | D/Es | D/E | D/E/Es | D/Es | D/E | | | D/E | D/E | D/E | D/E | | |
| | Intel Atom® Processor X Series (EHL) | D/E | D | D/E | D | D/E | D | D/E | D/E | D | D/E | | | D/E | D/E | D/E | D/E | | |
| oneVPL GPU | Intel® Iris® Xe (TGL/RKL/ADL) Intel® Iris® Xe MAX (DG1) | D/E/Es | D/Es | D/E | D (TGL only) | D/E/Es | D/Es | D/E | D/E/Es | D/Es | D/E | D/Es | D | D/E | D/E | D/E | D/E | D | D |
| | Intel® ARC | D/E | D/E | D/E | D | D/E | D/E | D/E | D/E | | D/E | D/E | D | D/E | D/E | D/E | D/E | D | D/E |

注: この表では、2 つのエンコードを表しています。

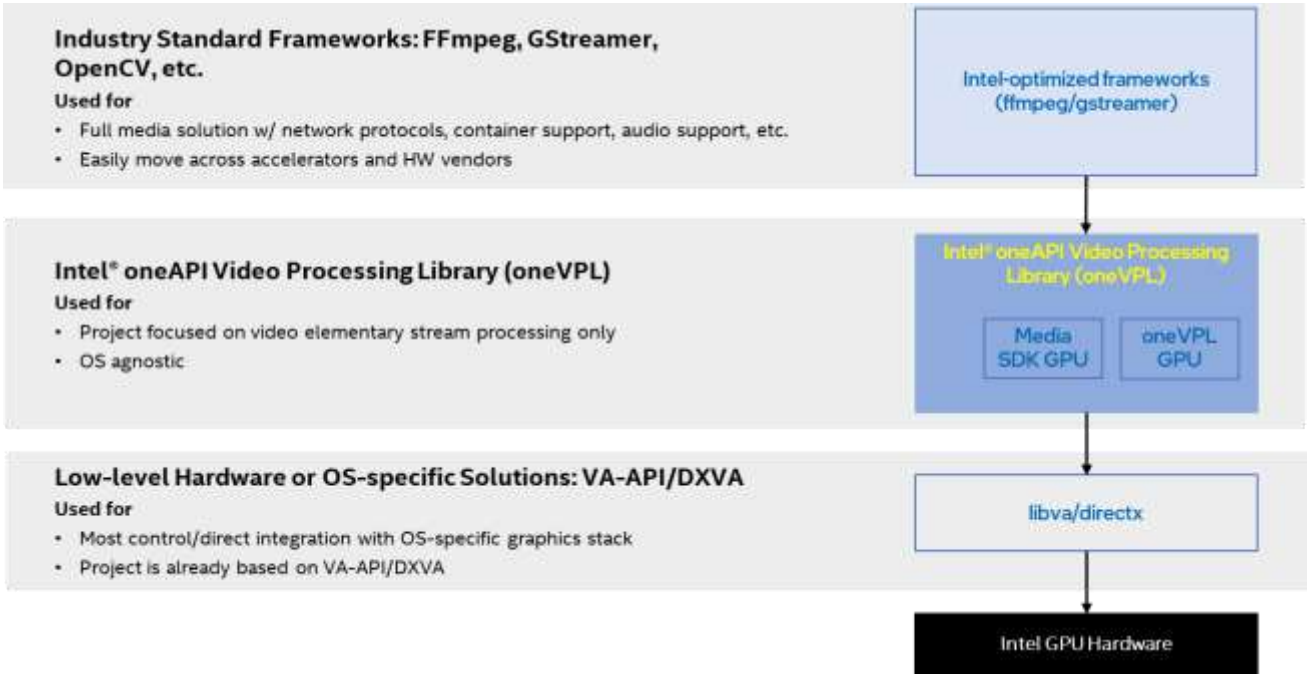
- E = 低消費電力 VDEnc によるハードウェア・エンコード
- Es = によるハードウェア・エンコード (PAK) + シェーダー (メディアカーネル + VME)

インテル® Arc™ A シリーズおよびインテル® サーバー GPU (Arctic Sound-M†) に AV1 エンコードが追加されました。VP9 の後継となるこの最先端の技術は、スタック、セグメント化、フィルム・グレイン・フィルター処理、その他の新機能によるエンコード制御を追加します。これらの機能により、特定のビットレートでエンコード品質を向上させたり、ビットレートを下げて品質を向上させることができます。



## ハードウェア・アクセラレーション向けのメディア API オプション

インテル® アーキテクチャー (CPU、GPU) でビデオ処理を高速化するには、いくつかの方法があります。最も効果的なオプションを選択するには、選択するツールと目的が一致していることを確認します。




このように、ツールには高レベルなものと低レベルなものがあります。libva* や DirectX* のような OS 固有のツールで提供される極めて低レベルの制御は必要ですか? また、このような低レベルのアプリケーションを開発する十分な時間的余裕はありますか? または、Linux* から Windows* に容易にコードを移植でき、より高レベルのツールを利用してコーディングの時間を節約するほうが重要ですか?

アプローチの選択肢を要件に適合させるには、次の表を参考にしてください。

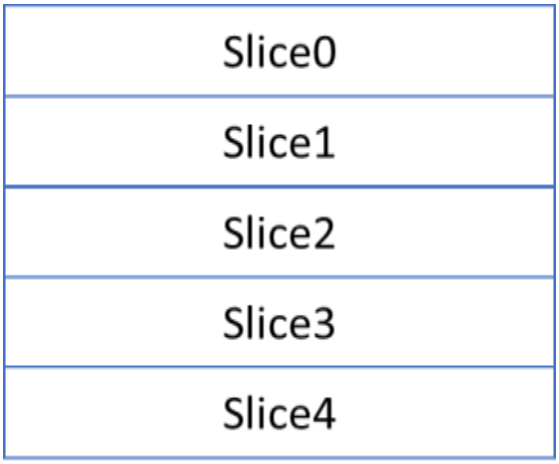
| | インテル® ビデオ・プロセシング・ライブラリー | メディア・フレームワーク (FFmpeg* & GStreamer) | 低レベル/OS 固有のソリューション (libva & DirectX) |
|---|---|---|---|
| 機能 | フレーム処理演算を限定した初歩的なビデオストリーム処理 | 完全なスタック (ネットワーク・プロトコル、コンテナのサポート、オーディオのサポート) | OS グラフィックスのスタックを直接操作 |
| ハードウェア機能の制御レベル | 中 | 低 | 高 |
| 移植性 | 高 | 高 | 低 |

## メディア・パイプラインの並列処理

GPGPU では、並列処理は ND Range の分割方法と関連するエッジ条件などに注目します。複数のアクセラレーターが分割された空間で動作し、グリッド全体で同じアルゴリズムを実行することができます (SIMD)。これは、エンコードとデコードの例ではありません。

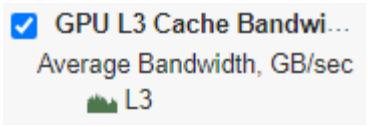
エンコーダーやデコーダーの内部実装を詳しく調査して、各フレームを処理する際に並列化の可能性を見つけるのではなく、通常は、処理全体をブラックボックスとして扱います。コーデックのデコード実装は、あるボックスを別のボックスに置き換えるように、交換可能であることを意図しています。エンコードの置き換えでは、さらに広範囲のパラメーターの影響を考慮する必要があるため複雑になります。しかし、考え方は同じであり、ホットスポットや内部ループの最適化ではなく、全体の最適化をハードウェアに最適なものに置き換えます。

理論的には、フレーム内をスライス単位で演算を並列化することができます。

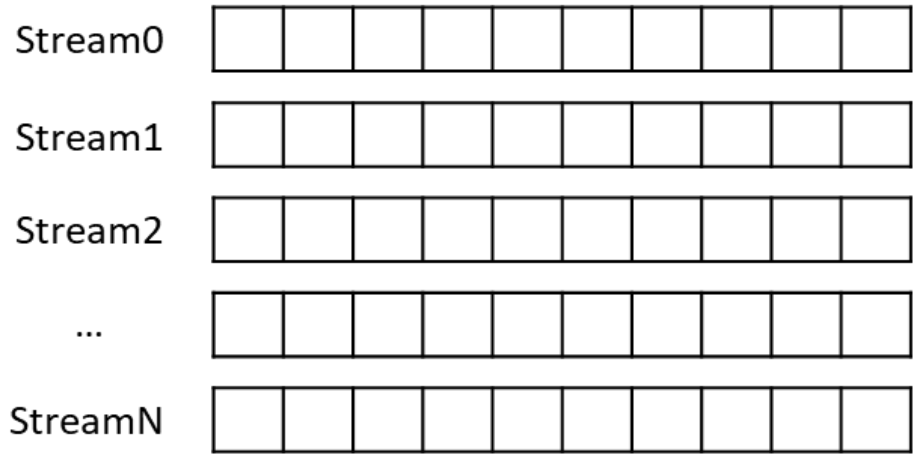



これは、あまり実用的ではありません。モーション検索はスライス境界を検知できないため、スライス数が増加すると全体の圧縮品質に影響を与えます。スライスには追加のヘッダーバイトも必要になります。

単一のストリームを非同期で処理することができますが、スケーラブルではありません。フレーム間の依存関係が並列処理を妨げていますが、依存関係をオフにすると特定のビットレートで品質が低下します。処理中にフレーム数を増やすとレイテンシーも増加します。




品質を維持しながらレイテンシーを軽減して効率良くスケーリングする最良の方法は、複数のストリームを同時に処理することです (注: 非リアルタイム処理では、フレームがセグメント/GOP の境界を越えて存在しないため、単一のストリームでもセグメントとして並列に処理できます)。



このアプローチは、アクセラレーター間でスケールする理想的な並列性を実現します。ストリーム間に依存関係がないため、各アクセラレーターは調整なしに可能な限り高速に処理を行うことができます。ハイパーエンコードの場合、iGPU と dGPU に個別のストリームをスケジュールするほうが高速です。

oneAPI の観点から見ると、これらのプロパティーは SYCL* との相互運用性を大幅に簡略化します。メディア操作は、通常カーネル内部では実行されないため、API や開発レベルでの考慮事項は少なくなります。メディア操作は、カーネルにデータを供給する (ソースとして動作)、またはソースが供給するデータのシンクとして動作します。パフォーマンスに関する懸念事項は、メディア操作とカーネル間のハンドオフが同期につながり、単一ストリーム内で非同期に処理する可能性が減少することです。そのため、複数のストリームを同時に処理することが、この問題に対する最善の解決策となります。

### メディア操作の最適化

アルゴリズムはハードウェアで実装されるため、メディア開発における主な懸念事項はデータの局所性、同期、そしてハードウェアを常にビジーにするワークのパイプラインを提供することです。

**データの局所性:** フレームを GPU 上に維持し、不要な CPU へのコピーを避けます。メディアエンジンは、GPU のメモリー階層に接続されているため、データはスライスとメディアエンジンのコンポーネント間でローカルに共有できます。GPGPU の観点からは、これらの操作はローカル GPU のデータで動作します。フレームは、このハードウェアとエグゼキューション・ユニット間で低レイテンシー/コピーなしで共有されます。これは、ディスクリート GPU ではさらに重要であり、PCI バスで生フレームを移動するコストを軽減できます。

**同期:** 複数のハードウェア・ユニットは独立して機能するため非同期に動作することができます。最高のパフォーマンスを達成するには、アプリケーションは可能な限り頻繁に CPU との同期を強制する必要があります。アクセラレーターが中断することなく処理を続行できるようにアルゴリズムを設計します。

**ハードウェアをビジーに保つ:** 命令キャッシュがフルに維持されていないと、エンジンのクロックは低下します。フルクロックに復帰するには数ミリ秒を要します。

## メディア・パイプラインの相互運用とメモリー共有

メディアエンジンの機能は、次のような低レベルの OS 固有のインターフェイスで使用されます。

- Linux* 向けの VA-API (ビデオ・アクセラレーション API)
- Windows* 向けの Microsoft* DirectX* ビデオ・アクセラレーション

上記のような低レベルのインターフェイス上に構築される各種高レベル・メディア・フレームワークも同様です。

- インテル® ビデオ・プロセシング・ライブラリー (インテル® VPL)
- FFmpeg* および libav*
- GStreamer

それぞれのメディア・フレームワークは、デバイスとコンテキストの作成、メモリー割り当て、およびタスク投入の独自インターフェイスを定義しています。また、大部分のフレームワークでは、メモリー・オブジェクトをほかのメモリーハンドルに変換するエクスポート/インポート・インターフェイスも用意されています。

- 高レベル・メディア・フレームワーク (FFmpeg*、GStreamer) は、低レベル・メディア・ハンドル (VA-API、DirectX* サーフェス) 間との変換をサポートしています。

- 低レベル・メディア・インターフェイス (VA-API、DirectX*) は、Linux* のバッファーや Windows* の NT ハンドルのような OS 固有の汎用 GPU メモリーハンドルへの変換をサポートしています。
- DMA バッファー/NT ハンドルと USM デバイスポインター間のレベルゼロサポート変換。

これらのインターフェイスを組み合わせると、メディア・フレームワークを介して送信されたメディア操作と、SYCL* キューに送信された SYCL* + 計算カーネル間で、ゼロコピーのメモリー共有が可能になります。これには、メディア・フレームワークと同一 GPU デバイス上に作成された SYCL* キューと SYCL* デバイスが、レベルゼロのバックエンド (OpenCL* バックエンドではなく) を使用すると仮定します。

メモリーハンドルの変換は、複数のステージ (FFmpeg*/GStreamer、VA-API/DirectX*、DNA/NT、レベルゼロ、SYCL*) で行われますが、変換されたすべてのメモリーハンドルは同じ物理メモリーブロックを参照します。そのため、あるメモリーハンドルにデータを書き込むと、ほかのすべてのメモリーハンドルでそのデータを利用できます (書き込みと読み取りの同期が適切に行われていることが前提)。

以下に、メディア・フレームワークと SYCL* 間でゼロコピーバッファーを共有する際に使用されるインターフェイスを示します。

1. (Linux*) VA-API から DMA-BUF (英語)
2. (Windows*) DirectX から NT ハンドル (英語)
3. DMA-BUF または NT ハンドルからレベルゼロ (英語)

DMA-BUF または NT ハンドルからレベルゼロが作成するメモリーポインター (上記 3) は、メディアメモリーの割り当てとメディア操作で使用される同一 GPU デバイス上で動作する SYCL* カーネルからのみアクセス可能な USM デバイスポインターです。この USM ポインターは、ホストからはアクセスできず、CPU やほかの XPU デバイス上で動作する SYCL* カーネルからもアクセスできません。

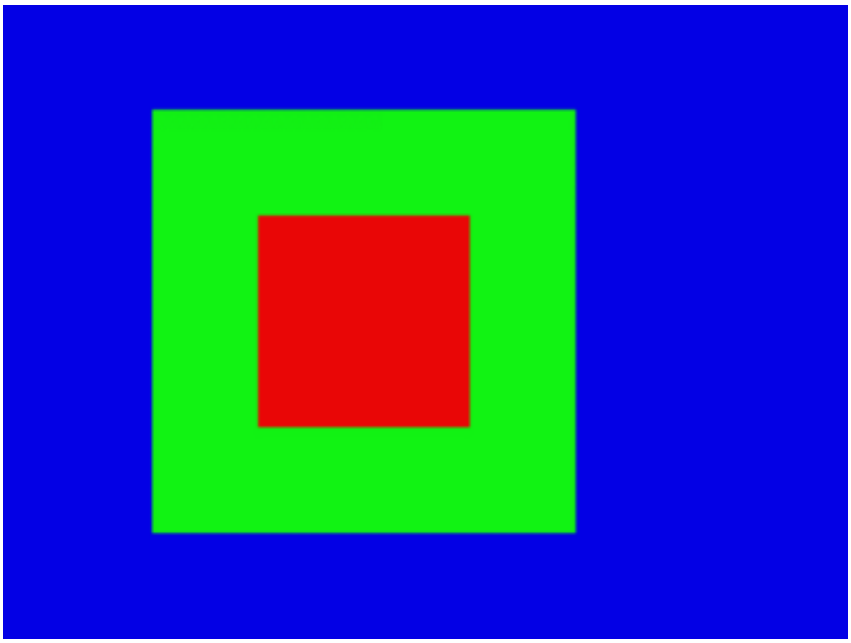
次の項の例は、上記の 1 と 3 のインターフェイス、および合成ビデオデータ (移動する矩形) を使用して、VA-API と SYCL* 間でゼロコピーバッファーの共有を実証しています。GPU メディアエンジンの FFmpeg*/GStream ビデオデコード/エンコードと GPU 計算エンジン上の SYCL* カーネルを使用したさらに高度な例については、インテル® DL Streamer メモリー相互運用 API (プレビュー) (英語) とインテル® DL Streamer のサンプル (英語) を参照してください。

## VA-API と SYCL* メモリー共有の例

この節では次の例を紹介します。

1. NUM_FRAMES フレームに対して、共有 VA-API サーフェスと USM デバイスポインターを割り当てます。
2. フレーム上で移動する矩形を描画する VA-API 呼び出しを送信します。
3. ステップ 2 で VA-API が作成した矩形の内側にサブ矩形を描画するため SYCL* カーネルを送信します。
4. すべてのフレームを同期し、RGB データをファイルに書き込みます。

この例で生成される出力フレームは次のようになります。



この例は、Linux* 向けに作成されており、oneAPI パッケージのほかに以下に示す追加のパッケージをインストールする必要があります (Ubuntsu* で apt パッケージ・マネージャーを使用してインストールする例)。

```
$ sudo apt install intel-level-zero-gpu level-zero-dev
$ sudo apt install intel-media-va-driver-non-free libva-dev libva-drm2
```

レベルゼロと VA-API ライブラリーの連携が必要です。

```
$ icpx -fsycl memory-sharing-with-media.cpp -lze_loader -lva -lva-drm
```

実行例では、いくつかのメディアプレーヤー (ex、ffplay など) で直接再生したり、次のように FFmpeg* コマンドを使用して圧縮ビデオ・フォーマットに符号変換できるファイル (`output.bgra`) を生成しています。

```
$ FFmpeg* -f rawvideo -pix_fmt bgra -s 320x240 -i output.bgra output.mp4
```

生成されたファイル (`output.mp4`) は、任意のメディアプレーヤーで再生できます。

```
$ ffplay output.mp4
```

```
1.  // SYCL*
2.  #include <CL/sycl.hpp>
3.
4.  // SYCL* oneAPI 拡張
5.  #include <sycl/ext/oneapi/backend/level_zero.hpp>
6.
7.  // レベルゼロ
8.  #include <level_zero/ze_api.h>
9.
10. // VA-API
11. #include <va/va_drm.h>
12. #include <va/va_drmcommon.h>
13.
14. #include <cstdio>
15. #include <fcntl.h>
16. #include <unistd.h>
```

```
#include <vector>

#define OUTPUT_FILE "output.bgra"

#define VAAPI_DEVICE "/dev/dri/renderD128"

#define FRAME_WIDTH 320
#define FRAME_HEIGHT 240

#define RECT_WIDTH 160
#define RECT_HEIGHT 160
#define RECT_Y (FRAME_HEIGHT - RECT_HEIGHT) / 2

#define NUM_FRAMES (FRAME_WIDTH - RECT_WIDTH)

#define VA_FORMAT VA_FOURCC_BGRA
#define RED 0xffff0000
#define GREEN 0xff00ff00
#define BLUE 0xff0000ff

#define CHECK_STS(_FUNC)
  {
    auto _sts = _FUNC;
    if (_sts != 0) {
      printf("Error %d calling " #_FUNC, (int)_sts);
      return -1;
    }
  }

VASurfaceID alloc_va_surface(VADisplay va_display, int width, int height) {
  VASurfaceID va_surface;
  VASurfaceAttrib surface_attrib{};
  surface_attrib.type = VASurfaceAttribPixelFormat;
  surface_attrib.flags = VA_SURFACE_ATTRIB_SETTABLE;
  surface_attrib.value.type = VAGenericValueTypeInteger;
  surface_attrib.value.value.i = VA_FORMAT;
  vaCreateSurfaces(va_display, VA_RT_FORMAT_RGB32, width, height, &va_surface,
                   1, &surface_attrib, 1);
  return va_surface;
}

int main() {
  // GPU デバイスとレベルゼロ·バックエンドで SYCL* キューを作成し、
  // レベルゼロ·コンテキストとデバイスを照会
  sycl::queue sycl_queue{sycl::ext::oneapi::filter_selector(
      "level_zero")}; // { sycl::gpu_selector() }
  auto ze_context = sycl::get_native<sycl::backend::ext_oneapi_level_zero>(
      sycl_queue.get_context());
  auto ze_device = sycl::get_native<sycl::backend::ext_oneapi_level_zero>(
      sycl_queue.get_device());

  // VA-API コンテキストの作成 (VADisplay)
  VADisplay va_display = vaGetDisplayDRM(open(VAAPI_DEVICE, O_RDWR));
  if (!va_display) {
    printf("Error creating VADisplay on device %s\n", VAAPI_DEVICE);
    return -1;
  }
  int major = 0, minor = 0;
  CHECK_STS(vaInitialize(va_display, &major, &minor));

  // VA-API サーフェスの作成
  VASurfaceID surfaces[NUM_FRAMES];
  for (int i = 0; i < NUM_FRAMES; i++) {
    surfaces[i] = alloc_va_surface(va_display, FRAME_WIDTH, FRAME_HEIGHT);
  }

  // 各 VA-API サーフェスを USM デバイスポインターに変換
```

```
84.   // (VA-API とレベルゼロ間で共有されるゼロコピーバッファー)
85.   void *device_ptr[NUM_FRAMES];
86.   size_t stride;
87.   for (int i = 0; i < NUM_FRAMES; i++) {
88.     // VASurface から DMA-FD をエクスポート
89.     VADRMPRIMESurfaceDescriptor prime_desc{};
90.     CHECK_STS(vaExportSurfaceHandle(va_display, surfaces[i],
91.                                     VA_SURFACE_ATTRIB_MEM_TYPE_DRM_PRIME_2,
92.                                     VA_EXPORT_SURFACE_READ_WRITE, &prime_desc));
93.     auto dma_fd = prime_desc.objects->fd;
94.     auto dma_size = prime_desc.objects->size;
95.     stride = prime_desc.layers[0].pitch[0] / sizeof(uint32_t);
96.
97.     // DMA-FD をレベルゼロ･デバイス･ポインターにインポート
98.     ze_external_memory_import_fd_t import_fd = {
99.         ZE_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMPORT_FD,
100.        nullptr, // pNext
101.        ZE_EXTERNAL_MEMORY_TYPE_FLAG_DMA_BUF, dma_fd};
102.    ze_device_mem_alloc_desc_t alloc_desc = {};
103.    alloc_desc.stype = ZE_STRUCTURE_TYPE_DEVICE_MEM_ALLOC_DESC;
104.    alloc_desc.pNext = &import_fd;
105.    CHECK_STS(zeMemAllocDevice(ze_context, &alloc_desc, dma_size, 1, ze_device,
106.                               &device_ptr[i]));
107.
108.    // DMA-FD をクローズ
109.    close(dma_fd);
110.  }
111.
112.  // サイズ 1 x 1 の VA-API サーフェスを作成し､GREEN のピクセルを書き込み
113.  VASurfaceID surface1x1 = alloc_va_surface(va_display, 1, 1);
114.  VAImage va_image;
115.  void *data = nullptr;
116.  CHECK_STS(vaDeriveImage(va_display, surface1x1, &va_image));
117.  CHECK_STS(vaMapBuffer(va_display, va_image.buf, &data));
118.  *(uint32_t *)data = GREEN;
119.  CHECK_STS(vaUnmapBuffer(va_display, va_image.buf));
120.  CHECK_STS(vaDestroyImage(va_display, va_image.image_id));
121.
122.  // 背景を BLUE に塗りつぶし､1 x 1 サーフェスを移動する
123.  // GREEN の矩形にスケールアップする VA-API 呼び出し
124.  VAConfigID va_config_id;
125.  VAContextID va_context_id;
126.  CHECK_STS(vaCreateConfig(va_display, VAProfileNone, VAEntrypointVideoProc,
127.                           nullptr, 0, &va_config_id));
128.  CHECK_STS(vaCreateContext(va_display, va_config_id, 0, 0, VA_PROGRESSIVE,
129.                            nullptr, 0, &va_context_id));
130.  for (int i = 0; i < NUM_FRAMES; i++) {
131.    VAProcPipelineParameterBuffer param{};
132.    param.output_background_color = BLUE;
133.    param.surface = surface1x1;
134.    VARectangle output_region = {int16_t(i), RECT_Y, RECT_WIDTH, RECT_HEIGHT};
135.    param.output_region = &output_region;
136.    VABufferID param_buf;
137.    CHECK_STS(vaCreateBuffer(va_display, va_context_id,
138.                             VAProcPipelineParameterBufferType, sizeof(param),
139.                             1, &param, &param_buf));
140.    CHECK_STS(vaBeginPicture(va_display, va_context_id, surfaces[i]));
141.    CHECK_STS(vaRenderPicture(va_display, va_context_id, &param_buf, 1));
142.    CHECK_STS(vaEndPicture(va_display, va_context_id));
143.    CHECK_STS(vaDestroyBuffer(va_display, param_buf));
144.  }
145.
146.#if 0
147.    // Linux* では､i915 KMD ドライバーがインテルのメディアおよび計算ドライバーから
148.    // 送信されたリード/ライトを同期するため､同期はオプションです
```

```
149.    for (int i = 0; i < NUM_FRAMES; i++) {
150.        CHECK_STS(vaSyncSurface(va_display, surfaces[i]));
151.    }
152.#endif
153.
154.  // SYCL* カーネルを送信して、GREEN の矩形に RED のサブの矩形を書き込み
155.  std::vector<sycl::event> sycl_events(NUM_FRAMES);
156.  for (int i = 0; i < NUM_FRAMES; i++) {
157.    uint32_t *ptr = (uint32_t *)device_ptr[i] +
158.                    (RECT_Y + RECT_HEIGHT / 4) * stride + (i + RECT_WIDTH / 4);
159.    sycl_events[i] = sycl_queue.parallel_for(
160.        sycl::range<2>(RECT_HEIGHT / 2, RECT_WIDTH / 2), [=](sycl::id<2> idx) {
161.          auto y = idx.get(0);
162.          auto x = idx.get(1);
163.          ptr[y * stride + x] = RED;
164.        });
165.  }
166.
167.  // すべての SYCL* カーネルを同期
168.  sycl::event::wait(sycl_events);
169.
170.  // VA-API サーフェスをシステムメモリーにマップしてファイルに書き込み
171.  FILE *file = fopen(OUTPUT_FILE, "wb");
172.  if (!file) {
173.    printf("Error creating file %s\n", OUTPUT_FILE);
174.    return -1;
175.  }
176.  for (int i = 0; i < NUM_FRAMES; i++) {
177.    CHECK_STS(vaDeriveImage(va_display, surfaces[i], &va_image));
178.    CHECK_STS(vaMapBuffer(va_display, va_image.buf, &data));
179.    fwrite(data, 1, FRAME_HEIGHT * FRAME_WIDTH * 4, file);
180.    CHECK_STS(vaUnmapBuffer(va_display, va_image.buf));
181.    CHECK_STS(vaDestroyImage(va_display, va_image.image_id));
182.  }
183.  fclose(file);
184.  printf("Created file %s\n", OUTPUT_FILE);
185.
186.  // デバイスポインターと VA-API サーフェスを解放
187.  for (int i = 0; i < NUM_FRAMES; i++)
188.    zeMemFree(ze_context, device_ptr[i]);
189.  vaDestroySurfaces(va_display, surfaces, NUM_FRAMES);
190.
191.  return 0;
192.}
```

## SYCL* によるぼかし (Blur) の例

これから概念を組み合わせたインテル® ビデオ・プロセシング・ライブラリー (インテル® VPL) の例 (現在は単一ストリームのみ) については、`sycl-blur` を参照してください。このサンプルでは、次の手順でフレームを入力、操作、出力するビデオ API とインテル® VPL 間とのメモリー相互運用を示します。

- SYCL* を設定
- インテル® VPL のセッションを設定
- インテル® VPL VPP を初期化
- フレームで以下を繰り返し
  - フレームをファイルから読み取り
  - GPU 上で VPP リサイズ/カラー空間変換を実行
  - GPU サーフェスへのアクセス権を取得して USM へ変換
  - GPU で SYCL* カーネル (blur) を実行
  - フレームをファイルへ出力

このサンプルは、https://github.com/oneapi-src/oneVPL/tree/master/examples/interop/dpcpp-blur (英語) から入手できます。

この例からインテル® VPL と SYCL* 間の相互作用は、フレームレベルであることが分かります。インテル® VPL がフレームを供給して SYCL* カーネルがそれを処理します。L0 (Linux*) でゼロコピー機能が有効な OS 環境では、libva* のフレームデータは USM として SYCL* に供給されます。libva* の生フレームを新しい USM サーフェスにコピーする代わりに、アプリケーションは GPU 上のフレームを libva* サーフェスとして処理し、USM の場合と同じようにメモリーで操作を開始できます。

この例はさらに簡単にできますが、多くの設計を簡素化し、ゼロコピーの利点を十分に紹介できないのが残念です。

- 生フレームをストレージから読み取り、ストレージに書き込む。これは全体のフレームレートに影響します。
- VPP データは、システムメモリーとして読み取られ、ビデオメモリーに変換されます。
- パイプラインは各フレーム間で同期されます。

ゼロコピーは、ハイパフォーマンスなアプリケーションの核となる概念です。

# インテル® グラフィックス・パフォーマンス・アナライザーによるパフォーマンス解析

## 概要

### インテル® グラフィックス・パフォーマンス・アナライザー (インテル® GPA)

インテル® GPA は、シングル/マルチ CPU プラットフォーム、およびシングル/マルチ GPU プラットフォームで実行されるアプリケーションを解析するパフォーマンス解析ツールです。GUI とコマンドスクリプトの両方でデータを詳しく解析できます。

インテル® GPA は、4 つの GUI ツールとコマンドライン・ツールで構成されます。

- GUI ツール
  - Graphics Monitor - インテル® GPA ツールのハブであり、トレースおよびフレーム・キャプチャーを開始できます。
  - System Analyzer - CPU と GPU のアクティビティーをランタイムに解析します。
  - Graphics Trace Analyzer - 数秒間のデータ・キャプチャーで、CPU/GPU 間の相互作用を詳しく解析できます。
  - Graphics Frame Analyzer - 単独または複数のフレームで GPU のアクティビティーを詳しく解析できます。

- コマンドライン・インターフェイス・ツール
  - インテル® GPA Framework - スクリプト化が可能なコマンドライン・インターフェイスにより、単独または複数のフレームから広範囲なデータを取得して解析できます。

この章では、GPU の動作を詳しく理解するため、Graphics Frame Analyzer の機能に注目します。

(注: インテル® GPA Framework は、Graphics Frame Analyzer のバックエンドであり、同じデータを取得することができます。また、インテル® GPA Framework を使用すると、プロファイル作業を自動化できます。)

## Graphics Frame Analyzer の機能

Graphics Frame Analyzer が提供する便利な機能の一部を紹介します。

- 一連のフレームのうち、どのフレームに最も時間がかかるか即座に確認できます。
- 高度なプロファイル・モードを使用すると、パイプラインの状態と GPU メトリックからアプリケーションで最もホットなボトルネックを自動的に計算するためあるフレームに対して最適化を行うと、アプリケーションのほかの部分も最適化される可能性があります。
- ジオメトリー - ワイヤーフレームとソリッド。
  - 画面外に描画したか数秒で確認できます。
  - ジオメトリーをあらゆる角度から動的に観察できます。
- テクスチャー
  - 描画呼び出しごとにテクスチャーを可視化できます。
  - 持続時間が長い描画呼び出しが、フレーム内の重要ではない部分を描画していないか確認します。
- シェーダー
  - 最も時間を要するシェーダー・コマンドラインを特定します。
  - 各 DXR (DirectX* レイトレーシング) シェーダーの呼び出し回数や、シェーダーのベクトルエンジンの占有率などを観察できます。
- レンダリング状態の確認 - ボタンをクリックするだけで、テクスチャーやピクセル、イベントを無効にしてボトルネックの原因を特定し、メトリックやその他のデータの変化をすぐに確認できます。

サポートされる API

- Direct3D* 11
- Direct3D* 12 (DirectX* 12 Ultimate を含む)
- Vulkan*

## エグゼキューション・ユニットのストール、アクティブ、スループット

Graphics Frame Analyzer は、フレームの持続期間、API 呼び出し、リソースデータ、イベントデータなどを素早く確認できる、初心者から上級者まで利用可能な強力なツールです。各データ要素が意味することの理解度を深めることで、パフォーマンスの問題の根本的な原因を特定しやすくなります。

## 実行ストール、実行アクティブ、実行スループット

これら 3 つのデータ要素の相互関係を解釈する方法を知ることは、GPU に関連するアプリケーションの相互作用をさらに深く理解することにつながります。

## EU、XVE (Xe ベクトルエンジン) およびインテル® XMX

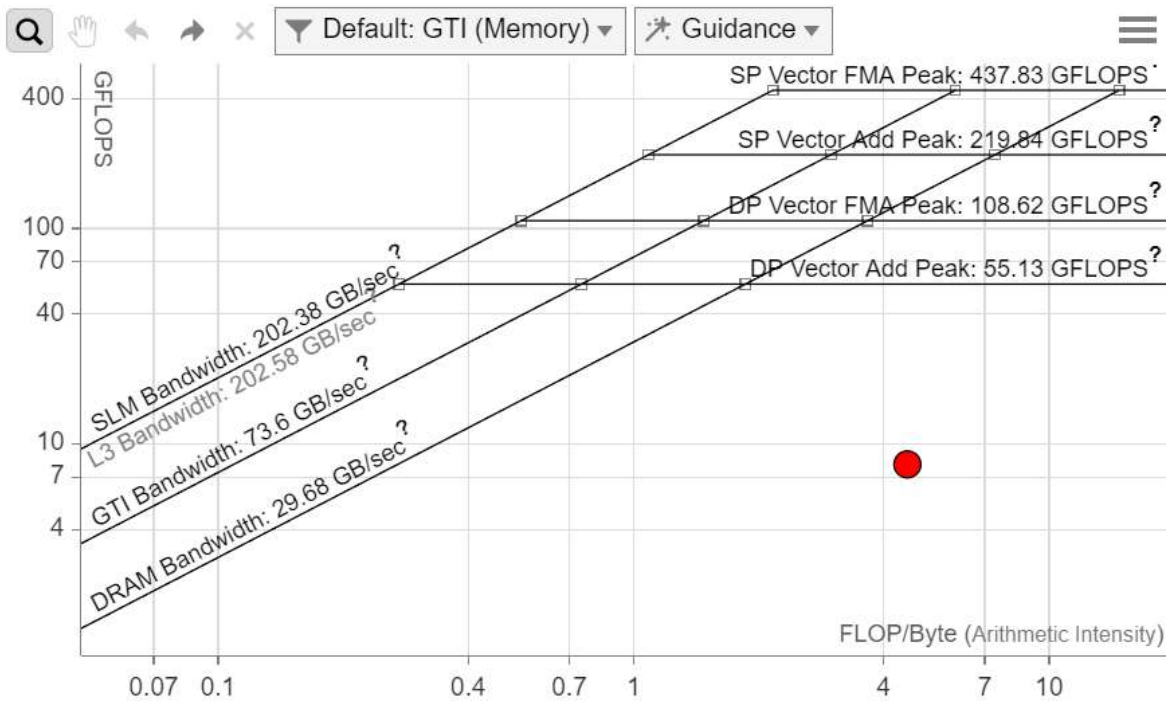
このドキュメントの「インテル® Xe GPU アーキテクチャー」の項で説明したように、Xe LP および前世代のインテル® GPU では、EU (エグゼキューション・ユニット) が GPU の計算ユニットとなります。Xe HPG と Xe HPC では、計算ユニットに Xe コアを導入しています。後者のプラットフォームでは、各 Xe コアは ALU (算術論理演算ユニット) — 8 つのベクトルエンジン (XVE) と 8 つの行列拡張 (XMX) — で構成されます。

Graphics Frame Analyzer が Xe LP またはそれ以前のアーキテクチャーで実行される場合、EU アクティブ、EU ストール、および EU スループットのデータラベルが表示されます。新しいアーキテクチャーでは、XVE アクティブ、XVE ストール、および XVE スループットのデータラベルが表示されます。ここでは、リファレンス・アーキテクチャーとして Xe LP を使用するため、EU を表記します。しかし、EU であろうが XVE であろうが、ストール/アクティブ/スループットの関係は同じようにパフォーマンスに影響することを忘れないでください。

## 3 つのデータ要素を理解する

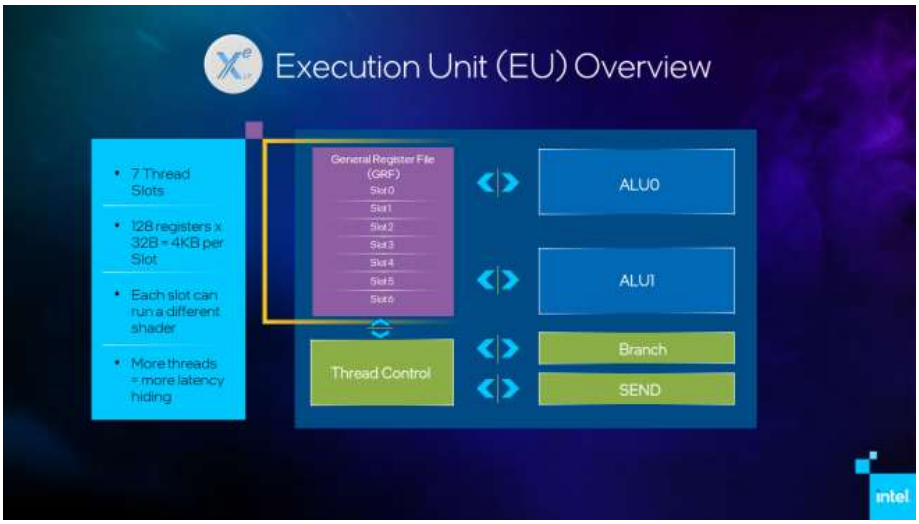
始めに、96 個の EU を持つ Xe LP のユニット全体から、1 つの EU にドリルダウンして見ていきます。この GPU の各 EU の汎用レジスターファイル (GRF) は、7 つのスレッドを保持します。次の図は、GPU 全体から単一の EU へのドリルダウンを示しています。

### 単一の EU にドリルダウン



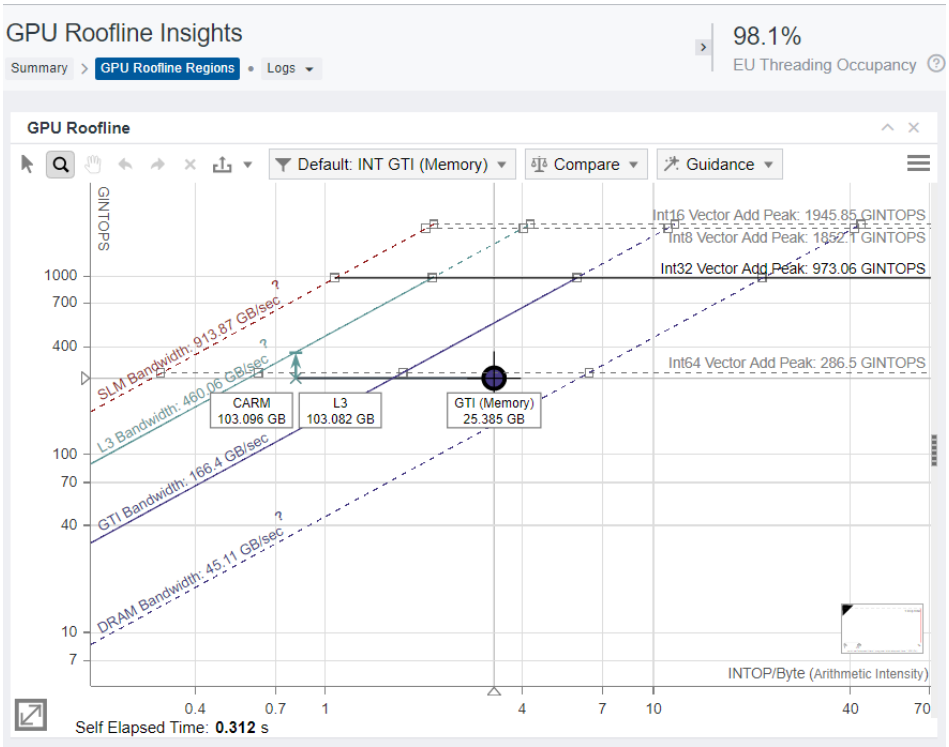

EU を詳しく見ていきます。次の図に示される要素は、主に GRF の 7 つのスレッドスロットに焦点を当て、SEND ユニットおよび 2 つの ALU とスレッドの相互作用の重要性を示しています。

### GRF、ALU、スレッド制御 (Thread Control)、分岐 (Branch)、および送信 (SEND) ユニット




スレッド化のシナリオについて考えてみます。次の図に GRF の内容を示します。この EU の GRF には、この時点で 1 つのスレッドのみがロードされていることが分かります。つまり、単一スレッドが一定時間実行されていることを示しています。これは、命令を実行するため ALU が 1 つ、または 2 つ呼び出されることを意味します。ある時点で、スレッドはデータをリードまたはライトする必要があり、これにより SEND ユニットがアクティブになります。スレッドがリード/ライトの完了を待機する間、EU はストールします。スレッドはロードされていますが、何も計算を行いません。

### GRF には単一のスレッドがある




このシナリオを拡張した次の図では、EU に 2 つ目のスレッドがあることが分かります。EU の GRF に 2 つ目のスレッドがロードされている場合、最初のスレッドが SEND ユニットを呼び出した時点で実行を停止する代わりに、EU は 2 つ目のスレッドの命令を実行します。2 つ目のスレッドが SEND ユニットを使用するコマンドを呼び出すと、最初のスレッドが実行可能になるまで EU はストールします。この状況で EU に 3 つ目のスレッドがある、または最初のスレッドが SEND 処理を終え実行可能である場合、すべての待機時間が隠匿されスレッドには待機が生じる可能性がありますが、EU はストールすることはないでしょう。

### EU がデータのリード/ライトを待機する間 ALU は何も行わない (命令を実行しない)

GPU

| Compute Task | CARM (EU <-> Data Port) | | EU Array | | | EU Threading Occupancy |
|---|---|---|---|---|---|---|
| | Total, GB | L3 Cache Line Utilization | Active | Stalled | Idle | |
| [Outside any task] | 0.000 | 0.0% | 0.2% | 0.3% | 99.6% | 0.4% |
| zeCommandListAppendMemoryCopyReg | 0.000 | 0.0% | 21.5% | 34.3% | 44.2% | 39.6% |
| zeCommandListAppendBarrier | 0.000 | 0.0% | 0.0% | 0.0% | 100.0% | 0.0% |
| matrixMultiply2<int, (unsigned long)2048> | 103.096 | 100.0% | 47.8% | 52.2% | 0.0% | 98.3% |

## 用語

以下の定義において、計算された結果データは、

- すべての EU の平均値を示します。
- フレーム全体のデータと、フレームの選択された部分のデータの両方で構成されます。選択されるのは、1 つの呼び出しでも、呼び出しのセットでも、フレームのセットでもかまいません。

**アイドル**

EU アイドルは、スレッドがロードされていない時間のパーセンテージです。

**アクティブ**

EU アクティブは、ALU0 または ALU1 が何らかの命令を実行していた時間のパーセンテージです。この値が低い原因は、ストールが多いか EU がアイドル状態であることです。

**ストール**

EU ストールは、1 つ以上のスレッドがロードされているにもかかわらず、リードまたはライトを待機しているため、どのスレッドも実行できない時間のパーセンテージを示します。

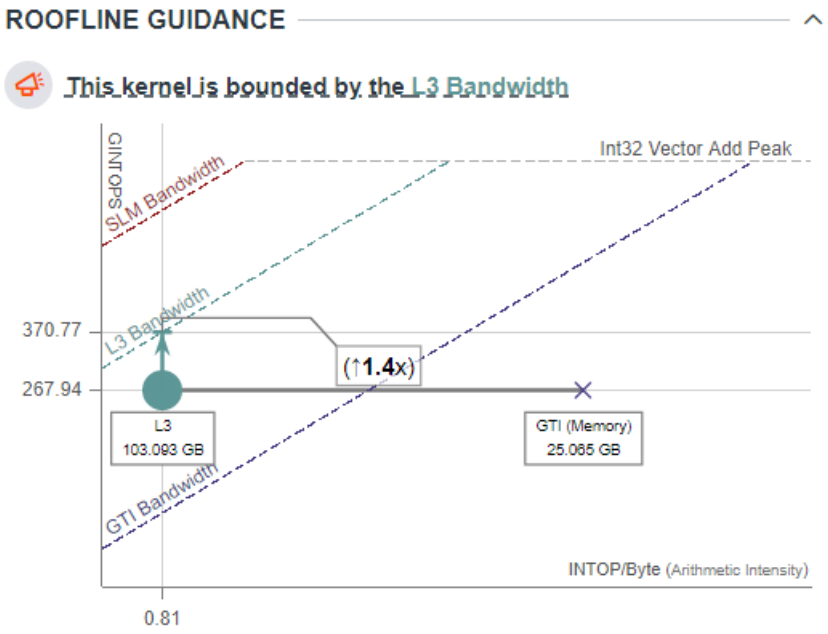
**スレッド占有率**

EU スレッドの占有率は、GRF スロットを占有した (スレッド負荷)パーセンテージです。これは可能な限り高い値にする必要があります。EU スレッドの占有率が低い場合、バーテックス・フェッチやスレッドのディスパッチなど先行する H/W ブロックのボトルネック、または計算シェーダーである場合は適切でない SIMD 幅や共有メモリーの利用を示します。

96 個の EU がそれぞれ 1 つのスレッドを実行すると、7 スロット/EU のうち 1 スロットが占有され、スレッドの占有率は 1/7 (14%) になります。

2 つの EU にスレッドがロードされておらず (アイドル状態) で、ほかの 94 の EU に 6 つのスレッドがロードされていると仮定すると、占有率 = (0 + 0 + 6 * 94) / 672 = 84% になります。

Graphics Frame Analyzer に表示されるスレッドの占有率 (Thread Occupancy) 値は、選択した時間範囲におけるすべての EU の平均占有率を示しています。ほかのハードウェアでは、EU や XVE の数が異なるかもしれませんが、計算はすべてのエグゼキューション・ユニットで行われます。例えば、次の図の左に表示されるフレームには、フレーム全体の時間 (6ms) でスレッド占有率は変化していますが、96 個すべての EU の時間平均は 77% となっています。また、フレーム期間中の一部のスレッド占有率も確認できます。図の右側では、最も時間がかかる 3 つの描画呼び出しを選択して、アプリケーションのこれらのビットを実行するのに要した 1.9 ミリ秒間のスレッド占有率が 85.3% であることが確認できます。

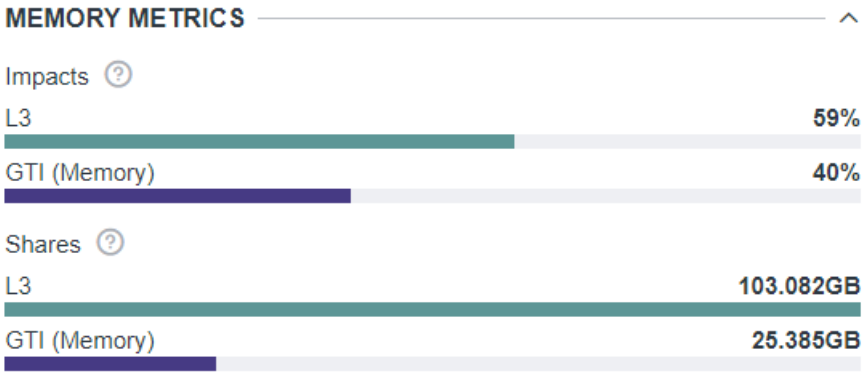



## Graphics Frame Analyzer

このデータをインテル® GPA の Graphics Frame Analyzer で確認します。使い方を説明する 8 本のビデオ (英語)と記事が用意されています。

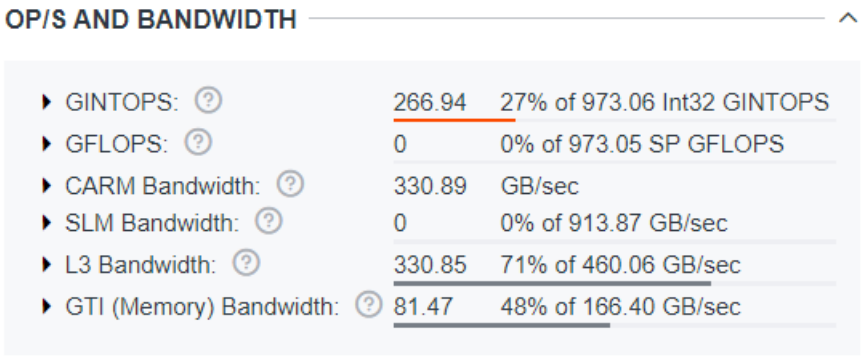
フレームを開いた直後の Graphics Frame Analyzer では、次の図のような表示を確認できます。フレームを開くとフレーム全体のデータのパーセンテージ値が表示されます。これは、アクティブ、ストール、スループットなどのデータがフレーム時間で 96 EU を平均したパーセンテージを意味します。

### データ値はすべてのフレーム時間における全 EU の平均値



その後、単独の描画呼び出しまたは呼び出しセットを選択すると、選択したフレームに応じてデータが再計算されます。

**選択後 (この場合 91、94、95 の呼び出し)、これらの呼び出しデータのみを表示するよう再計算**




さらに、ストリームをキャプチャーするとマルチフレーム表示で開きます。また、単一のフレームや複数のフレームを選択できます。複数のフレームを選択した場合、再計算されるデータは選択されたすべてのフレームデータの合計になります。

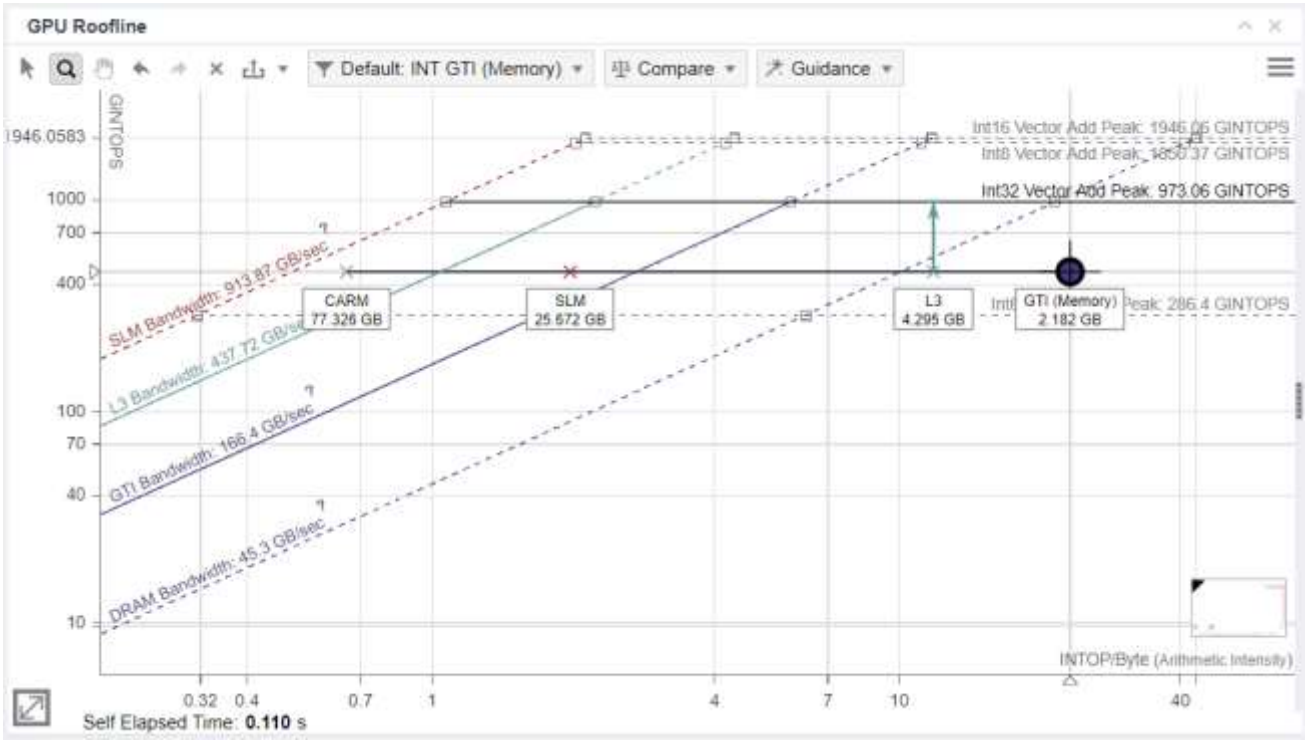
効率良くプロファイルを行うには、GPU がどのように動作してメトリックが何を意味しているか理解することが重要ですが、問題のタイプを知るためフレーム内の描画呼び出しを手動で解析する必要はありません。そのような状況での解析を支援するため、インテル® GPA は自動ホットスポット解析 (高度なプロファイル・モード) を提供します。

## ホットスポット解析

EU アーキテクチャーをある程度理解できたら、その知識をどのようにプロファイルに活用できるか見ていきましょう。

高度なプロファイル・モードを有効にすると、Graphics Frame Analyzer はボトルネックのタイプとパイプラインの状態によってボトルネックを特定します。このように分類することで、ある問題を修正することで、その部分的な問題だけではなく問題のカテゴリー全体が影響を受けることが多いという利点もあります。

Graphics Frame Analyzer ツールの左上にあるボタンをクリックして、ホットスポット解析を有効にします (以下の図を参照)。上部のバーチャートにボトルネックが表示され、左ペインの **[API Log]** がボトルネックのタイプを示すように変化します。ボトルネックをクリックすると、メトリックビューにボトルネックの詳細が表示されます (メトリックの説明、ボトルネックを軽減するヒントなど)。

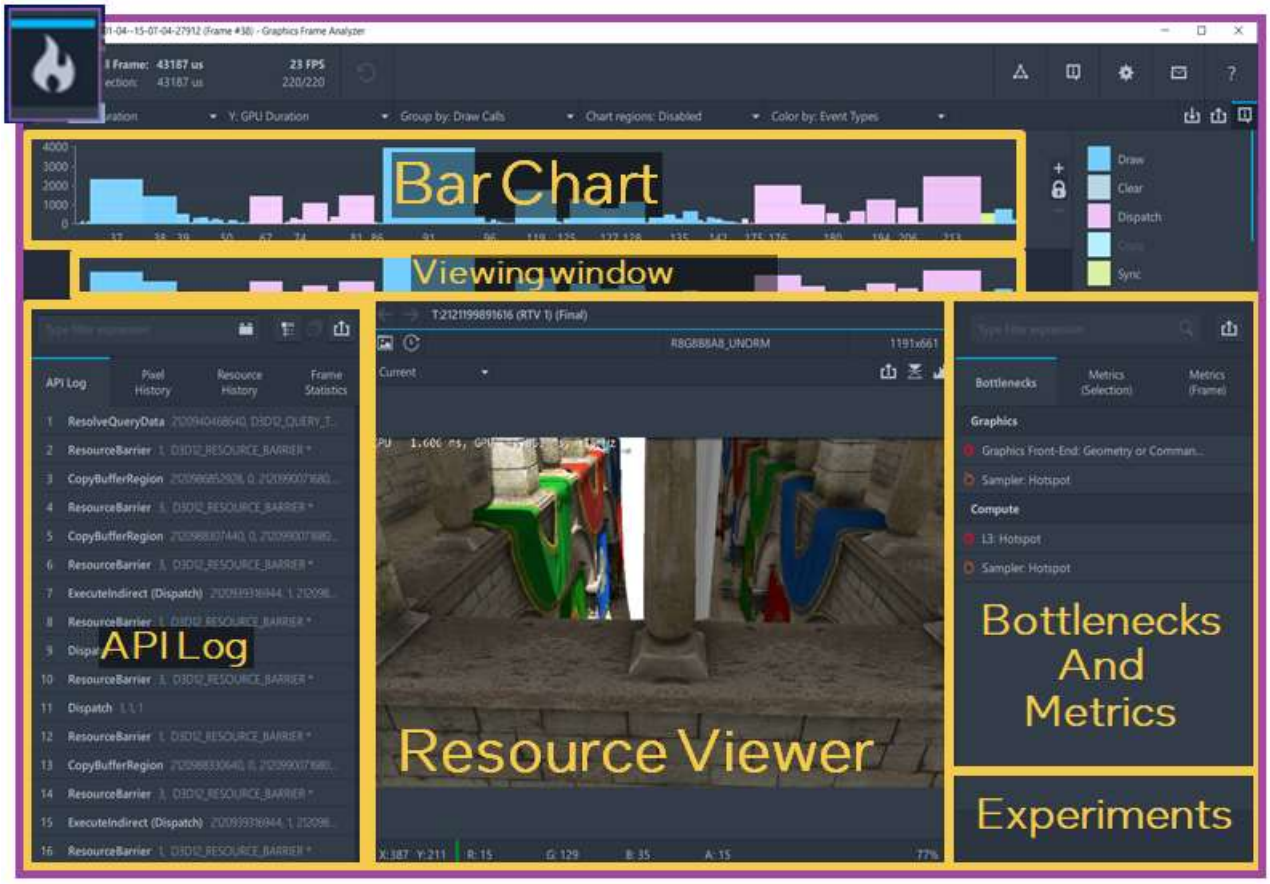



**ホットスポット: L2 キャッシュ**

**L2 キャッシュ・ホットスポットの特性**

アプリケーションのスレッド占有率が高く 90% 近くある場合は問題ありません。しかし、スレッド占有率が高く、5～10% 以上のストールがある場合は、L2 キャッシュがボトルネックになっている可能性があります。

Graphics Frame Analyzer でフレームを開き、右にある **[Metrics Viewer]** パネルをご覧ください。占有率は 90% 以上ですが、EU にはまだストールが残っています。これは EU スレッドがメモリーからのデータを待機していることを意味します。




**Shader Profiler**

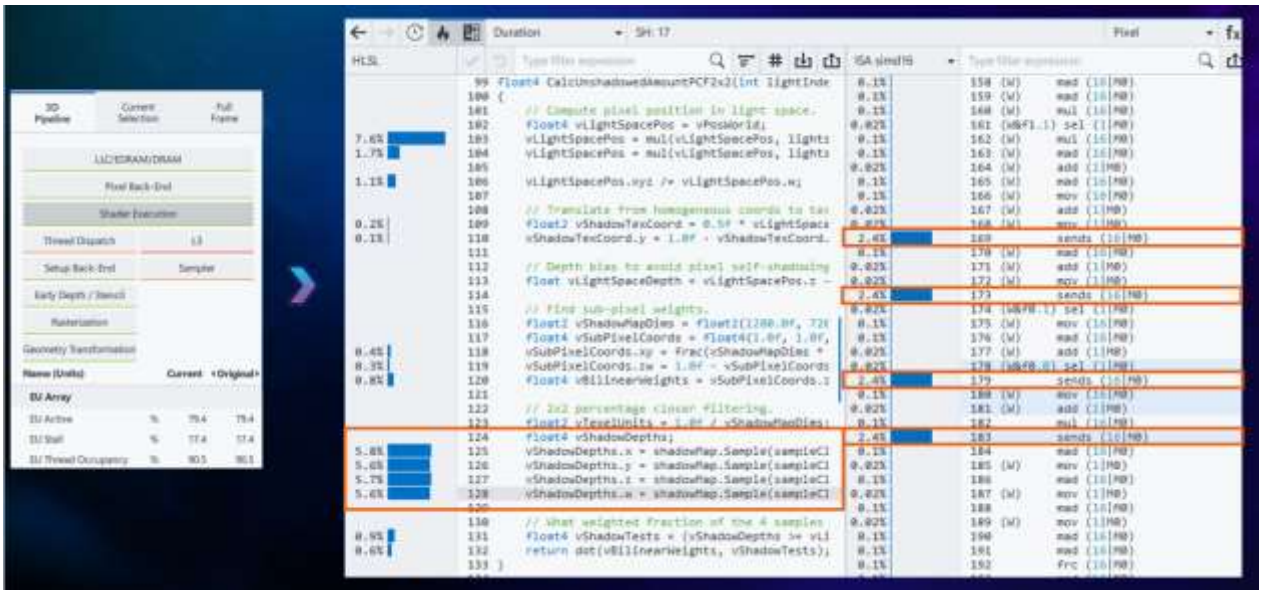
さらに詳しく解析するには、Shader Profiler を使用して命令ごとの実行レイテンシーを確認します。皆さんもご存じのように、レイテンシーとストールは必ずしも同じではありません。レイテンシーが大きい命令ほどストールする可能性が高くなります。EU ストールのボトルネックを調査する場合、Shader Profiler はストールの原因となる可能性が高い命令とその情報を提供します。

**Shader Profiler を有効にする**

次に示すように Shader Profiler にアクセスいます。**[Resource]** リストの任意のシェーダー (ここでは SH:17) をクリップすると、**[Resource]** パネルにシェーダーコードが表示されます。次の **[Resource]** ペインの上部にあるフレームボタンをクリックすると、最も時間を要する行にタイミングが注釈されたシェーダーコードが表示され、実行回数と持続時間 (フレーム時間の消費パーセンテージ) を切り替えできます。

**アセンブリー・コードにシェーダーコードをマップ**

L2 キャッシュがボトルネックである可能性がある場合、SEND ユニットからの送信コマンドを含むアセンブリー・コードを調査する必要があります。シェーダーコードの上にある **[Resource]** パネルの　ボタンをクリックすると、シェーダーコードからアセンブリー・コードへのマッピングが示されます。



**L2 ボトルネックの原因を特定**

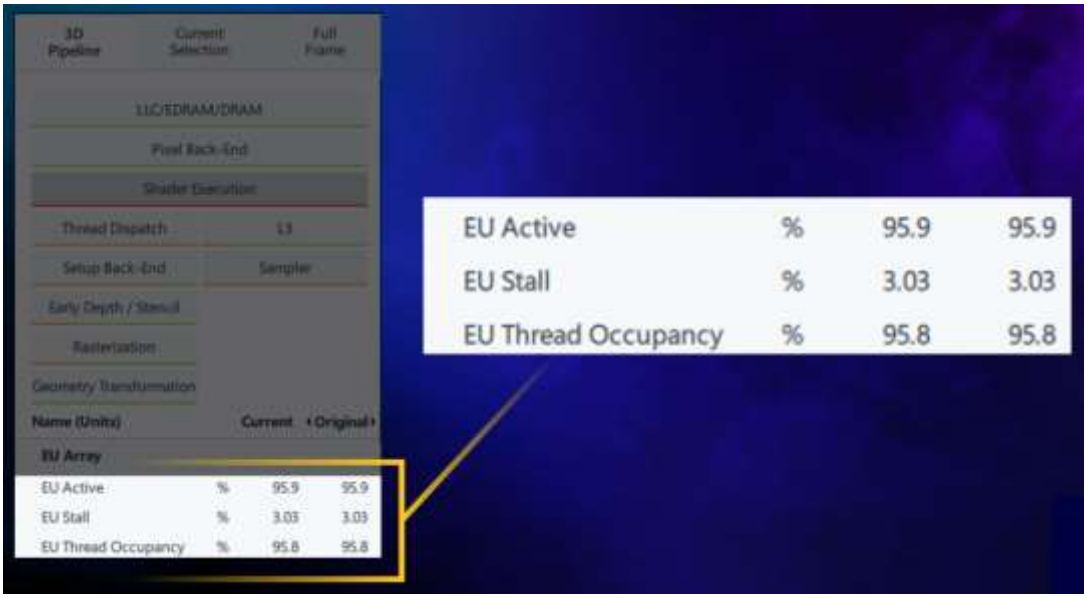
L2 キャッシュがボトルネックとなる根本的な原因を見つけるには、アセンブリー・コードをスクロールして最も時間がかかる SEND 命令を探します。そして、それらに関連するシェーダーコード領域を特定します。

図の場合、`ShadowMap` からサンプリングして定数バッファーをリードする `CalcUnshadowedAmountPCF2x2` 関数がボトルネックの原因となっています。

**ホットスポット: シェーダー実行**

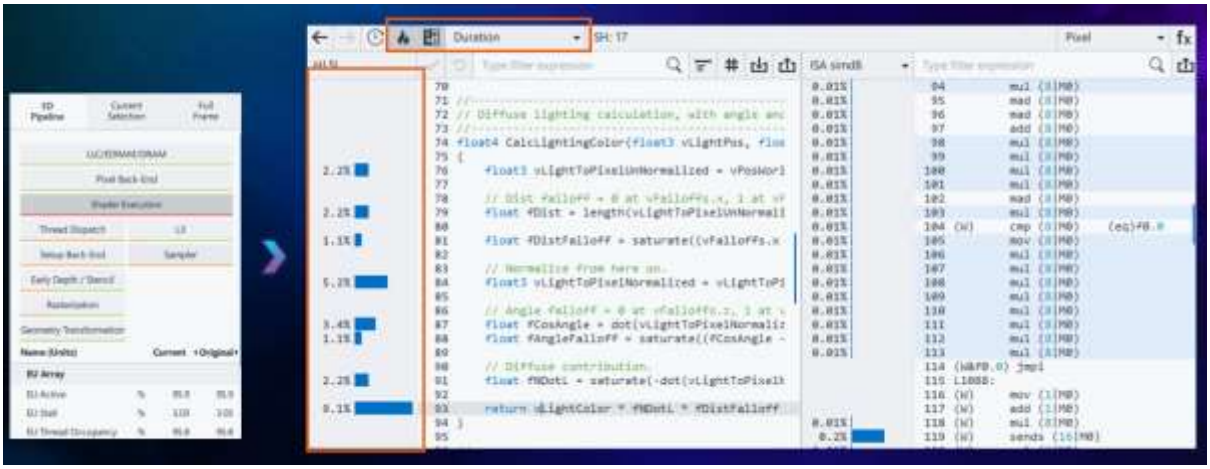
**シェーダー実行のボトルネックの特性**

シェーダー実行のボトルネックには、スレッド占有率が極めて高く、ストール時間が短い特徴があります。これらは許容できます。しかし、アプリケーションの実行時間全体を短縮するのであれば、シェーダーコード自体を最適化する必要があります。



### シェーダー実行ボトルネックの原因を特定

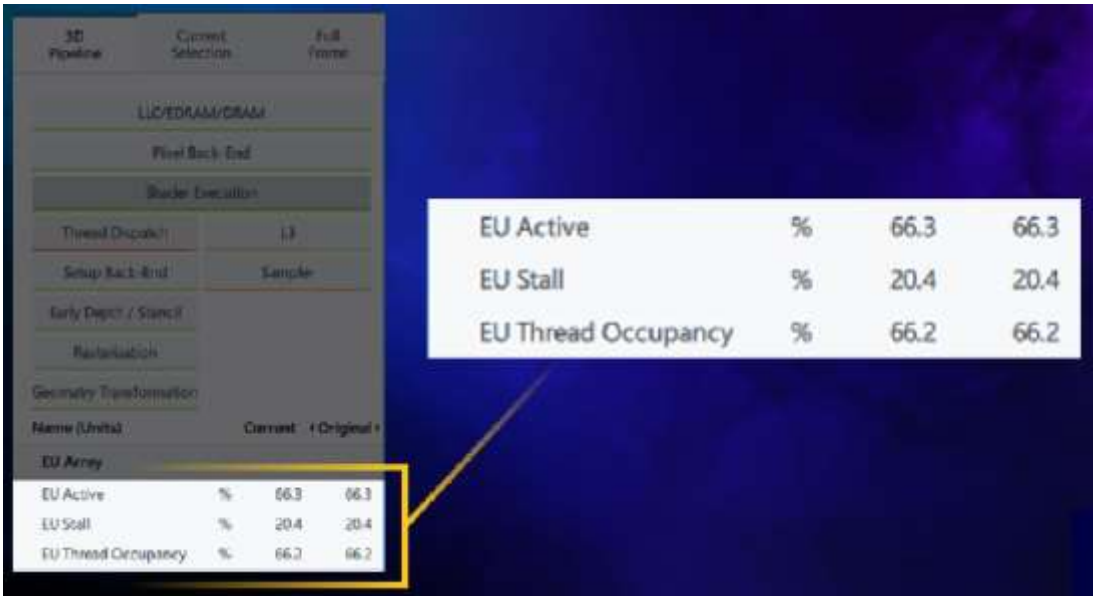
シェーダー実行のボトルネックを見つけるには、シェーダーコードの演算によるホットスポットを解析する必要があります。これには、Shader Profiler を持続時間モード (Duration Mode) に切り替えて、コードをスクロールし、長い実行時間を要するシェーダーコード行を見つけます。`CalcLightingColor` 関数は、単純演算と超越演算の両方を含む計算を行います。図は、単一シェーダーに含まれるこの関数が、フレーム時間全体のおよそ 20% を消費していることを示しています。このボトルネックを解消するには、アルゴリズム自体を最適化する必要があります。



### ホットスポット: スレッドのディスパッチ

### スレッド・ディスパッチのボトルネックの特性

ホットスポット解析の最後の例には、スレッドのディスパッチがボトルネックになっている描画呼び出しのシーケンスがあります。このケースでは、ストール率がかなり高く (20%)、スレッド占有率が低く (66%) なっています。前述のように、低い占有率は EU にロードされているスレッド数が不足していることが原因である可能性があります。これには、シェーダーコードを修正してストール時間を改善するのではなく、EU 全体の占有率を高める必要があります。

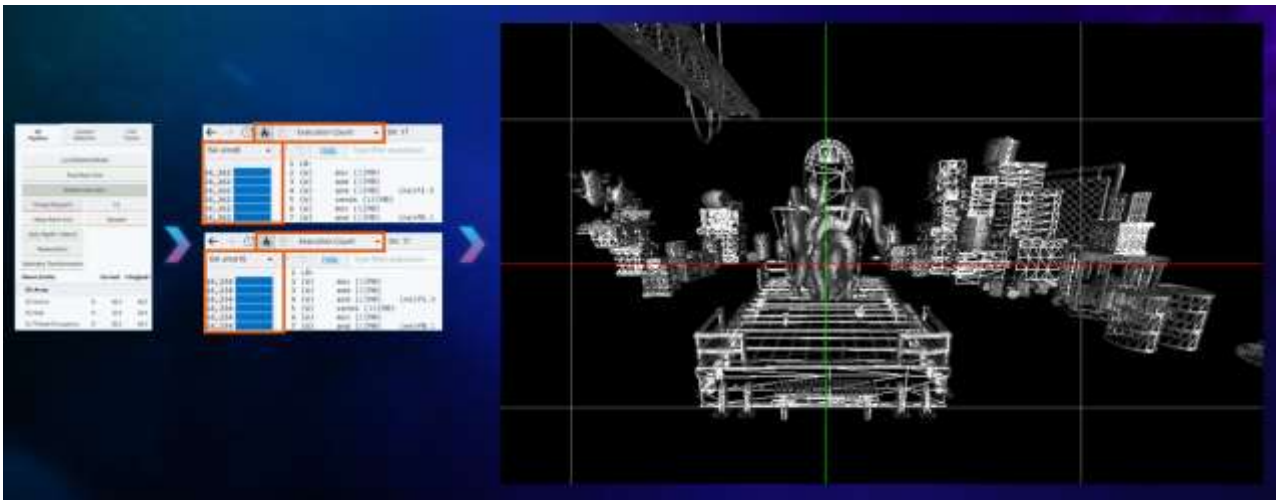


**スレッド・ディスパッチのボトルネックの原因を特定**

SIMD8 と SIMD16 のどちらが適切でしょう?

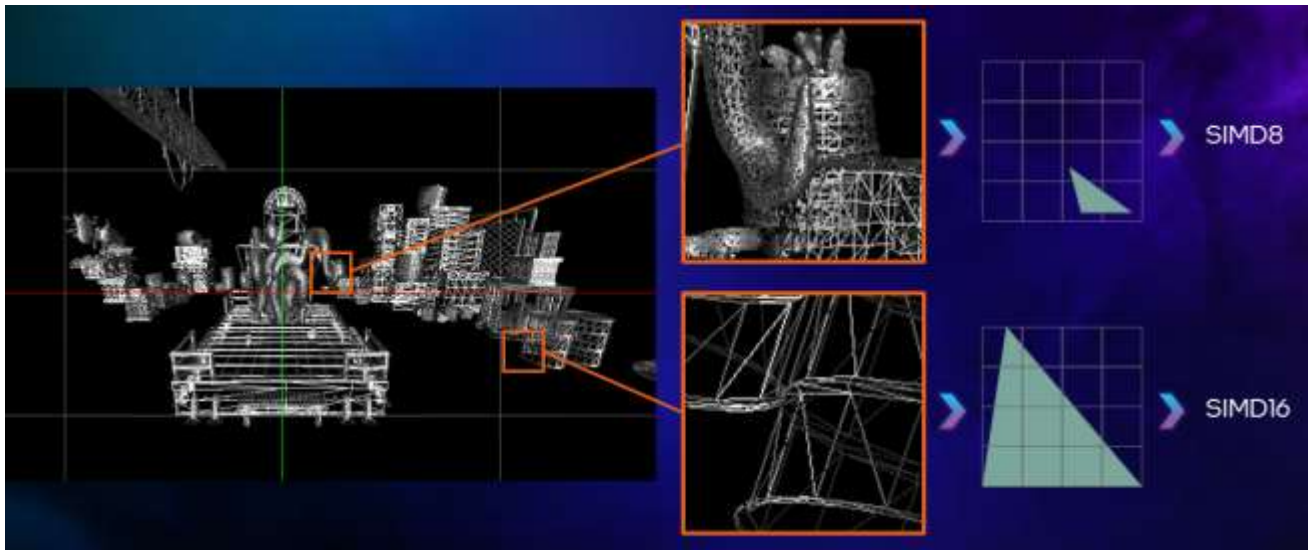
Shader Profiler を開いて、持続時間モードではなく、命令が何回実行されたかカウントする実行カウント (Execution Count) モードに切り替えます。図では、ピクセルシェーダーが SIMD8 と SIMD16 の両方にコンパイルされていることに注目してください。Shader Profiler のレポートを見ると、SIMD8 の命令は 24,000 回実行されていますが、SIMD16 の命令実行は 16,000 回であり、1.5 倍の差があることが分かります。

SIMD16 のスレッドは SIMD8 のスレッドに比べ、一度のディスパッチで 2 倍の演算が可能であるため、多くの SIMD16 スレッドを実行することが望ましいと考えることができます。なぜ SIMD8 スレッドがディスパッチされるのでしょうか? ピクセルシェーダーに 2 つの SIMD バージョンが必要な理由を考えてみましょう。



**ジオメトリーの調査**

これらの描画呼び出しのジオメトリーは、かなり粒度が細かくなっています。観測された異常性は、GPU がピクセル・シェーディングを行う方法に起因します。シェーダー・コンパイラーは、SIMD8 と SIMD16 の 2 つのバージョンを生成します。これは、ピクセルシェーダーのディスパッチャーがラスタライズの結果に基づいて、実行する処理を選択できるようにするため必要なことです。ハードウェアは、ピクセルを 1 つずつシェードするわけではないことを理解する必要があります。シェーディングはグループ単位で行われます。1 つのピクセル・シェーダー・ハードウェア・スレッドは、SIMD16 を使用するため一度に 16 ピクセルをシェーディングします。SIMD16 では、プリミティブのラスタライズが少ない、あるいは 1 つしかない場合でも、GPU は 16 ピクセルをシェーディングして不要なピクセルを廃棄します。そのため、ピクセルシェーダーのディスパッチャーは廃棄を少なくするため、小さなプリミティブに対して SIMD8 をスケジュールします。ここでは、このケースを想定しています。忠実度が高いジオメトリー (小さなプリミティブが多い) が多いため、膨大な数の SIMD8 呼び出しが発生しました。このようなパフォーマンスの問題を解決するには、ゲーム内でジオメトリー LOD を使用する必要があることは、想像のとおりです。




**まとめ**

| ボトルネックのタイプ | 特性化 |
|---|---|
| L2 キャッシュ | 高占有率<br>高ストール |
| シェーダー実行 | 高占有率<br>低ストール |
| スレッドのディスパッチ | 高ストール<br>低占有率 |

このように、シナリオが異なればアプローチも異なります。ときには、GPU をフル稼働させるため、CPU のワークをスピードアップさせたほうが良い場合もあります。一方、シェーダーコードを最適化するほうが適しているケースもあります。また、プリミティブのフォーマットや次元、レイアウトを変更するほうが良い場合もあります。それぞれのシナリオで、Graphics Frame Analyzer はリソースの解析を容易にし、開発者がアプリケーションのフレームレートを最適化する方法を決定するのを支援します。

インテル® GPA を使用して GPU のパフォーマンスを最適化する方法は、インテル® GPA の使用事例 (英語)、ディープダイブとクイックヒント (英語) を参照してください。

# 参考資料

詳細については、以下を参照してください。

- インテル® oneAPI DPC++/C++ コンパイラー デベロッパー・ガイドおよびリファレンス (英語)
- インテル® oneAPI プログラミング・ガイド
- インテル® Fortran コンパイラー・クラッシックおよびインテル® Fortran コンパイラー・デベロッパー・ガイドとリファレンス (英語)
- インテル® oneAPI DPC++/C++ コンパイラーおよびインテル® Fortran コンパイラー向けの GPU への OpenMP オフロード導入ガイド (英語)
- インテル® oneAPI DPC++/C++ コンパイラーにおける OpenMP* 機能と拡張のサポート
- インテル® Fortran コンパイラーで実装される Fortran 言語と OpenMP* 機能
- インテル® oneAPI マス・カーネル・ライブラリー・デベロッパー・リファレンス - C (英語)
- OpenMP API 5.2 の仕様 (英語)
- OpenMP API 5.1 の例 (英語)
- 『Data Parallel C++』 James Reinders 共著 (英語)
- SYCL* 2020 仕様 リビジョン 6 日本語版
- oneAPI ゼロレベル仕様 (英語)

# 法務上の注意書き

本資料は、(明示されているか否かにかかわらず、また禁反言によるとよらずにかかわらず) いかなる知的財産権のライセンスも許諾するものではありません。

インテルは、明示されているか否かにかかわらず、いかなる保証もいたしません。ここにいう保証には、商品適格性、特定目的への適合性、および非侵害性の黙示の保証、ならびに履行の過程、取引の過程、または取引での使用から生じるあらゆる保証を含みますが、これらに限定されるわけではありません。

本資料には、開発中の製品、サービスおよびプロセスについての情報が含まれています。ここに記載されているすべての情報は、予告なく変更されることがあります。インテルの最新の予測、スケジュール、製品仕様およびロードマップをご希望の方は、インテルの担当者までお問い合わせください。

本資料で説明されている製品には、エラッタと呼ばれる設計上の不具合が含まれている可能性があり、公表されている仕様とは異なる動作をする場合があります。現在確認済みのエラッタについては、インテルまでお問い合わせください。

性能に関するテストに使用されるソフトウェアとワークロードは、性能がインテル® マイクロプロセッサー用に最適化されていることがあります。

SYSmark* や MobileMark* などの性能テストは、特定のコンピューター・システム、コンポーネント、ソフトウェア、操作、機能に基づいて行ったものです。結果はこれらの要因によって異なります。製品の購入を検討される場合は、他の製品と組み合わせた場合の本製品の性能など、ほかの情報や性能テストも参考にして、パフォーマンスを総合的に評価することをお勧めします。さらに詳しい情報をお知りになりたい場合は、http://www.intel.com/benchmarks/ (英語) を参照してください。

性能の測定結果はシステム構成の日付時点のテストに基づいています。また、現在公開中のすべてのセキュリティー・アップデートが適用されているとは限りません。詳細については、公開されている構成情報を参照してください。絶対的なセキュリティーを提供できる製品またはコンポーネントはありません。

実際の費用と結果は異なる場合があります。

インテルのテクノロジーを使用するには、対応したハードウェア、特定のソフトウェア、またはサービスの有効化が必要となる場合があります。

Intel、インテル、Intel ロゴ、その他のインテルの名称やロゴは、Intel Corporation またはその子会社の商標です。

* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

©Intel Corporation.