インテル® oneAPI プログラミング・ガイド

Intel Corporation
www.intel.com (英語)

注意事項:

本ドキュメントはレイアウト調整および校閲を行っておりません。誤字脱字、製品名や用語の表記、レイアウト等の不具合が含まれる可能性があることを予めご了承ください。

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『Intel® oneAPI Programming Guide』(**バージョン 2024.0、更新日 2023/11/7**) を iSUS 編集長が翻訳した参考訳です。原文は更新される可能性があります。原文と翻訳文の内容が異なる場合は原文を優先してください。

目次

1 はじめに 7
1.1 oneAPI プログラミングの概要 8
1.2 インテル® oneAPI ツールキットの配布について 9
1.3 関連ドキュメント 10
2 oneAPI プログラミング・モデル 11
2.1 SYCL* を使用した C++ のデータ並列処理 11
2.1.1 キューラムダ参照を使用した簡単なサンプルコード 11
2.1.2 関連情報 13
2.2 C/C++ または Fortran と OpenMP* オフロード・プログラミング・モデル 13
2.2.1 基本的な OpenMP* target 構造 14
2.2.2 `map` 変数 14
2.2.3 `omp target` を使用するコンパイル 16
2.2.4 OpenMP* オフロードの追加のリソース 16
2.3 デバイスの選択 17
2.3.1 ホストコードでの DPC++ デバイス選択 17
2.3.2 デバイス選択の例 18
2.3.3 ホストコードでの OpenMP* デバイスの確認と選択 19
2.4 SYCL* スレッドとメモリー階層 20
2.4.1 スレッド階層 20
2.4.2 メモリー階層 20
2.4.3 データ・プリフェッチを使用して GPU のメモリー・レイテンシーを削減 21
3 oneAPI 開発環境の設定 22
3.1 インストール・ディレクトリー 22
3.2 環境変数 23
3.3 `setvars`、`oneapi-vars` および `vars` ファイル 23
3.4 GPU ドライバーまたはプラグインをインストール (オプション) 23
3.5 `modulefile` (Linux* のみ) 24
3.6 Windows* で `setvars` および `oneapi-vars` スクリプトを使用 24
3.6.1 コンポーネント・ディレクトリー・レイアウトと統合ディレクトリー・レイアウトの違い 24
3.6.2 統合ディレクトリー・レイアウトの利点 25
3.6.3 Visual Studio Code* 拡張 25
3.6.4 コマンドライン引数 26
3.6.5 実行方法 27
3.6.6 確認方法 27
3.6.7 複数の実行 27
3.6.8 統合ディレクトリー・レイアウトの環境変数 29
3.6.9 `ONEAPI_ROOT` 環境変数 29
3.6.10 Windows* で `setvars.bat` 設定ファイルを使用 29
3.6.11 Microsoft* Visual Studio* で `setvars.bat` スクリプトを自動化 33
3.7 Linux* で `setvars` および oneapi-vars スクリプトを使用 34
3.7.1 コンポーネント・ディレクトリー・レイアウトと統合ディレクトリー・レイアウトの違い 35
3.7.2 統合ディレクトリー・レイアウトの利点 35

3.7.3 コマンドライン引数 36
3.7.4 実行方法 37
3.7.5 複数の実行 39
3.7.6 統合ディレクトリー・レイアウトの環境変数 40
3.7.7 ONEAPI_ROOT 環境変数 40
3.7.8 Linux* で setvars.sh 設定ファイルを使用 41
3.7.9 Eclipse* で servars.sh スクリプトを自動化 45
3.7.10 SETVARS_CONFIG 環境変数の状態 46
3.7.11 SETVARS_CONFIG 環境変数の定義 46
3.8 Linux* で modulefile を使用 47
3.8.1 modulefiles ディレクトリーの作成 49
3.8.2 システムに Tcl modulefile 環境をインストール 50
3.8.3 modulefiles-setup.sh スクリプトの使用 51
3.8.4 バージョン管理 52
3.8.5 複数の modulefile 52
3.8.6 oneAPI で使用する modulefile の記述法を理解する 52
3.8.7 modulefiles による module load コマンドの使用 53
3.8.8 関連情報 53
3.9 oneAPI アプリケーションで CMake* を使用 53
4 oneAPI プログラムのコンパイルと実行 55
4.1 単一ソースのコンパイル 55
4.2 コンパイラーの起動 55
4.3 インテル® oneAPI DPC++/C++ コンパイラーの標準オプション 56
4.4 コンパイル例 56
4.4.1 API ベースのコード 57
4.4.2 ダイレクト・プログラミング 59
4.5 コンパイルの手順 60
4.5.1 従来のコンパイル手順 (ホストのみのアプリケーション) 60
4.5.2 SYCL* オフロードコードのコンパイル手順 61
4.5.3 JIT のコンパイル手順 62
4.5.4 AOT のコンパイル手順 63
4.5.5 ファットバイナリー 64
4.6 CPU 手順 64
4.6.1 従来の CPU 向け手順 65
4.6.2 CPU オフロードの手順 65
4.6.3 CPU へコードをオフロード 68
4.6.4 CPU コードの最適化 69
4.6.5 CPU コマンドの例 70
4.6.6 CPU アーキテクチャー向けの事前 (AOT) コンパイル 70
4.6.7 複数の CPU コア上でバイナリーの実行をコントロール 71
4.7 GPU 手順 73
4.7.1 GPU オフロードの手順 74
4.7.2 GPU コマンドの例 80
4.7.3 GPU アーキテクチャー向けの事前 (AOT) コンパイル 81
4.8 FPGA 手順 81

4.8.1 FPGA 向けのコンパイルが特殊である理由 81
4.8.2 SYCL* FPGA コンパイルの種別 82
5 API ベースのプログラミング 86
5.1 インテル® oneAPI DPC++ ライブラリー (インテル® oneDPL) 86
5.1.1 インテル® oneDPL ライブラリーの使い方 87
5.1.2 インテル® oneDPL サンプルコード 87
5.2 インテル® oneAPI マス・カーネル・ライブラリー (インテル® oneMKL) 87
5.2.1 インテル® oneMKL の使い方 88
5.2.2 インテル® oneMKL サンプルコード 89
5.3 インテル® oneAPI スレッディング・ビルディング・ブロック (インテル® oneTBB) 93
5.3.1 インテル® oneTBB の使い方 93
5.3.2 インテル® oneTBB サンプルコード 93
5.4 インテル® oneAPI データ・アナリティクス・ライブラリー (インテル® oneDAL) 94
5.4.1 インテル® oneDAL の使い方 94
5.4.2 インテル® oneDAL サンプルコード 95
5.5 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) 95
5.5.1 インテル® oneCCL の使い方 95
5.5.2 インテル® oneCCL サンプルコード 96
5.6 インテル® oneAPI ディープ・ニューラル・ネットワーク・ライブラリー(インテル® oneDNN) 96
5.6.1 インテル® oneDNN の使い方 97
5.6.2 インテル® oneDNN サンプルコード 99
5.7 その他のライブラリー 99
6 ソフトウェア開発プロセス 100
6.1 SYCL* と DPC++ へのコードの移行 100
6.1.1 C++ から SYCL* への移行 100
6.1.2 DPC++ コンパイラーを使用した CUDA* から SYCL* への移行 100
6.1.3 OpenCL* コードから SYCL* への移行 101
6.1.4 CPU、GPU、および FPGA 間の移行 101
6.2 コンポーザビリティー 104
6.2.1 C/C++ OpenMP* および SYCL* のコンポーザビリティー 104
6.2.2 OpenCL* コードの相互運用性 106
6.3 DPC++ と OpenMP* オフロード処理のデバッグ 106
6.3.1 SYCL* と OpenMP* 開発向けの oneAPI デバッグツール 107
6.3.2 オフロード処理のトレース 119
6.3.3 オフロード処理のデバッグ 121
6.3.4 オフロードのパフォーマンスを最適化 139
6.4 パフォーマンス・チューニング・サイクル 142
6.4.1 ベースラインの確定 142
6.4.2 オフロードするカーネルの特定 143
6.4.3 カーネルをオフロード 143
6.4.4 SYCL* アプリケーションの最適化 143
6.4.5 再コンパイル、実行、プロファイル、そして繰り返し 145
6.5 oneAPI ライブラリーの互換性 146
6.6 SYCL* 拡張 146
7 用語集 147

8 著作権と商標について 150

このガイドでは次のことを学ぶことができます。

- oneAPI プログラミングの概要: oneAPI、インテル® oneAPI ツールキット、および関連するリソースの基本を理解します。
- oneAPI プログラミング・モデル: C、C++、および Fortran の SYCL* および OpenMP* オフロード向けの oneAPI プログラミング・モデルについて紹介します。
- oneAPI 開発環境の設定: oneAPI アプリケーションの開発環境の設定方法を説明します。
- oneAPI プログラムのコンパイルと実行: 各種アクセラレーター (CPU、FPGA など) 向けのコードをコンパイルする詳細を説明します。
- API ベースのプログラミング: 共通 API と関連ライブラリーの簡単な紹介、およびバッファーの使用法の詳細を説明します。
- ソフトウェア開発プロセス: デバッガーやパフォーマンス・プロファイラーなど各種 oneAPI ツールを使用したソフトウェア開発手順の概要、および特定のアクセラレーター (CPU、FPGA など) 向けのコードの最適化を紹介します。

1 はじめに

現代のコンピューター・アーキテクチャーで高い計算パフォーマンスを達成するには、最適化され、電力効率に優れた、スケーラブルなコードが必要です。従来のハイパフォーマンス・コンピューティング (HPC) を始めとして AI、ビデオ解析、データ解析においてハイパフォーマンスの需要は増え続けています。

中央処理装置 (CPU) とグラフィックス処理ユニット (GPU) は、計算エンジンの基本ですが、計算の需要が進化するにつれ、CPU と GPU の違いや、それぞれに最適なワークロードは必ずしも明確ではありません。

現代のワークロードの多様性から、単一のアーキテクチャーですべてのワークロードに対応するのは困難になっており、アーキテクチャーも多様化しています。必要とするパフォーマンスを達成するには、CPU、GPU、AI、および FPGA アクセラレーターに配置されたスカラー、ベクトル、行列、および空間 (SVMS) アーキテクチャーの組み合わせが求められます。

今日、CPU とアクセラレーター (GPU など) 向けのコーディングには、異なる言語、ライブラリー、そしてツールを利用する必要があります。これは、それぞれのハードウェア・プラットフォームは個別のソフトウェア資産を必要とし、異なるターゲット・アーキテクチャー全体ではアプリケーション・コードの再利用が制限されることを意味します。

oneAPI プログラミング・モデルは、SYCL* と呼ばれるプログラミング言語と最新の C++ 機能を使用して並列処理を表現することにより、CPU とアクセラレーターのプログラミングを簡素化します。SYCL* は、単一のソース言語でホスト (CPU など) とアクセラレーター (GPU など) のコードの再利用を可能にし、実行とメモリーの依存関係を明確にします。

SYCL* コード内のマッピング機能により、ハードウェアまたはハードウェア・セットに最適化されたワークロードを実行するためアプリケーションが移行されます。アクセラレーターを使用できないプラットフォームでも、ホストを利用することでデバイスコードの開発とデバッグを簡素化できます。

oneAPI は、既存の C/C++ または Fortran コードで OpenMP* オフロード機能を使用する CPU およびアクセラレーターのプログラミングもサポートします。

CPU と GPU のどちらを使用するか決定する方法については、「CPU と GPU: 両方を最大限に活用する」(英語) を参照してください。

oneAPI プログラミング・モデルを理解したら、『oneAPI GPU 最適化ガイド』でソフトウェアの最適化方法を理解してください。

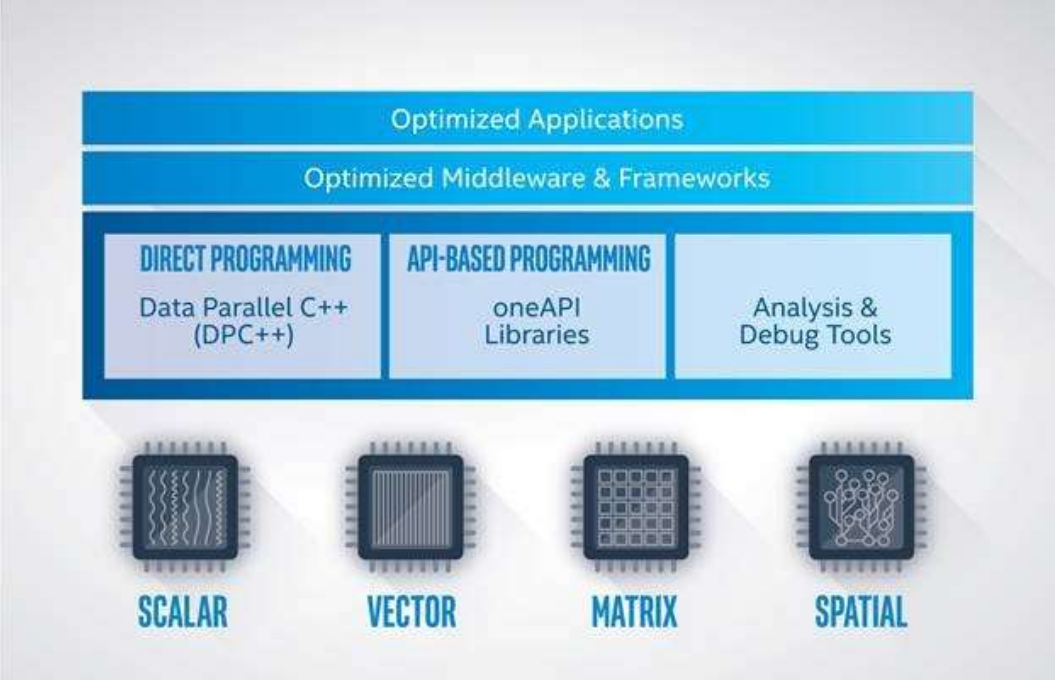
> **注:** すべてのプログラムにおいて、インテル® oneAPI が提供する単一プログラミング・モデルの恩恵を得られるわけではありません。プログラムに適しているかどうかは、設計、実装、およびプログラムで使用する oneAPI プログラミング・モデルを理解する必要があります。

oneapi.com (英語) で oneAPI イニシアチブとプログラミング・モデルの詳細をご覧ください。このサイトでは、oneAPI 仕様、SYCL* 言語ガイドと API リファレンスなど、その他のリソースが提供されます。

1.1 oneAPI プログラミングの概要

oneAPI プログラミング・モデルは、複数のワークロード・ドメインにまたがる広範囲のパフォーマンス・ライブラリーを含む、ハードウェア・ターゲット全体で利用できる開発者向けツールの包括的かつ統合された資源を提供します。ライブラリーには、ターゲット・アーキテクチャーごとにカスタマイズされた関数が含まれているため、同じ関数呼び出しを使用して、サポートされるすべてのアーキテクチャーで最適なパフォーマンスを実現できます。

oneAPI プログラミング・モデル



上の図に示すように、oneAPI プログラミング・モデルを利用するアプリケーションは、CPU から FPGA まで複数のターゲット・ハードウェア・プラットフォームで実行できます。インテルは、一連のツールキットの一部として oneAPI 製品を提供しています。インテル® oneAPI ベース・ツールキットとインテル® HPC ツールキット、およびその他のツールキットは、特定の開発者のワークロード要件を満たす補完的なツールを備えています。例えば、インテル® oneAPI ベース・ツールキットには、インテル® oneAPI DPC++/C++ コンパイラー、インテル® DPC++ 互換性ツール (インテル® DPCT)、ライブラリー、および解析ツールが含まれます。

- 既存の CUDA* コードを DPC++ コンパイラーでコンパイルするため SYCL* に移行しようとする開発者は、**インテル® DPC++ 互換性ツール** を使用して既存のプロジェクトを DPC++ を使用した SYCL* に移行できます。
- **インテル® oneAPI DPC++/C++ コンパイラー**は、アクセラレーターをターゲットとするコードのダイレクト・プログラミングをサポートします。ダイレクト・プログラミングは、ユーザーコードで使用されるアルゴリズムで API が利用できない場合にパフォーマンスを向上するコーディング手法です。CPU と GPU ターゲット向けにはオンラインとオフラインコンパイルがサポートされ、FPGA ターゲット向けにはオフラインコンパイルのみがサポートされます。

- API ベースのプログラミングは、最適化済みのライブラリー・セットを介してサポートされます。oneAPI 製品で提供されるライブラリー関数は、サポートされるターゲット・アーキテクチャー向けに事前チューニングされているため、開発者の介在は必要ありません。例えば、**インテル® oneAPI マス・カーネル・ライブラリー** の BLAS ルーチンは、CPU ターゲットと同様に GPU ターゲットに最適化されています。
- また、コンパイルされた SYCL* アプリケーションは、**インテル® VTune™ プロファイラー**や**インテル® Advisor** などのツールを使用して、パフォーマンス、安定性、エネルギー効率の目標を達成するため、解析およびデバッグできます。

インテル® oneAPI ベース・ツールキットは、インテル® デベロッパー・ゾーン (英語) から無料でダウンロードできます。

インテル® Parallel Studio XE やインテル® System Studio を利用しているユーザーは、インテル® HPC ツールキット (英語) に興味を持つかもしれません。

1.2 インテル® oneAPI ツールキットの配布について

インテル® oneAPI ツールキットは、複数の配布経路から入手できます。

- 製品のローカル・インストール: インテル® デベロッパー・ゾーン (英語) からインテル® oneAPI ツールキットをインストールします。特定のインストールに関する情報は、「インストール・ガイド」(英語) を参照してください。
- コンテナまたはリポジトリーからインストール: サポートされるコンテナまたはリポジトリーからインテル® oneAPI ツールキットをインストールします。それぞれの手順については、「インストール・ガイド」(英語) を参照してください。
- 事前インストールされたインテル® デベロッパー・クラウド: 最新のインテル® ハードウェアにアクセスする無料の開発サンドボックスを使用して、インテル® oneAPI ツールを選択します。インテル® デベロッパー・クラウドの詳細 (英語) を確認して、無料アクセスにご登録ください。

1.3 関連ドキュメント

次のドキュメントは、これから oneAPI プロジェクトを導入する開発者向けの入門資料として役立ちます。

- インテル® oneAPI ツールキットの導入ガイド
  - インテル® oneAPI ベース・ツールキット導入ガイド (Linux* | Windows*) (英語)
  - インテル® HPC ツールキット導入ガイド (Linux* | Windows*) (英語)
- インテル® oneAPI ツールキットのリリースノート
  - インテル® oneAPI ベース・ツールキット (英語)
  - インテル® HPC ツールキット (英語)
- 言語リファレンス
  - SYCL* 言語ガイドと API リファレンス (英語)
  - SYCL* 仕様 PDF (バージョン 1.2.1) (英語)
  - SYCL* 仕様 PDF (バージョン 2020) (英語 (Rev8) | 日本語 (Rev6))
  - James Reinders, Ben Ashbaugh, James Broadman, Michael Kinsner, John Pennycook, and Xinmin Tian 著『Data Parallel C++ : Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL』(英語) — 本書の一部は、Creative Commons のライセンス (英語) の下で再利用されています。
  - LLVM/OpenMP* 関連のドキュメント (英語)
  - OpenMP* 仕様 (英語)

2 oneAPI プログラミング・モデル

ヘテロジニアス・コンピューティングでは、ホスト・プロセッサーはアクセラレーター・デバイスの利点を活用して、コードをより効率良く実行します。

oneAPI プログラミング・モデルは、データ並列 C++ (DPC++) および OpenMP* (Fortran、C、C++) の 2 つのヘテロジニアス・コンピューティング方式をサポートします。

SYCL* はクロスプラットフォームの抽象化レイヤーで、アプリケーションのホストとカーネルのコードが同じソースファイルに含まれる、ヘテロジニアス・プロセッサー用のコードを標準的な ISO C++ を使用して記述することができます。DPC++ オープンソース・プロジェクトは、LLVM C++ コンパイラーに SYCL* のサポートを追加しています。インテル® oneAPI DPC++/C++ コンパイラーは、インテル® oneAPI ベース・ツールキットに含まれています。

OpenMP* は 20 年以上に渡り標準化されてきたプログラミング言語であり、インテルは OpenMP* 標準のバージョン 5 を実装しています。OpenMP* のオフロード機能をサポートするインテル® oneAPI DPC++/C++ コンパイラーは、インテル® oneAPI ベース・ツールキットおよびインテル® HPC ツールキットに含まれます。OpenMP* オフロードをサポートするインテル® Fortran コンパイラー・クラシックとインテル® Fortran コンパイラーは、インテル® HPC ツールキットで提供されます。

注: OpenMP* は FPGA デバイスではサポートされません。

次のセクションでは、それぞれの言語について簡単に説明し詳細情報の参照先を示します。

2.1 SYCL* を使用した C++ のデータ並列処理

C++ で生産性の高いデータ並列プログラミングを行うオープンで、複数ベンダーによる、マルチアーキテクチャーのサポートは、SYCL* をサポートする標準 C++ によって実現されます。SYCL* (‘シクル’ と読みます) は、ロイヤルティー・フリーのクロスプラットフォームの抽象化レイヤーで、アプリケーションのホストとカーネルのコードが同じソースファイルに含まれる、ヘテロジニアス・プロセッサー用のコードを標準的な ISO C++ を使用して記述することができます。DPC++ オープンソース・プロジェクトは、LLVM C++ コンパイラーに SYCL* のサポートを追加しています。

2.1.1 キューラムダ参照を使用した簡単なサンプルコード

SYCL* の導入を示す最良の方法は、簡単なサンプルを使用することでしょう。SYCL* は最新の C++ をベースとしているため、この例ではラムダ式や一様初期化など近年 C++ に追加された、いくつかの機能を使用しています。開発者がこれらの機能に精通していなくても、それらの意味と機能はサンプルのコンテキストから明らかになります。SYCL* によるプログラミングの経験を積んでいくと、これらの新しい C++機能は自然に受け入れられるでしょう。

次のサンプルコードは、`a[0] = 0`、`a[1] = 1`、... のように配列の各要素をそのインデックス値に設定します。

```
1.  #include <CL/sycl.hpp>
2.  #include <iostream>
3.
4.  constexpr int num=16;
5.  using namespace sycl;
6.
7.  int main() {
8.    auto r = range{num};
9.    buffer<int> a{r};
10.
11.   queue{}.submit([&](handler& h) {
12.     accessor out{a, h};
13.     h.parallel_for(r, [=](item<1> idx) {
14.       out[idx] = idx;
15.     });
16.   });
17.
18.   host_accessor result{a};
19.   for (int i=0; i<num; ++i)
20.     std::cout << result[i] << "\n";
21. }
```

最初に気付くことは、ソースファイルが 1 つしかないことです。つまり、ホストコードとオフロードされるアクセラレーター・コードの両方がこの単一のソースファイルから生成されます。次に注目すべき点は、構文が標準の C++ であるということです。並列処理を表現する新しいキーワードやプラグマは使用されていません。代わりに、並列処理は C++ クラスを介して表現されています。例えば、9 行目にある `buffer` クラスはデバイスにオフロードされるデータを表し、11 行目の `queue` クラスはホストからアクセラレーターへの接続を表します。

ロジックは次のように動作します。8 行目と 9 行目で、初期値を持たない 16 個の `int` 要素の `buffer` を作成します。この `buffer` は配列のように作用します。11 行目でアクセラレーター・デバイスに接続するキュー (`queue`) を作成します。この簡単な例では、SYCL* ランタイムがデフォルトのアクセラレーター・デバイスを選択しますが、アプリケーションによっては、システムのトポロジーを調査して特定のアクセラレーションを選択することもできます。キューが作成されると、この例では `submit()` メンバー関数を呼び出して、アクセラレーターにワークを送信します。この `submit()` 関数の引数はラムダ関数であり、ホスト上ですぐに実行されます。ラムダ関数は次の 2 つのことを行います。1 つは、12 行目でアクセサーを作成します。アクセサーはバッファーの要素を書き込むことができます。次に、13 行目で `parallel_for()` 関数を呼び出してコードをアクセラレーターで実行します。

`parallel_for()` の呼び出しには 2 つの引数があります。1 つはラムダ関数であり、もう 1 つはバッファー内の要素数を示す範囲オブジェクト `r` です。SYCL* は、ラムダ関数がレンジ内のインデックスごとに一度 (バッファー要素ごとに 1 回)、アクセラレーターで呼び出されるように調整します。ラムダは、12 行目で作成された `out` アクセサーを使用してバッファー要素に値を割り当てるだけです。この簡単な例では、ラムダ呼び出し間に依存関係がないため、SYCL* はアクセラレーターで最も効率良い方法で自由に並行して実行できます。

`parallel_for()` を呼び出した後、ホストのコードはアクセラレーターの完了を待たずに処理を続行します。ホストが次に行うことは、18 行目でバッファーの要素を読み取る `host_accessor` を作成することです。SYCL* は、このバッファーがアクセラレーターによって書き込まれたことを認識するため、`host_accessor` コンストラクター (18 行目) は、`parallel_for()` によって送信されたワークが完了するまでブロックされます。アクセラレーターのワークが完了すると、ホストコードは 18 行目以降を続行し、`out` アクセサーを使用してバッファーから値を読み取ります。

2.1.2 関連情報

この SYCL* の概要は、完全なチュートリアルを目指すものではなく、言語機能の一部分を紹介するだけです。ローカルメモリー、バリア、SIMD など一般にアクセラレーター・ハードウェアで使用するため学ぶべきことはほかにもたくさんあります。一度に複数のアクセラレーター・デバイスにワークを送信する機能もあり、1 つのアプリケーションが複数のデバイスで同時にワークを並行して実行することもできます。

以下のリソースは、DPC++ を使用して SYCL* を学習して習得するのに役立ちます。

- 「インテルのサンプルを使用した SYCL* の調査」(英語) では、GitHub* から入手できるサンプル・アプリケーションの紹介とリンクを示しています。
- 「DPC++ 基礎サンプルコードのウォークスルー」(英語) は、最初の一歩である「HelloWorld」アプリケーションに相当する DPC++ ベクトル加算のサンプルコードを詳しく見ていきます。
- oneapi.com (英語) サイトでは、クラスとそれらのインターフェイスの説明が記載された『言語ガイドと API リファレンス』(英語) が公開されています。また、4 つのプログラミング・モデル (プラットフォーム、実行モデル、メモリーモデル、およびカーネル・プログラミング・モデル) を詳しく説明しています。
- 「DPC++ エッセンシャル・トレーニング・コース」(英語) は、インテル® デベロッパー・クラウドで Jupyter* Notebook を使用するガイド付きの学習コースです。iSUS から日本語パッケージが提供されています。
- 『Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL* (データ並列 C++: C++ と SYCL* を使用したヘテロジニアス・システムのプログラミング向けに DPC++ を習得)』(英語) は、SYCL* とヘテロジニアス・プログラミングに関連するプログラミングの概念と言語の詳細を紹介する書籍です。

2.2 C/C++ または Fortran と OpenMP* オフロード・プログラミング・モデル

インテル® oneAPI DPC++/C++ コンパイラーおよびインテル® Fortran コンパイラーを使用すると、OpenMP* ディレクティブを使用してワークをインテルのアクセラレーター・デバイスにオフロードし、アプリケーションのパフォーマンスを向上できます。

このセクションでは、OpenMP* ディレクティブを使用して計算をアクセラレーター・デバイスにオフロードする方法について説明します。OpenMP* ディレクティブに慣れていない開発者は、『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) や『インテル® Fortran 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) の OpenMP* サポートのセクションで基本的な使い方をご覧いただけます。

注: OpenMP* は FPGA デバイスではサポートされません。

2.2.1 基本的な OpenMP* target 構造

OpenMP* `target` 構造は、ホストからターゲットデバイスへ制御を移行するために使用されます。変数はホストとターゲットデバイスでマッピングされます。ホストスレッドは、オフロードされた計算が完了するまで待機します。ほかの OpenMP* タスクは、ホストで非同期に実行できます。それには、`nowait` 節を使用して、スレッドがターゲット領域の完了を待機しないようにします。

C/C++

次の C++ のコードは、SAXPY 計算をアクセラレーターにオフロードします。

```
#pragma omp target map(tofrom:fa), map(to:fb,a)
#pragma omp parallel for firstprivate(a)
for(k=0; k<FLOPS_ARRAY_SIZE; k++)
       fa[k] = a * fa[k] + fb[k]
```

配列 `fa` は、計算の入力と出力の両方で使用されるため、アクセラレーターの `to` と `from` にマップされます。配列 `fb` と変数 `a` は計算の入力であり変更されることがないため、その出力をコピーする必要はありません。変数 `FLOPS_ARRAY_SIZE` はアクセラレーターに暗黙にマップされます。ループ・インデックス `k` は、OpenMP* 仕様に従って暗黙的にプライベートです。

Fortran

この Fortran コードは、行列乗算をアクセラレーターにオフロードします。

```
!$omp target map(to: a, b ) map(tofrom: c )
!$omp parallel do private(j,i,k)
      do j=1,n
         do i=1,n
            do k=1,n
               c(i,j) = c(i,j) + a(i,k) * b(k,j)
            enddo
         enddo
      enddo
!$omp end parallel do
!$omp end target
```

配列 `a` と `b` はアクセラレーターの入力にマップされ、配列 `c` はアクセラレーターの入力と出力にマップされます。変数 `n` はアクセラレーターに暗黙にマップされます。ループ・インデックスは OpenMP* の仕様に従って自動的に `private` となるため、`private` 節はオプションです。

2.2.2 map 変数

ホストとアクセラレーター間のデータ共有を最適化するため、`target data` ディレクティブは変数をアクセラレーターにマップし、変数はその領域の範囲内でターゲットのデータ領域に維持されます。この機能は、複数のターゲット領域にまたがって変数をマップするのに役立ちます。

C/C++

```
#pragma omp target data [節[[,] 節],…]
  構造化ブロック
```

Fortran

```
!$omp target data [節[[,] 節],…]
  構造化ブロック
!$omp end target data
```

節の使用例

節には次の 1 つ以上を指定できます。詳細は、TARGET DATA (英語) を参照してください。

- DEVICE (整数式)
- IF ([TARGET DATA :] スカラー論理式)
- MAP ([[マップタイプ修飾子 [,]] マップタイプ:] リスト)

  注: マップタイプには以下を複数指定できます。

  - alloc
  - to
  - from
  - tofrom
  - delete
  - release
- SUBDEVICE ([整数定数,] 整数式[: 整数式[: 整数式]])
- USE_DEVICE_ADDR (リスト) // ifx でのみ利用可能
- USE_DEVICE_PTR (ポインターリスト)

注: SUBDEVICE 節は以下のケースでは無視されます。

- ZE_FLAT_DEVICE_HIERARCHY が FLAT または COMBINED に設定されている。
- LIBOMPTARGET_DEVICES 環境変数が SUBDEVICE/SUBSUBDEVICE に設定されている。
- ONEAPI_DEVICE_SELECTOR 環境変数を使用してデバイスを選択している。

```
DEVICE (整数式)
IF ([TARGET DATA :] スカラー論理式)
MAP ([[マップタイプ修飾子 [,]] マップタイプ: alloc | to | from | tofrom | delete | release] リスト)
SUBDEVICE ([整数定数,] 整数式[ : 整数式[ : 整数式]])
USE_DEVICE_ADDR (リスト) // ifx でのみ利用可能
USE_DEVICE_PTR (ポインターリスト)
```

`target update` ディレクティブまたは、`map` 節で `always` マップ修飾子を使用して、ホストの変数をデバイスの対応する変数と同期することができます。

2.2.3 `omp target` を使用するコンパイル

次のコマンドは、OpenMP* `target` を使用するアプリケーションをコンパイルする例を示します。

C/C++

- Linux*:

```
$ icx -fiopenmp -fopenmp-targets=spir64 code.c
```

- Windows* (`icx` または `icpx` を使用):

```
$ icx /Qiopenmp /Qopenmp-targets=spir64 code.c
```

Fortran

- Linux*:

```
$ ifx -fiopenmp -fopenmp-targets=spir64 code.f90
```

- Windows*:

```
$ ifx /Qiopenmp /Qopenmp-targets=spir64 code.f90
```

2.2.4 OpenMP* オフロードの追加のリソース

- インテルは、OpenMP* ディレクティブを使用してアクセラレーターをターゲットとする以下の具体的なサンプルを、https://github.com/oneapi-src/oneAPI-samples/tree/master/DirectProgramming (英語) で提供しています。

  具体的なサンプルには以下があります。

- 行列乗算 (英語) は、2 つの大きな行列を乗算して結果を検証する簡単なプログラムです。このプログラムは、SYCL* または OpenMP* の 2 つの方法で実装されます。
- ISO3DFD (英語) サンプルは、等方性媒質における 3 次元有限差分波伝搬を参照しています。このサンプルは、3D 等方性媒質を伝搬する波形をシミュレートする 3 次元ステンシルであり、複雑なアプリケーションで OpenMP* アクセラレーター・デバイスをターゲットとして高いパフォーマンスを実現する一般的な課題といくつかの手法を示しています。
- `openmp_reduction` (英語) は円周率を求める簡単なプログラムです。このプログラムは、インテル® アーキテクチャー・ベースの CPU およびアクセラレーター向けの C++ および OpenMP* により実装されています。
- LLVM/OpenMP* ランタイム (英語) は、利用可能な各種タイプのランタイムについて説明しており、OpenMP* オフロードをデバッグする際に役立ちます。
- oneAPI GPU 最適化ガイド (英語 | 日本語) では、oneAPI プログラムで最高の GPU パフォーマンスを実現するさまざまなヒントが提供されています。
- インテル® ツールを使用した OpenMP* アプリケーションのオフロードと最適化 (英語) では、OpenMP* ディレクティブを使用してアプリケーションに並列処理を追加する方法を説明しています。
- openmp.org の例題ドキュメントでは、第 4 章でアクセラレーターと `target` 構造に焦点を当てています。https://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf (英語)
- 『Using OpenMP - the Next Step (OpenMP* を使用する - 次のステップ)』は、OpenMP* の優れた参考書籍です。第 6 章では、ヘテロジニアス・システムにおける OpenMP* のサポートについて説明しています。この書籍の追加情報については、https://www.openmp.org/tech/using-openmp-next-step (英語) をご覧ください。
- OpenMP* オフロード機能の導入 (英語) では、サポートされるオプションやサンプルコードなど、インテル® コンパイラーで OpenMP* オフロードを使用する方法の詳細については、インテル® コンパイラーのデベロッパー・ガイドおよびリファレンス』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® Fortran コンパイラー・クラシックおよびインテル® Fortran コンパイラー・デベロッパー・ガイドおよびリファレンス (英語)

2.3 デバイスの選択

デバイス (CPU、GPU または FPGA など) へのコードのオフロードは、DPC++ アプリケーションと OpenMP* アプリケーションの両方で利用できます。

2.3.1 ホストコードでの DPC++ デバイス選択

ホストコードは明示的にデバイスタイプを選択できます。デバイスを選択には、キューを選択して次のいずれかのデバイスを初期化します。

- `default_selector`
- `cpu_selector`

- gpu_selector
- accelerator_selector

default_selector が使用されると、カーネルは利用可能な計算デバイス (すべて、または ONEAPI_DEVICE_SELECTOR 環境変数の値に基づくサブセット) から選択するヒューリスティックに基づいて実行されます。

特定のデバイスタイプ (cpu_selector や gpu_selector) を使用する場合、指定されたデバイスタイプがプラットフォームで利用可能であるが、ONEAPI_DEVICE_SELECTOR で指定されるフィルターに含まれていなければなりません。指定したデバイスが利用できない場合、ランタイムシステムはデバイスが利用できないことを示す例外をスローします。このエラーは、事前コンパイル (AOT) したバイナリーが、指定するデバイスタイプを含まないプラットフォームで実行される場合にスローされることがあります。

注: DPC++ アプリケーションは、サポートされる任意のターゲット・ハードウェアで実行できますが、特定のターゲット・ハードウェアで最高のパフォーマンスを引き出すにはチューニングが必要です。例えば、CPU 向けにチューニングされたコードは、変更なしでは GPU アクセラレーターでは高速に実行できない可能性があります。

ONEAPI_DEVICE_SELECTOR は、DPC++ ランタイムで使用されるランタイム、計算デバイスタイプ、計算デバイス ID を利用可能なすべての組み合わせのサブセットに制御できる複雑な環境変数です。計算デバイス ID は、SYCL* API、clinfo または sycl-ls (0 から始まる番号) によって返される ID に対応し、その ID を持つデバイスが特定のタイプであるか、特定のランタイムをサポートするかは関係ありません。プログラムが特定のセレクター (gpu_selector など) を使用して、ONEAPI_DEVICE_SELECTOR のフィルターで除外されたデバイスを要求すると、例外がスローされます。使い方と設定可能な値の例については、GitHub* の環境変数の説明をご覧ください

https://github.com/intel/llvm/blob/sycl/sycl/doc/ EnvironmentVariables.md (英語)

sycl-ls ツールを使用して、システムで利用可能なデバイスを確認できます。SYCL* や DPC++ プログラムを実行する前に、このツールでデバイスを確認することを推奨します。sycl-ls は、ONEAPI_DEVICE_SELECTOR に設定されている文字列を各デバイスのプリフィクスとして出力します。sycl-ls の出力形式は、[ONEAPI_DEVICE_SELECTOR] プラットフォーム名、デバイス名、デバイスのバージョン [ドライバーのバージョン] です。次の例で各行の先頭の角かっこ ([ ]) で囲まれた文字列は、プログラムが実行される特定のデバイスを指定する ONEAPI_DEVICE_SELECTOR 文字列です。

2.3.2 デバイス選択の例

```
$ sycl-ls
[opencl:acc:0] Intel® FPGA Emulation Platform for OpenCL™, Intel® FPGA Emulation Device 1.2
[2021.12.9.0.24_005321]
[opencl:gpu:1] Intel® OpenCL HD Graphics, Intel® UHD Graphics 630 [0x3e92] 3.0 [21.37.20939]
[opencl:cpu:2] Intel® OpenCL, Intel® Core™ i7-8700 CPU @ 3.20GHz 3.0 [2021.12.9.0.24_005321]
[level_zero:gpu:0] Intel® Level-Zero, Intel® UHD Graphics 630 [0x3e92] 1.1 [1.2.20939]
[host:host:0] SYCL host platform, SYCL host device 1.2 [1.2]
```

デバイス選択に関する詳しい情報は、『DPC++ 言語ガイドと API リファレンス』(英語) で入手できます。

2.3.3 ホストコードでの OpenMP* デバイスの確認と選択

OpenMP* では、開発者がデバイス上でコードを実行できるか確認および設定する API が用意されています。ホストコードはデバイス番号を明示的に選択および設定できます。開発者は、特定のオフロード領域ごとに `device` 句を使用して、オフロード領域を実行するターゲットデバイスを指定できます。

- `int omp_get_num_procs (void)` API は、デバイスで使用可能なプロセッサー数を返します。
- `void omp_set_default_device(int device_num)` API は、コードまたはデータをオフロードするデフォルトのターゲットデバイスを設定します。
- `int omp_get_default_device(void) API` は、デフォルトのターゲットデバイスを返します。
- `int omp_get_num_devices(void)` API は、コードまたはデータをオフロードできるホスト以外のデバイスの数を返します。
- `int omp_get_device_num(void)` API は、呼び出したスレッドが実行されているデバイスのデバイス番号を返します。
- `int omp_is_initial_device(int device_num)` API は、現在のタスクがホストデバイスで実行されている場合は `true` を返し、それ以外は `false` を返します。
- `int omp_get_initial_device(void)` API は、ホストデバイスを表すデバイス番号を返します。

開発者は、環境変数 `LIBOMPTARGET_DEVICETYPE = [ CPU | GPU ]` で実行するデバイスタイプを選択できます。CPU や GPU のように特定のデバイスが指定される場合、そのデバイスがプラットフォームで利用可能であることが求められます。指定するデバイスが利用できない場合、ランタイムシステムは環境変数 `OMP_TARGET_OFFLOAD` に従って動作します。`OMP_TARGET_OFFLOAD=mandatory` の場合、要求されたデバイスが利用できないという意味のメッセージを出力します。それ以外の場合はベースデバイス (通常は CPU) でフォールバック実行されます。デバイスの選択に関する追加機能は、OpenMP* 5.2 仕様:で確認できます。

環境変数に関する詳細は、以下の GitHub* ページから入手できます。
https://github.com/intel/llvm/blob/sycl/sycl/doc/ EnvironmentVariables.md. (英語)

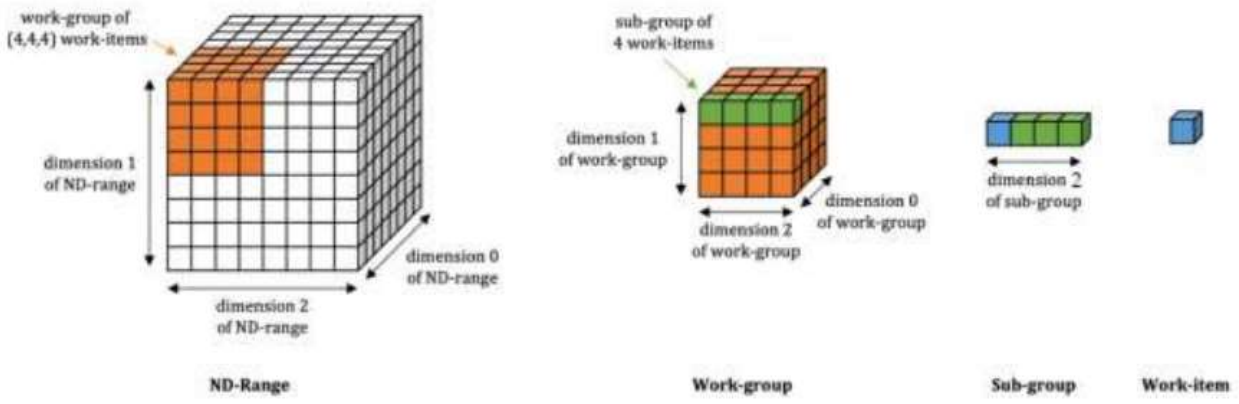
関連情報

- FPGA デバイスセレクター

2.4 SYCL* スレッドとメモリー階層

2.4.1 スレッド階層

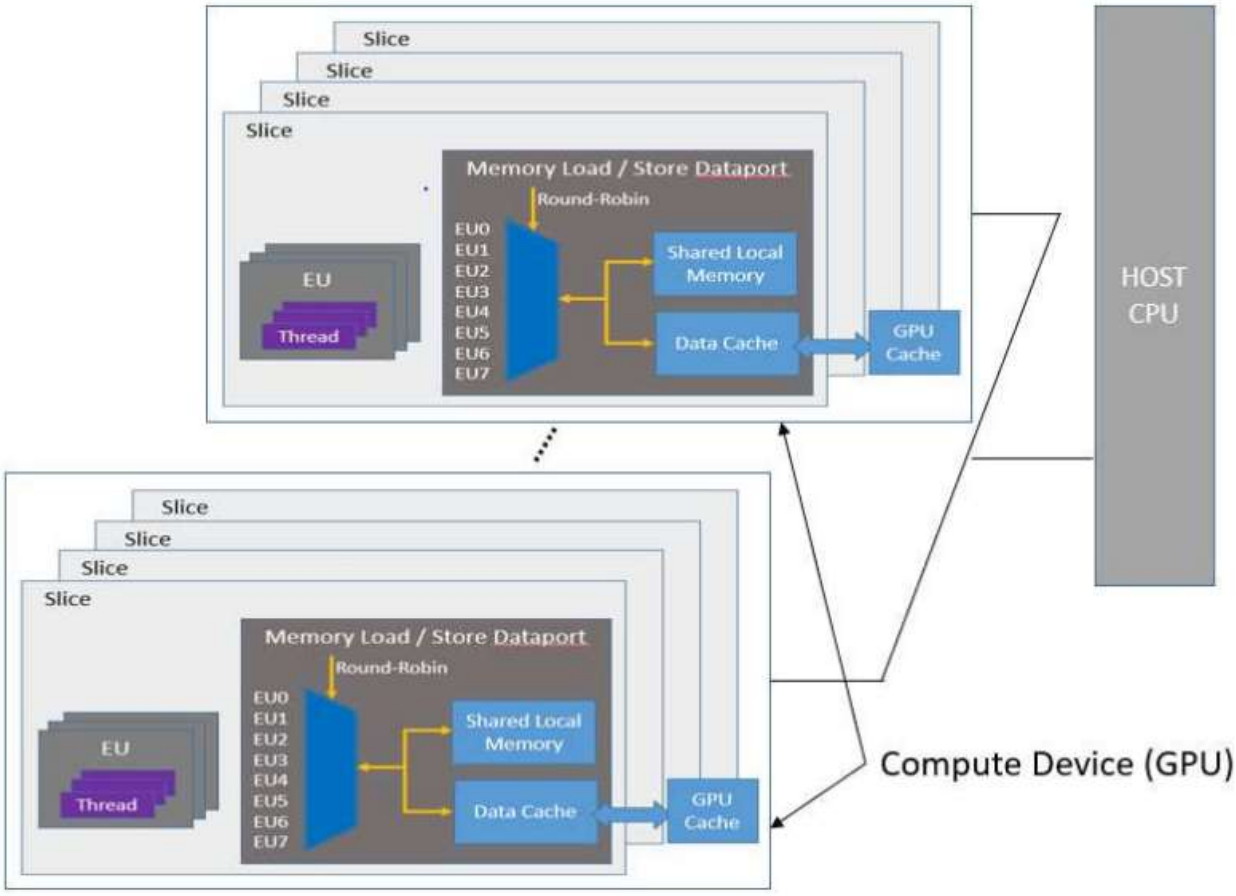
SYCL* 実行モデルでは、GPU 実行の抽象化されたビューを提供します。SYCL* スレッド階層は、ワーク項目の 1 次元、2 次元、または 3 次元のグリッドで構成され、work-group と呼ばれる同じサイズのスレッドグループにグループ化されます。work-group 内のスレッドはさらに、sub-group と呼ばれる同じサイズのベクトルグループに分割されます。



この階層が GPU またはインテル® グラフィックスを搭載する CPU でどのように機能するかは、『oneAPI GPU 最適化ガイド』の「SYCL* スレッドのマッピングと GPU 占有率」を参照してください。

2.4.2 メモリー階層

汎用 GPU (GPGPU) 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。それぞれの計算デバイスは、実行ユニット (EU) または X^e 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) で構成されます。次の図に示すように、計算デバイスには、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) などが含まれることもあります。アプリケーションは、ホストのソフトウェア (ホスト・フレームワークごと) と、事前定義されたデカップリング・ポイントで VE で実行するためにホストから送信されたカーネルの組み合わせとして構築されます。



汎用 GPU (GPGPU) 計算モデル内のメモリー階層の詳細は、『oneAPI GPU 最適化ガイド』の「GPU 実行モデル概要」を参照してください。

2.4.3 データ・プリフェッチを使用して GPU のメモリー・レイテンシーを削減

データをプリフェッチすると、ライトバックの量とレイテンシーが減少し、インテル® GPU のパフォーマンス向上につながります。

oneAPI におけるプリフェッチの仕組みについては、「oneAPI GPU 最適化ガイド」の「プリフェッチ」の節を参照してください。

3 oneAPI 開発環境の設定

インテル® oneAPI ツールは、このドキュメントの最初にある「インテル® oneAPI ツールキットの配布について」で説明するように、いくつかの形式で利用できます。『インテル® oneAPI インストール・ガイド』(英語) の指示に従って、ツールを入手してインストールします。

3.1 インストール・ディレクトリー

Windows* システムでは、インテル® oneAPI 開発ツールキット (ベース、HPC、レンダリングなど) は通常、コンポーネントのデフォルト・ディレクトリーである `C:\Program Files (x86)\Intel\oneAPI\` にインストールされます。ツールキットをインストールすると、統合ディレクトリーが作成され、`C:\Program Files (x86)\Intel\oneAPI\<toolkit-version>` 内のそれぞれのコンポーネントのディレクトリーにリンクされます。

Linux* システムでは、インテル® oneAPI 開発ツールキット (ベース、HPC、レンダリングなど) は通常、コンポーネントのデフォルト・ディレクトリーである `/opt/intel/oneapi/` にインストールされます。ツールキットをインストールすると、統合ディレクトリーが作成され、`/opt/intel/oneapi/<toolkit-version>/` 内のそれぞれのコンポーネントのディレクトリーにリンクされます。

コンポーネント・ディレクトリーのレイアウトと統合ディレクトリーのレイアウトの違いについては、「Windows* で setvars.bat および oneapi-vars.bat スクリプトを使用する」、または「Linux* で setvars および oneapi-vars スクリプトを使用する」を参照してください。

> **注:** 2024 リリース以降では、macOS* はインテル® oneAPI ツールキットおよびコンポーネントでサポートされなくなりました。oneAPI スレッディング・ビルディング・ブロック (oneTBB) やインテル® Implicit SPMD Program Compiler など、いくつかのオープンソース・プロジェクトは、引き続き Apple シリコン上の macOS* をサポートします。これらのツール開発の貢献と協力は歓迎します。

デフォルトのインストール先はインストール中に変更できます。

oneAPI インストール・ディレクトリー内には、開発システムにインストールされているコンパイラー、ライブラリー、解析ツール、およびそのほかのツールを含むフォルダーが含まれます。正確なファイルは、インストールされるツールキットとインストール中に選択されるオプションによって異なります。oneAPI インストール・ディレクトリー内のほとんどのフォルダーは、コンポーネント名に直結する分かりやすい名前が付いています。例えば、`mkl` フォルダーにはインテル® oneAPI マス・カーネル・ライブラリー (oneMKL) が含まれ、`ipp` フォルダーにはインテル® IPP ライブラリーが含まれます。

3.2 環境変数

インテル® oneAPI ツールキットの一部のツールは、次の環境変数に影響されます。

- コンパイルとリンク処理の制御 (`PATH`、`CPATH`、`INCLUDE` など)
- デバッガー、解析ツール、およびローカルヘルプの場所 (`PATH`、`MANPATH` など)
- ツール固有のパラメーターと動的 (共有) リンク・ライブラリーの特定 (`LD_LIBRARY_PATH`、`CONDA_*` など)

3.3 `setvars`、`oneapi-vars` および `vars` ファイル

インストールされるすべてのインテル® oneAPI ツールキットには、親スクリプト `setvars` と、ツール固有のスクリプト `vars` が含まれます (`setvars.sh` と env/`vars.sh` は Linux*、`setvars.bat` と `vars.bat` は Windows*)。これらのスクリプトが実行 (または `source`) されると、各インテル® oneAPI 開発ツールに必要なローカル環境変数が設定されます。

統合ディレクトリー・レイアウトは、2024.0 から実装されています。上位レベルの `oneapi-vars` スクリプトで共通の環境変数を初期化し、オプションの `etc/*/vars.sh` (Linux*) および `etc\*\vars.bat` (Windows*) スクリプトで、`oneapi-vars` スクリプトで設定されないコンポーネント固有の環境変数を初期化します。

次のセクションでは、インテル® oneAPI の `setvars`、`oneapi-vars` および `vars` スクリプトを使用して、インテル® oneAPI 開発環境を初期化する方法を詳しく説明します。

- Windows* で `setvars` と oneapi-vars スクリプトを使用
- Linux* で `setvars` と oneapi-vars スクリプトを使用

3.4 GPU ドライバーまたはプラグインをインストール (オプション)

C++ と SYCL* を使用して、インテル、AMD*、または NVIDIA* GPU で実行できる 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して実行するには、対応するドライバーや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 を使用するには、最新のインテル® GPU ドライバー (英語) をインストールします。
- インテル® oneAPI DPC++ コンパイラーで AMD* GPU と使用するには、Codeplay から oneAPI for AMD* GPU プラグイン (英語) を入手してインストールします (Linux* のみ)。
- インテル® oneAPI DPC++ コンパイラーで NVIDIA* GPU と使用するには、Codeplay から oneAPI for NVIDIA* GPU プラグイン (英語) を入手してインストールします (Linux* のみ)。

3.5 modulefile (Linux* のみ)

環境モジュール (英語) を利用するユーザーは、インテル® oneAPI ツールキットのインストール・パッケージに含まれる modulefile ファイルを使用して、開発環境を初期化することがあります。インテル® oneAPI の modulefile スクリプトは Linux* 環境でのみサポートされており、setvars、oneapi-vars および vars スクリプトの代わりに使用することができます。modulefile ファイルと setvars 環境スクリプトを混在して使用しないでください。

インテル® oneAPI の modulefile を使用して、インテル® oneAPI 開発環境を初期化する方法の詳細については、「Linux* で modulefile を使用」をご覧ください。

3.6 Windows* で setvars および oneapi-vars スクリプトを使用

バージョン 2024.0 では、統合ディレクトリー・レイアウトが実装されました。複数のツールキットのバージョンがインストールされている場合、統合レイアウトにより開発環境に特定のツールキットのバージョンの一部としてリリースされたコンポーネントのバージョンが含まれるようにする機能が実装され、PATH 名がショートカットされて長い PATH 名の問題の解決にも役立ちます。

新しい統合ディレクトリー・レイアウトでは、共通フォルダー (bin、lib、include、share など) にコンポーネントが共にインストールされていることが分かります。これらの共通フォルダーは、ツールキットのバージョン番号に基づいて命名される最上位フォルダーにあります。以下に例を示します。

```
"C:\Program Files(x86)\Intel\oneAPI\2024.0\"
|-- bin
|-- lib
|-- include
...など...
```

2024.0 以前に使用されていたディレクトリー・レイアウトは、新規および既存のインストールで引き続きサポートされます。以前のレイアウトは、コンポーネント・ディレクトリー・レイアウトと呼ばれます。コンポーネント・ディレクトリー・レイアウトまたは統合ディレクトリー・レイアウトを使用するオプションが追加されました。

3.6.1 コンポーネント・ディレクトリー・レイアウトと統合ディレクトリー・レイアウトの違い

ほとんどのインテル® oneAPI コンポーネントのフォルダーには、oneAPI 開発作業をサポートするそれぞれのコンポーネントに必要な環境変数を設定する env\vars.bat スクリプトが含まれています。例えば、デフォルトのインストールでは、Windows* のインテル® インテグレーテッド・パフォーマンス・プリミティブ (インテル® IPP) の vars スクリプトは、C:\Program Files (x86)\Intel\oneAPI\ipp\latest\env\vars.bat に配置されます。このパスは、env\vars 環境変数設定スクリプトを含むすべてのインテル® oneAPI コンポーネントで共有されます。

コンポーネント・ディレクトリー・レイアウトでは、各コンポーネント向けの env\vars スクリプトは、直接またはまとめて呼び出すことができます。まとめて呼び出すには、oneAPI インストール・ディレクトリーにある setvars.bat スクリプトを使用します。これは、Windows* マシンのデフォルトのインストールでは、C:\Program Files (x86)\Intel\oneAPI\setvars.bat にあります。

統合ディレクトリー・レイアウトは、開発環境の初期化に env\vars スクリプトを使用しません。代わりに、各コンポーネントは、コンポーネントに共通の共有フォルダーに包括されます。つまり、各コンポーネントは、ヘッダーファイルを単一の共通インクルード・フォルダーに提供し、そのライブラリー・ファイルを単一の共通 lib フォルダーに提供することになります。

3.6.2 統合ディレクトリー・レイアウトの利点

統合ディレクトリー・レイアウトを使用することで setvars 設定ファイルを構成して維持したり、複数の oneAPI ツールキットをインストールして個別に環境を作成する必要がなく、異なるツールキットのバージョンの切り替えがはるかに容易になりました。また、一部の Windows* 開発者によって厄介な問題である Windows* 開発システムの環境変数、特に PATH 変数の長さを制限するのにも役立ちます。

統合ディレクトリー・レイアウトの環境変数は、一括でのみ設定できます。環境変数を初期化するには oneapi-vars.bat スクリプトを使用します。Windows* 上のデフォルトの統合ディレクトリー・レイアウトのインストールでは、スクリプトは C:\Program Files (x86)\Intel\oneAPI\<toolkit-version>\oneapi-vars.bat にあります。<toolkit-version> は、インストールした oneAPI ツールキットのバージョン番号に対応します (例: C:\Program Files (x86)\Intel\oneAPI\2024.0\oneapi-vars.bat または C:\Program Files (x86)\Intel\oneAPI\2024.1\oneapi-vars.bat など)。

引数なしで setvars.bat スクリプトを実行すると、システムにインストールされているすべての <コンポーネント>\latest\env\vars.bat スクリプトが実行されます。これらの環境変数設定スクリプトを実行した後、Windows* の set コマンドを使用して環境変数を確認できます。

引数なしで oneapi-vars.bat スクリプトを実行すると、スクリプトが配置されているバージョンの環境が構成されます。また、統合ディレクトリーのインストースの一部であるオプションの C:\Program Files (x86)\Intel\oneAPI\<toolkit-version>\etc\<component>\vars.sh スクリプトも実行されます。このスクリプトで変更された環境変数は、oneapi-vars.bat スクリプトの実行後に、Windows* の set や env コマンドで確認できます。

oneapi-vars.bat スクリプトの仕組みの詳細は、「統合ディレクトリー・レイアウトの環境の初期化」を参照してください。

3.6.3 Visual Studio Code* 拡張

Visual Studio Code* 開発者は、oneAPI 環境拡張機能をインストールして、Visual Studio* Code で setvars.bat を実行できます。詳細については、「Visual Studio* Code でインテル® oneAPI ツールキットを使用する」(英語) をご覧ください。

注: setvars.bat、oneapi-vars.bat スクリプト (または個別の vars.bat スクリプト) により変更された環境は永続的ではありません。これらの変更は、setvars.bat、oneapi-vars.bat スクリプトが実行された cmd.exe セッションでのみ有効です。

3.6.4 コマンドライン引数

setvars.bat、oneapi-vars.bat スクリプトはいくつかのコマンドライン引数をサポートしており、--help オプションで引数の一覧を表示できます。

コンポーネント・ディレクトリー・レイアウト

```
$ "C:\Program Files (x86)\Intel\oneAPI\setvars.bat" --help
```

統合ディレクトリー・レイアウト

```
$ "C:\Program Files (x86)\Intel\oneAPI\<toolkit-version>\oneapi-vars.bat" --help
```

--config=file 引数と setvars.bat、oneapi-vars.bat スクリプトから呼び出される vars.bat スクリプトへの追加引数をインクルードする機能を使用して、環境設定をカスタマイズできます。--config=file オプションは、setvars.bat スクリプトによってのみサポートされます。

--config=file 引数は、特定のインテル® oneAPI コンポーネントの環境の初期化機能を提供するとともに、特定のバージョンの環境を初期化することもできます。例えば、インテル® IPP ライブラリーとインテル® oneAPI マス・カーネル・ライブラリー (oneMKL) の環境のみを設定するには、これら 2 つのインテル® oneAPI コンポーネントの vars.bat 環境スクリプトのみを呼び出すように setvars.bat、oneapi-vars.bat スクリプトに指示する設定ファイルを渡します。詳細と利用例については、「Windows* で setvars.bat の設定ファイルを使用」をご覧ください。

setvars.bat、oneapi-vars.bat のヘルプメッセージに記載されていないコマンドライン引数は、そのまま vars.bat スクリプトに渡されます。つまり、setvars.bat、oneapi-vars.bat スクリプトが認識できない引数は、コンポーネントの vars.bat スクリプトで使用されるものと見なし、それらの引数をすべてのコンポーネントの vars.bat スクリプトに渡します。最もよく使用される追加の引数は、ia32 と intel64 です。これらは、インテル® コンパイラー、インテル® IPP、インテル® oneAPI マス・カーネル・ライブラリー (oneMKL)、およびインテル® oneAPI スレッディング・ビルディング・ブロック (oneTBB) ライブラリーでアプリケーションのターゲット・アーキテクチャーを指示するために使用されます。

システムに複数バージョンの Microsoft* Visual Studio* がインストールされている場合、vs2017、vs2019 または vs2022 引数を setvars.bat、oneapi-vars.bat コマンドラインに追加することで、Visual Studio* 環境のいずれかをインテル® oneAPI 環境の初期化に使用するかを指定できます。デフォルトでは、Visual Studio* の最新バージョンが使用されます。

注: Microsoft* Visual Studio* 2017 のサポートはインテル® oneAPI 2022.1 では非推奨となり、将来のリリースで削除される予定です。

個々の vars.bat スクリプトを調べて、受け入れるコマンドライン引数があればそれを決定します。

3.6.5 実行方法

コンポーネント・ディレクトリー・レイアウト

```
<install-dir>\setvars.bat
```

PowerShell ウィンドウで setvars.bat または vars.bat スクリプトを実行するには、以下を使用します。

```
$ cmd.exe "/K" '"C:\Program Files (x86)\Intel\oneAPI\setvars.bat" && powershell'
```

統合ディレクトリー・レイアウト

```
<install-dir>\<tookit-version>\oneapi-vars.bat
```

PowerShell ウィンドウで oneapi-vars.bat または vars.bat スクリプトを実行するには、以下を使用します。

```
$ cmd.exe "/K" '"C:\Program Files (x86)\Intel\oneAPI\< tookit-version >\oneapi-vars.bat" &&
powershell'
```

3.6.6 確認方法

setvars.bat、oneapi-vars.bat を実行した後、SETVARS_COMPLETED 環境変数で設定の成功を確認できます。setvars.bat、oneapi-vars.bat が成功すると、SETVARS_COMPLETED には 1 が設定されます。

```
$ set | find "SETVARS_COMPLETED"
```

戻り値

```
SETVARS_COMPLETED=1
```

SETVARS_COMPLETED=1 以外の場合、setvars.bat、oneapi-vars.bat は設定に失敗したことを意味します。

3.6.7 複数の実行

各コンポーネントの env\vars.bat スクリプトの多くは、PATH、CPATH、およびそのほかの環境変数に変更を加えるため、最上位の setvars.bat、oneapi-vars.bat スクリプトは同じセッションで同じ vars.bat を複数回呼び出

すことはできません。これは、特に %PATH% 環境変数が原因で環境変数の文字数が長くなりすぎないようにします。設定可能な文字数を超えると、ターミナルセッションで予期しない動作を招くことがあるため回避する必要があります。

これを強制するには、setvars.bat、oneapi-vars.bat に --force オプションを指定します。この例では、ユーザーが setvars.bat、oneapi-vars.bat を 2 度実行しています。setvars.bat、oneapi-vars.bat がすでに実行されているため、2 回目の実行は停止します。

コンポーネント・ディレクトリー・レイアウト

```
$ <install-dir>\setvars.bat
 initializing oneAPI environment ...
(SNIP: lot of output)
:: oneAPI environment initialized
```

```
$ <install-dir>\setvars.bat
.. code-block:: WARNING: setvars.bat has already been run.Skipping re-execution.
To force a re-execution of setvars.bat, use the '--force' option.
Using '--force' can result in excessive use of your environment variables.
```

次は、ユーザーが <install-dir>\setvars.bat --force を実行し、初期化が成功した例です。

```
$ <install-dir>\setvars.bat --force
:: initializing environment ...
(SNIP: lot of output)
:: oneAPI environment initialized
```

統合ディレクトリー・レイアウト

```
$  <install-dir>\<toolkit-version>oneapi-vars.bat
:: initializing oneAPI environment ...
(SNIP: lot of output)
:: oneAPI environment initialized
```

```
$  <install-dir>\<toolkit-version>\oneapi-vars.bat
.. code-block:: WARNING: oneapi-vars.bat has already been run. Skipping re-execution.
   To force a re-execution of oneapi-vars.bat, use the '--force' option.
   Using '--force' can result in excessive use of your environment variables.
```

3 番目のインスタンスでは、ユーザーが <install-dir>\<toolkit-version>\oneapi-vars.bat --force を実行すると、初期化が成功します。

```
$ <install-dir>\<toolkit-version>\oneapi-vars.bat --force
:: initializing oneAPI environment ...
(SNIP: lot of output)
:: oneAPI environment initialized
```

3.6.8 統合ディレクトリー・レイアウトの環境変数

統合ディレクトリー・レイアウトの環境変数の初期化は、setvars.bat ではなく oneapi-vars.bat スクリプトによって行われます。oneapi-vars は setvars と似ていますが、微妙な違いがいくつかあります。

setvars スクリプトと oneapi-vars の主な違いは、setvars は環境変数 (ONEAPI_ROOT を除く) を定義しませんが、oneapi-vars は共通の環境変数を定義します。

コンポーネント・ディレクトリー・レイアウトは、各コンポーネントが機能するために必要な環境変数を定義します。例えば、コンポーネント・ディレクトリー・レイアウトでは、各コンポーネントはリンク可能なライブラリー・フォルダーを LD_LIBRARY_PATH に、そしてヘッダーを CPATH などに追加します。コンポーネントは、常に次の場所にある vars スクリプトを介してこれを実行します。

```
$ %ONEAPI_ROOT%\<toolkit-version>\opt\<component-name>\latest\env\vars.bat
```

統合ディレクトリー・レイアウトは、外部向けの include、lib、bin フォルダーを共有フォルダーに統合します。最上位レベルの oneapi-vars スクリプトは、これらの共通フォルダーを検出するのに必要な環境変数を定義します。例えば、setvars は LD_LIBRARY_PATH を $ONEAPI_ROOT\lib として定義し、CPATH を $ONEAPI_ROOT\include と定義します。

3.6.9 ONEAPI_ROOT 環境変数

ONEAPI_ROOT 環境変数は、スクリプトが実行されるときに最上位の setvars.bat と oneapi-vars によって設定されます。ONEAPI_ROOT 環境変数がすでに設定されている場合、setvars.bat と oneapi-vars はスクリプトを実行した cmd.exe セッションを一時的に上書きします。この変数は、oneapi-cli サンプルブラウザーと Microsoft* Visual Studio* および Visual Studio* Code サンプルブラウザーによって使用され、インテル® oneAPI ツールとコンポーネントの検出、および SETVARS_CONFIG 機能が有効である場合に setvars.bat や oneapi-vars スクリプトを検出するのに役立ちます。SETVARS_CONFIG 機能の詳細については、「Microsoft* Visual Studio* で setvars.bat スクリプトを自動化」をご覧ください。

2024.0 リリースでは、インストーラーは、ONEAPI_ROOT 変数を環境に追加しません。これをデフォルト環境に追加するには、ローカルの初期化ファイルまたはシステム環境変数で変数を定義します。

3.6.10 Windows* で setvars.bat 設定ファイルを使用

setvars.bat スクリプトは、それぞれの oneAPI ディレクトリーにある <install-dir>\latest\env\vars.bat スクリプトを実行することで、インテル® oneAPI ツールキットの環境変数を設定します。setvars.bat スクリプトを自動実行しないように Windows* システムを設定しない限り、新しいターミナルウィンドウを開くか Visual Studio*、Sublime Text*、またはそのほかの C/C++ エディターを起動するたびに setvars.bat スクリプトが実行されます。詳細は、「システムの設定」(英語) を参照してください。

注: 設定ファイルは、コンポーネント·ディレクトリー·レイアウトの `setvars.bat` でのみ使用できます。統合ディレクトリー·レイアウトは、`oneapi-vars.bat` を使用しますが、これは設定ファイルをサポートしません。レイアウトの詳細については、「Windows* で setvars および oneapi-vars スクリプトを使用」を参照してください。

以下に設定ファイルを使用して環境変数を管理する方法を説明します。

3.6.10.1. バージョンと構成

一部のインテル® oneAPI ツールは複数バージョンのインストールがサポートされます。複数バージョンをサポートするツールのディレクトリー構造は次のようになります (デフォルトのインストールを想定し、例としてコンパイラーを使用します)。

```
\Program Files (x86)\Intel\oneAPI\compiler\
|-- 2021.1.1
|-- 2021.2.0
`-- latest -> 2021.2.0
```

例:

すべてのツールには、そのコンポーネントの最新バージョンのインストール先を示す `latest` という名前のショートカットがあります。`latest\env\` ディレクトリーにある `vars.bat` スクリプトは、`setvars.bat` によって実行されます (デフォルト)。

必要に応じて、設定ファイルを使用して特定のディレクトリーを示すよう `setvars.bat` をカスタマイズできます。

3.6.10.2. --config パラメーター

最上位の `setvars.bat` スクリプトは、カスタム `config.txt` ファイルを指定する `--config` パラメーターを受け入れます。

```
$ <install-dir>\setvars.bat --config="path\to\your\config.txt"
```

設定ファイルは任意の名前にすることができます。複数の設定ファイルを作成して、さまざまな開発環境やテスト環境を設定できます。例えば、最新バージョンのライブラリーを古いバージョンのコンパイラーでテストしたいこともあります。そのような場合に、setvars 設定ファイルを使用して環境を管理できます。

3.6.10.3. 設定ファイルの例

以下に簡単な設定ファイルの例を示します。

最新のコンポーネントをすべてロード

```
mkl=1.1
dldt=exclude
```

ただし以下のコンポーネントは除外

```
default=exclude
mkl=1.0
ipp=latest
```

設定テキストファイルは次の要件に従う必要があります。

- 改行で区切られたテキストファイル
- 各行は、key=value のペアで構成されます
- key には、oneAPI ディレクトリーの最上位 (%ONEAPI_ROOT% ディレクトリーにあるフォルダー) のコンポーネント名を指定します。同じ key が設定ファイルに複数定義されると、最後の key が優先されそれ以外は無視されます。
- value には、コンポーネント・ディレクトリーの最上位にあるバージョン・ディレクトリー名を指定します。これには、コンポーネント・ディレクトリーのレベルに存在する可能性があるショートカット (latest など) が含まれます。
  - また、value は exclude にすることもできます。これは、指定された key の vars.bat スクリプトを setvars.bat スクリプトで実行しないことを意味します。

key=value を default=exclude にすると特別な意味を持ちます。これは、設定ファイルに定義されているものを除き、それ以外のすべての env\vars.bat スクリプトの実行を除外します。以下に例を示します。

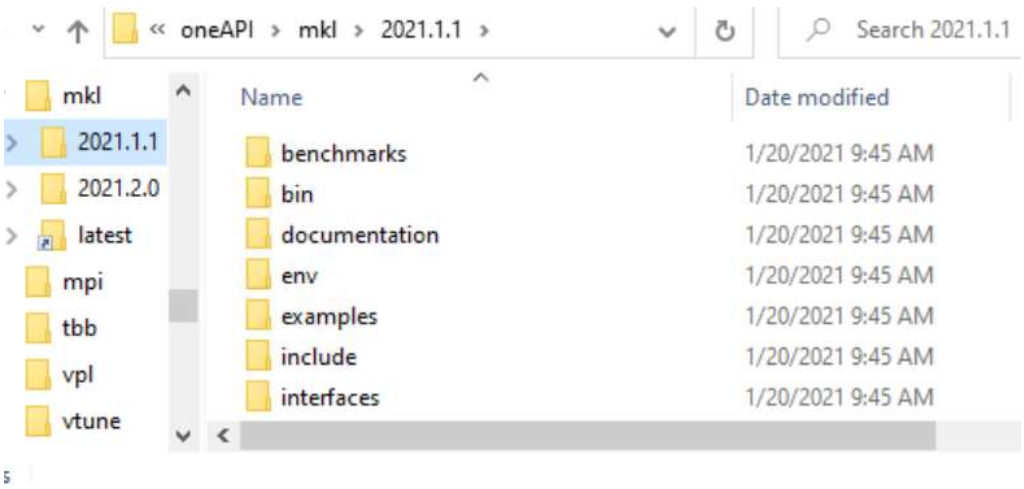
3.6.10.4. 設定ファイルのカスタマイズ

設定ファイルを使用して、特定のコンポーネントを除外したり、特定のバージョンを含めたり、特定のコンポーネントのバージョンのみを含めることができます。これには、設定ファイルの default=exclude 行を変更します。

デフォルトでは、setvars.bat は最新 (latest) のバージョンに対応する env\vars.bat スクリプトを処理します。

例えば、2 つのバージョン (2021.1.1 と 2021.2.0) のインテル® oneAPI マス・カーネル・ライブラリー (oneMKL) がインストールされていると仮定します。最新のバージョンを示すショートカットは 2021.2.0 であるため、デフォルトでは `setvars.bat` は `mkl` ディレクトリーの 2021.2.0 の `vars.bat` スクリプトを実行します。

2 つのバージョンの oneMKL と設定ファイル



特定のバージョンを指定

`setvars.bat` に `<install-dir>\mkl\2021.1.1\env\vars.bat` スクリプトを実行するように直接記述するには、設定ファイルに `mkl=2021.1.1` を追加します。

これにより、`setvars.bat` は、`mkl` ディレクトリー内の 2021.1.1 フォルダーにある `env\vars.bat` スクリプトを実行するようになります。インストールされている `mkl` 以外のコンポーネントでは、`setvars.bat` は最新バージョンのフォルダーにある `env\vars.bat` スクリプトを実行します。

特定のコンポーネントを除外

コンポーネントを除外する構文は次のようになります。

```
<key>=exclude
```

例えば、インテル® IPP を除外して、2021.1.1 の インテル® oneAPI マス・カーネル・ライブラリー (oneMKL) を含めるには次のようになります。

```
mkl=2021.1.1
ipp=exclude
```

この例は次のように作用します。

- `setvars.bat` は、インテル® oneAPI マス・カーネル・ライブラリー (oneMKL) 2021.1.1 の `env\vars.bat` スクリプトを実行します。
- `setvars.bat` は、インテル® IPP の `env\vars.bat` スクリプトを実行しません。

- `setvars.bat` は、そのほかのコンポーネントの最新バージョンの `env\vars.bat` スクリプトを実行します。

特定のコンポーネントを含める

特定のコンポーネントの `env\vars.bat` スクリプトを実行するには、最初にすべてのコンポーネントの `env\vars.bat` スクリプトを除外する必要があります。その後、`setvars.bat` で実行するコンポーネントを追加し直します。次の行を定義して、すべてのコンポーネントの `env\vars.bat` スクリプトを実行から除外します。

```
default=exclude
```

そして、`setvars.bat` がインテル® oneAPI マス・カーネル・ライブラリー (oneMKL) とインテル® IPP の `env/vars.bat` スクリプトのみを 実行するには、次の行を追加します。

```
default=exclude
mkl=2021.1.1
ipp=latest
```

この例は次のように作用します。

- `setvars.bat` は、インテル® oneAPI マス・カーネル・ライブラリー (oneMKL) 2021.1.1 の `env\vars.bat` スクリプトを実行します
- `setvars.bat` は、インテル® IPP の最新バージョンの `env\vars.bat` スクリプトを実行します。
- `setvars.bat` は、そのほかのコンポーネントの `env\vars.bat` スクリプトを実行しません。

3.6.11 Microsoft* Visual Studio* で `setvars.bat` スクリプトを自動化

> **注:** Microsoft* Visual Studio* 2017 のサポートはインテル® oneAPI 2022.1 では非推奨となり、将来のリリースで削除される予定です。

`setvars.bat` スクリプトは、インテル® oneAPI ツールキットを使用するために必要な環境変数を設定します。このスクリプトは、コマンドライン開発向けに新しいターミナルウィンドウを開くたびに実行する必要があります。`setvars.bat` スクリプトはまた、Microsoft* Visual Studio* の起動時に自動的に実行することもできます。`SETVARS_CONFIG` 環境変数を使用して `setvars.bat` スクリプトにインテル® oneAPI ツール固有の設定を行うように指示できます。

`setvars.bat` が環境変数を設定する方法の詳細については、「Windows* で `setvars` および `oneapi-vars` スクリプトを使用」を参照してください。

3.6.11.1. `SETVARS_CONFIG` 環境変数の状態

`SETVARS_CONFIG` 環境変数を使用して、Microsoft* Visual Studio* のインスタンスを起動したときにインテル® oneAPI 開発環境を自動的に設定できます。環境変数には 3 つの条件と状態があります。

- 未定義 (SETVARS_CONFIG 環境変数が存在しない)
- 定義されているが空 (値を含まないか空白である)
- setvars.bat 設定ファイルを示すように定義

SETVARS_CONFIG が定義されていないと、Visual Studio* 起動時に setvars.bat スクリプトは自動実行されません。SETVARS_CONFIG 環境変数は、インテル® oneAPI インストーラーによって定義されないため、これがデフォルト動作です。

SETVARS_CONFIG に値が設定されず空白のみが含まれる場合、Visual Studio* の起動時に setvars.bat スクリプトは自動的に実行されます。この場合、setvars.bat スクリプトはシステムにインストールされている**すべて**の oneAPI ツールの環境を初期化します。setvars.bat スクリプトの実行の詳細については、「Visual Studio* コマンドラインを使用したサンプル・プロジェクトのビルドと実行」(英語) を参照してください。

SETVARS_CONFIG に setvars 設定ファイルへの絶対パスが定義されている場合、Visual Studio* の起動時に setvars.bat スクリプトは自動的に実行されます。この場合、setvars.bat スクリプトは、setvars 設定ファイルで定義されるインテル® oneAPI ツールのみの環境を初期化します。setvars 設定ファイルを作成する方法の詳細は、「setvars.bat の設定ファイルを使用」をご覧ください。

setvars 設定ファイルは任意のファイル名にでき、Visual Studio* がその場所とファイルにアクセスして読み取り可能である限り、ハードディスク上の任意の場所に保存できます (Windows* システムにインテル® oneAPI ツールをインストールする際に、Visual Studio* に追加されるプラグインは SETVARS_CONFIG のアクションを実行します。そのため、Visual Studio* は setvars 設定ファイルの場所にアクセスできる必要があります)。

setvars 設定ファイルを空のままにすると、setvars.bat スクリプトはシステムにインストールされている**すべて**のインテル® oneAPI ツールの環境を初期化します。これは、SETVARS_CONFIG 変数に空の文字列を定義するのと同じです。setvars 設定ファイルの定義の詳細については、「setvars.bat の設定ファイルを使用」を参照してください。

3.6.11.2. SETVARS_CONFIG 環境変数の定義

SETVARS_CONFIG 環境変数は、インストール中に自動的に定義されないため、Visual Studio* を起動する前に (上記の規則に従って) 手動で環境変数を定義する必要があります)。SETVARS_CONFIG 環境変数は、Windows* の SETX コマンド、または Windows* GUI ツールで Win + R キーを押して表示されるダイアログに rundll32.exe sysdm.cpl,EditEnvironmentVariables と入力して定義できます。

3.7 Linux* で setvars および oneapi-vars スクリプトを使用

バージョン 2024.0 では、統合ディレクトリー・レイアウトが実装されました。複数のツールキットのバージョンがインストールされている場合、統合レイアウトにより開発環境に特定のツールキットのバージョンの一部としてリリースされたコンポーネントのバージョンが含まれるようにする機能が実装されます。

新しい統合ディレクトリー・レイアウトでは、共通フォルダー (bin、lib、include、share など) にコンポーネントが共にインストールされていることが分かります。これらの共通フォルダーは、ツールキットのバージョン番号に基づいて命名される最上位フォルダーにあります。以下に例を示します。

```
/opt/intel/oneAPI/2024.0/
|-- bin
|-- lib
|-- include
...など...
```

2024.0 以前に使用されていたディレクトリー・レイアウトは、新規および既存のインストールで引き続きサポートされます。以前のレイアウトは、コンポーネント・ディレクトリー・レイアウトと呼ばれます。コンポーネント・ディレクトリー・レイアウトまたは統合ディレクトリー・レイアウトを使用するオプションが追加されました。

3.7.1 コンポーネント・ディレクトリー・レイアウトと統合ディレクトリー・レイアウトの違い

ほとんどのインテル® oneAPI コンポーネントのフォルダーには、oneAPI 開発作業をサポートするそれぞれのコンポーネントに必要な環境変数を設定する env/vars.sh スクリプトが含まれています。例えば、デフォルトのインストールでは、Linux* のインテル® インテグレーテッド・パフォーマンス・プリミティブ (インテル® IPP) の vars スクリプトは、/opt/intel/oneapi/ipp/latest/env/vars.sh に配置されます。このパスは、env/vars 環境変数設定スクリプトを含むすべてのインテル® oneAPI コンポーネントで共有されます。

コンポーネント・ディレクトリー・レイアウトでは、各コンポーネント向けの env/vars スクリプトは、直接またはまとめて呼び出すことができます。まとめて呼び出すには、oneAPI インストール・ディレクトリーにある setvars.sh スクリプトを使用します。これは、Linux* マシンのデフォルトのインストールでは、/opt/intel/oneapi/setvars.sh にあります。

統合ディレクトリー・レイアウトは、開発環境の初期化に env/vars スクリプトを使用しません。代わりに、各コンポーネントは、コンポーネントに共通の共有フォルダーに包括されます。つまり、各コンポーネントは、ヘッダーファイルを単一の共通インクルード・フォルダーに提供し、そのライブラリー・ファイルを単一の共通 lib フォルダーに提供することになります。

3.7.2 統合ディレクトリー・レイアウトの利点

統合ディレクトリー・レイアウトを使用することで setvars 設定ファイルを構成して維持したり、複数の oneAPI ツールキットをインストールして個別に環境を作成する必要がなく、異なるツールキットのバージョンの切り替えがはるかに容易になりました。また、一部の Windows* 開発者によって厄介な問題である Windows* 開発システムの環境変数、特に PATH 変数の長さを制限するのにも役立ちます。

統合ディレクトリー・レイアウトの環境変数は、一括でのみ設定できます。環境変数を初期化するには oneapi-vars.sh スクリプトを使用します。Linux* 上のデフォルトの統合ディレクトリー・レイアウトのインストールでは、スク

リプトは /opt/intel/oneapi/<toolkit-version>/oneapi-vars.sh にあります。<toolkit-version> は、インストールした oneAPI ツールキットのバージョン番号に対応します。以下に例を示します。

```
/opt/intel/oneapi/2024.0/oneapi-vars.sh
/opt/intel/oneapi/2024.1/oneapi-vars.sh
```

引数なしで setvars.sh スクリプトを source すると、システムにインストールされているすべての <コンポーネント>/latest/env/vars.sh スクリプトが source されます。これらのスクリプトを source した後、env コマンドを使用して環境変数を確認できます。

引数なしで oneapi-vars.sh スクリプトを実行すると、スクリプトが配置されているバージョンの環境が構成されます。また、統合ディレクトリーのインストースの一部であるオプションの /opt/intel/oneapi/<toolkit-version>/etc/<component>/vars.sh スクリプトも source されます。このスクリプトで変更された環境変数は、oneapi-vars.sh スクリプトの実行後に、Linux* の env コマンドで確認できます。

> **注:** setvars.sh / oneapi-vars スクリプト (または個別の vars.sh スクリプト) により変更された環境は永続的ではありません。これらの変更は、setvars.sh / oneapi-vars 環境スクリプトが source されたターミナルセッションでのみ有効です。

oneapi-vars.sh スクリプトの仕組みの詳細は、「統合ディレクトリー・レイアウトの環境の初期化」を参照してください。

3.7.3 コマンドライン引数

setvars.sh スクリプトはいくつかのコマンドライン引数をサポートしており、--help オプションで引数の一覧を表示できます。

コンポーネント・ディレクトリー・レイアウト

システム全体でのインストール:

```
$ . /opt/intel/oneapi/setvars.sh --help
```

プライベート環境でのインストール:

```
$ . ~/intel/oneapi/setvars.sh --help
```

統合ディレクトリー・レイアウト

システム全体でのインストール:

```
$ . /opt/intel/oneapi/<version>/oneapi-vars.sh --help
```

プライベート環境でのインストール：

```
$ . ~/intel/oneapi/<version>/oneapi-vars.sh --help
```

`--config=file` 引数と `setvars.sh` / `oneapi-vars` スクリプトから呼び出される `vars.sh` スクリプトへの追加引数をインクルードする機能を使用して、環境設定をカスタマイズできます。`--config=file` オプションは、`setvars.sh` スクリプトでのみサポートされます。

`--config=file` 引数は、特定のインテル® oneAPI コンポーネントの環境の初期化機能を提供するとともに、特定のバージョンの環境を初期化することもできます。例えば、インテル® IPP とインテル® oneAPI マス・カーネル・ライブラリー (oneMKL) の環境のみを設定するには、これら 2 つのインテル® oneAPI コンポーネントの `vars.sh` 環境スクリプトのみを呼び出すように `setvars.sh` / `oneapi-vars` スクリプトに指示する設定ファイルを渡します。詳細と利用例については、「Linux* または macOS* で setvars.sh 設定ファイルを使用」をご覧ください。

`setvars.sh` / `oneapi-vars` のヘルプメッセージに記載されていないコマンドライン引数は、`vars.sh` スクリプトに渡されます。つまり、`setvars.sh` / `oneapi-vars` スクリプトが認識できない引数は、コンポーネントの `vars.sh` スクリプトで使用されるものと見なし、それらの引数をすべてのコンポーネントの `vars.sh` スクリプトに渡します。最もよく使用される追加の引数は、`ia32` と `intel64` です。これらは、インテル® コンパイラー、インテル® IPP、インテル® oneAPI マス・カーネル・ライブラリー (oneMKL)、およびインテル® oneAPI スレッディング・ビルディング・ブロック (oneTBB) ライブラリーでアプリケーションのターゲット・アーキテクチャーを指示するために使用されます。

個々の `vars.sh` スクリプトを調べて、受け入れるコマンドライン引数があればそれを決定します。

3.7.4 実行方法

コンポーネント・ディレクトリー・レイアウト

```
$ source <install-dir>/setvars.sh
```

統合ディレクトリー・レイアウト

```
$ source <install-dir>/<toolkit-version>/oneapi-vars.sh
```

注: csh など非 POSIX* シェルを使用する場合、次のコマンドを使用します。

コンポーネント・ディレクトリー・レイアウト

```
$ bash -c 'source <install-dir>/setvars.sh ; exec csh'
```

統合ディレクトリー・レイアウト

```
$ bash -c 'source <install-dir>/<toolkit-version>/oneapi-vars.sh ;
exec csh'
```

環境変数が正しく設定されていると、次のような確認メッセージが表示されますs。

エラーメッセージが表示された場合は、インテル® oneAPI ツールキットの診断ユーティリティーを使用してトラブルシューティングを行ってください。このユーティリティーは、不足している依存関係や権限エラーを検出するためシステムチェックを行います。詳細はこちら (英語) を参照してください。

または、`modulefiles` スクリプトを使用して開発環境をセットアップします。`modulefiles` スクリプトは、すべての Linux* シェルで動作します。

コンポーネントのリストとそれらのコンポーネントのバージョンを調整する場合は、`setvars` 設定ファイルを使用して開発環境をセットアップします。

3.7.5 複数の実行

各コンポーネントの env/vars.sh スクリプトの多くは、PATH、CPATH、およびそのほかの環境変数に変更を加えるため、最上位の setvars.sh / oneapi-vars スクリプトは同じセッションで同じ vars.sh を複数回呼び出すことはできません。これは、特に $PATH 環境変数が原因で環境変数の文字数が長くなりすぎないようにします。

これを強制するには、setvars.sh / oneapi-vars に --force オプションを指定します。この例では、ユーザーが setvars.sh / oneapi-vars を 2 回実行しています。setvars.sh / oneapi-vars がすでに実行されているため、2 回目の実行は停止します。

コンポーネント・ディレクトリー・レイアウト

```
$ source <install-dir>/setvars.sh
.. code-block:: initializing oneAPI environment ...
(SNIP: lot of output)
.. code-block:: oneAPI environment initialized ::
```

```
$ source <install-dir>/setvars.sh
.. code-block:: WARNING: setvars.sh has already been run.Skipping re-execution.
To force a re-execution of setvars.sh, use the '--force' option.
Using '--force' can result in excessive use of your environment variables
```

次は、ユーザーが setvars.sh --force を実行し、初期化が成功した例です。

```
$ source <install-dir>/setvars.sh --force
.. code-block:: initializing environment ...(SNIP: lot of output)
.. code-block:: oneAPI environment initialized ::
```

統合ディレクトリー・レイアウト

```
$ source <install-dir>/<version>/oneapi-vars.sh
.. code-block:: initializing oneAPI environment ...
(SNIP: lot of output)
.. code-block:: oneAPI environment initialized ::
```

```
$ source <install-dir>/<toolkit-version>/oneapi-vars.sh
 .. code-block:: WARNING: setvars.sh has already been run. Skipping re-execution.
    To force a re-execution of setvars.sh, use the '--force' option.
    Using '--force' can result in excessive use of your environment variables
```

3 番目のインスタンスでは、ユーザーが oneapi-vars.sh --force を実行すると、初期化が成功します。

```
$ source <install-dir>/ ``oneapi-vars.sh`` --force
 .. code-block:: initializing oneAPI environment ...
(SNIP: lot of output)
 .. code-block:: oneAPI environment initialized ::
```

3.7.6 統合ディレクトリー・レイアウトの環境変数

統合ディレクトリー・レイアウトは、2024.0 リリースで実装されました。この環境に不慣れな開発者は、「Linux* で setvars および oneapi-vars スクリプトを使用」を参照してください。

統合ディレクトリー・レイアウト環境の初期化は、setvars.sh スクリプトではなく oneapi-vars スクリプトによって行われます。oneapi-vars の使い方は setvars と似ていますが、微妙な違いがあります。

setvars スクリプトと oneapi-vars の主な違いは、setvars は環境変数 (ONEAPI_ROOT を除く) を定義しませんが、oneapi-vars は共通の環境変数を定義することです。

コンポーネント・ディレクトリー・レイアウトは、各コンポーネントが機能するために必要な環境変数を定義します。例えば、コンポーネント・ディレクトリー・レイアウトでは、各コンポーネントはリンク可能なライブラリー・フォルダーを LD_LIBRARY_PATH に、そしてヘッダーを CPATH などに追加します。コンポーネントは、常に次の場所にある vars スクリプトを介してこれを実行します。

```
$ %ONEAPI_ROOT%/<component-name >/<component-name>/env/vars.sh
```

統合ディレクトリー・レイアウトは、外部向けの include、lib、bin フォルダーを共有フォルダーに統合します。最上位レベルの oneapi-vars スクリプトは、これらの共通フォルダーを検出するのに必要な環境変数を定義します。例えば、setvars は LD_LIBRARY_PATH を $ONEAPI_ROOT/lib として定義し、CPATH を $ONEAPI_ROOT/include と定義します。

Modulefile は 2024.0 リリースでも引き続きサポートされ、setvars.sh を使用して環境設定を初期化する代わりに使用できます。Modulefile スクリプトは Linux* でのみサポートされます。

3.7.7 ONEAPI_ROOT 環境変数

ONEAPI_ROOT 環境変数は、スクリプトが実行されるときに最上位の setvars.sh および oneapi-vars.sh によって設定されます。ONEAPI_ROOT 環境変数がすでに設定されている場合、setvars.sh または oneapi-vars.sh スクリプトはそれを上書きします。この変数は、oneapi-cli サンプルブラウザーと Eclipse* および Visual Studio* Code サンプルブラウザーによって使用され、インテル® oneAPI ツールとコンポーネントの検出、および SETVARS_CONFIG 機能が有効である場合に setvars.sh スクリプトを検出するのに役立ちます。SETVARS_CONFIG 機能の詳細については、「Eclipse* で setvars.sh スクリプトを自動化」をご覧ください。

2024.0 リリースでは、インストーラーでは、インストーラーは ONEAPI_ROOT 環境変数を追加しません。これをデフォルト環境に追加するには、ローカルシェルの初期化ファイル (.bashrc など) または /etc/environment ファイルで ONEAPI_ROOT 変数を定義します。

3.7.8 Linux* で `setvars.sh` 設定ファイルを使用

注: 2024 リリース以降では、macOS* はインテル® oneAPI ツールキットおよびコンポーネントでサポートされなくなりました。oneAPI スレッディング・ビルディング・ブロック (oneTBB) やインテル® Implicit SPMD Program Compiler など、いくつかのオープンソース・プロジェクトは、引き続き Apple シリコン上の macOS* をサポートします。これらのツール開発の貢献と協力は歓迎します。

Linux* で環境を設定するには、次の 2 つの方法があります。

- このページで示すように、`setvars.sh` 設定ファイルを使用します。
- `modulefile` を使用します。

注: 設定ファイルは、コンポーネント・ディレクトリー・レイアウトの `setvars.sh` でのみ使用できます。統合ディレクトリー・レイアウトは、`oneapi-vars.sh` を使用しますが、これは設定ファイルをサポートしません。レイアウトの詳細については、「Linux* で setvars および oneapi-vars スクリプトを使用」を参照してください。

`setvars.sh` スクリプトは、それぞれの oneAPI ディレクトリーにある `<install-dir>/latest/env/vars.sh` スクリプトを `source` することで、インテル® oneAPI ツールキットで使用する環境変数を設定します。`setvars.sh` スクリプトを自動的に `source` するように Linux* システムを設定しない限り、新しいターミナルウィンドウを開くか Eclipse* またはそのほかの C/C++ IDE やエディターを起動する前に `source` する必要があります。詳細は、「システムの設定」(英語) を参照してください。

次に設定ファイルを使用して環境変数を管理する方法を説明します。

3.7.8.1. バージョンと構成

一部のインテル® oneAPI ツールは複数バージョンのインストールがサポートされます。複数バージョンのサポートするツールのディレクトリー構造は次のようになります。

```
intel/oneapi/compiler/
|-- 2021.1.1
|-- 2021.2.0
`-- latest -> 2021.2.0
```

例: 複数バージョンと環境変数

```
$ ls -l intel/oneapi/compiler/
total 8
drwxr-xr-x 8 ubuntu ubuntu 4096 Nov  9  2020 2021.1.1/
drwxrwxr-x 8 ubuntu ubuntu 4096 Apr  9 10:06 2021.2.0/
lrwxrwxrwx 1 ubuntu ubuntu    8 Apr  9 10:06 latest -> 2021.2.0/
$
```

すべてのツールには、そのコンポーネントの最新バージョンのインストール先を示す `latest` という名前のシンボリック・リンクがあります。`latest/env/` ディレクトリーにある `vars.sh` スクリプトは、`setvars.sh` によって `source` されます (デフォルト)。

必要に応じて、設定ファイルを使用して特定のディレクトリーを示すよう `setvars.sh` をカスタマイズできます。

3.7.8.2. --config パラメーター

最上位の `setvars.sh` スクリプトは、カスタム `config.txt` ファイルを指定する `--config` パラメーターを受け入れます。

```
$ source <install-dir>/setvars.sh --config="full/path/to/your/config.txt"
```

設定ファイルは任意の名前にすることができます。複数の設定ファイルを作成して、さまざまな開発環境やテスト環境を設定できます。例えば、最新バージョンのライブラリーを古いバージョンのコンパイラーでテストしたいこともあります。そのような場合に、`setvars` 設定ファイルを使用して環境を管理できます。

3.7.8.3. 設定ファイルの例

以下に簡単な設定ファイルの例を示します。

最新のコンポーネントをすべてロード

```
mkl=1.1
dldt=exclude
```

ただし以下のコンポーネントは除外

```
default=exclude
mkl=1.0
ipp=latest
```

設定テキストファイルは次の要件に従う必要があります。

- 改行で区切られたテキストファイル
- 各行は、`key=value` のペアで構成されます

- `key` には、oneAPI ディレクトリーの最上位 (`$ONEAPI_ROOT` ディレクトリーにあるフォルダー) のコンポーネント名を指定します。同じ `key` が設定ファイルに複数定義されると、最後の `key` が優先されそれ以外は無視されます。
- `value` には、コンポーネント・ディレクトリーの最上位にあるバージョン・ディレクトリー名を指定します。これには、コンポーネント・ディレクトリーのレベルに存在する可能性があるショートカット (`latest` など) が含まれます。
    - また、`value` は `exclude` にすることもできます。これは、指定された `key` の 環境変数 スクリプトを `setvars.sh` スクリプトで `source` しないことを意味します。

`key=value` を `default=exclude` にすると特別な意味を持ちます。これは、設定ファイルに定義されているものを除き、それ以外のすべての `env/vars.sh` スクリプトの `source` を除外します。以下に例を示します。

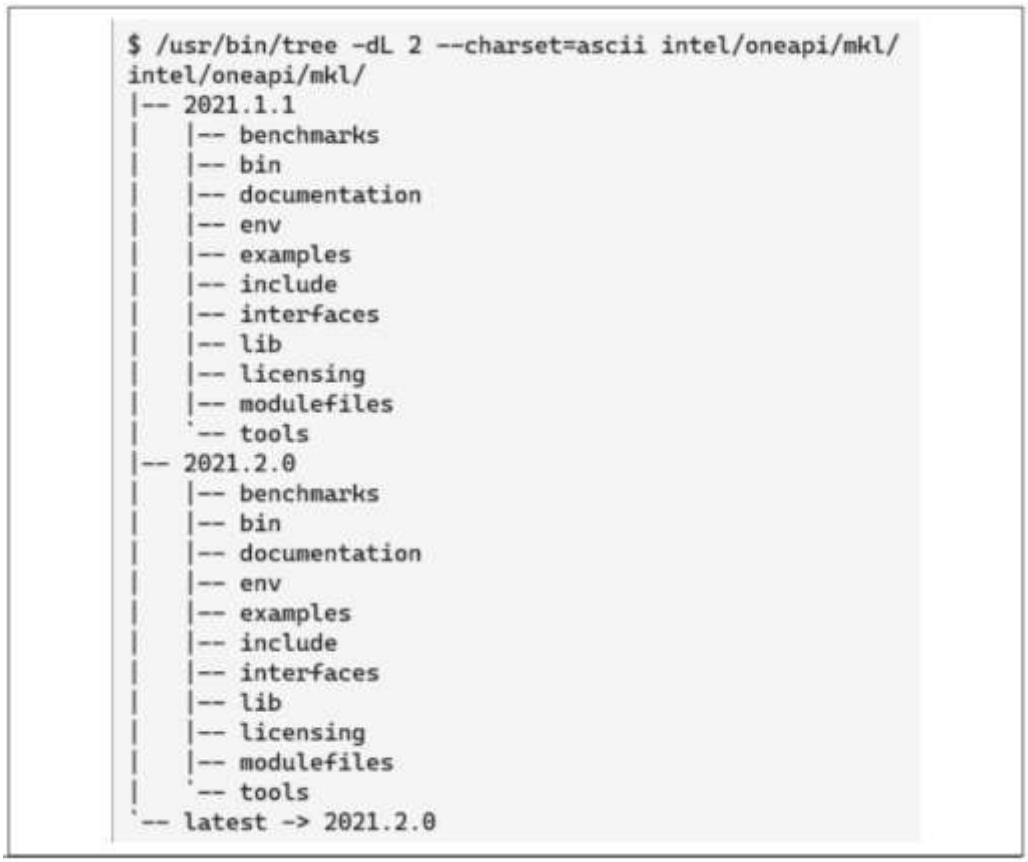
3.7.8.4. 設定ファイルのカスタマイズ

設定ファイルを使用して、特定のコンポーネントを除外したり、特定のバージョンを含めたり、特定のコンポーネントのバージョンのみを含めることができます。これには、設定ファイルの `default=exclude` 行を変更します。

デフォルトでは、`setvars.sh` は最新 (`latest`) のバージョンに対応する `env/vars.sh` スクリプトを処理します。

例えば、2 つのバージョン 2021.1.1 と 2021.2.0) のインテル® oneAPI マス・カーネル・ライブラリー (oneMKL) がインストールされていると仮定します。最新のバージョンを示す `symlink` は 2021.2.0 であるため、デフォルトでは `setvars.sh` は `mkl` ディレクトリーの 2021.2.0 の `vars.sh` スクリプトを実行します。

2 つのバージョンの oneMKL がインストールされている場合



```
$ /usr/bin/tree -dL 2 --charset=ascii intel/oneapi/mkl/
intel/oneapi/mkl/
|-- 2021.1.1
|   |-- benchmarks
|   |-- bin
|   |-- documentation
|   |-- env
|   |-- examples
|   |-- include
|   |-- interfaces
|   |-- lib
|   |-- licensing
|   |-- modulefiles
|   `-- tools
|-- 2021.2.0
|   |-- benchmarks
|   |-- bin
|   |-- documentation
|   |-- env
|   |-- examples
|   |-- include
|   |-- interfaces
|   |-- lib
|   |-- licensing
|   |-- modulefiles
|   `-- tools
`-- latest -> 2021.2.0
```

特定のバージョンを指定

`setvars.sh` に `<install-dir>/mkl/2021.1.1/env` スクリプトを `source` するように直接記述するには、設定ファイルに `mkl=2021.1.1` を追加します。

これにより、`setvars.sh` は、`mkl` ディレクトリー内の 2021.1.1 フォルダーにある `env/vars.sh` スクリプトを `source` するようになります。インストールされている `mkl` 以外のコンポーネントでは、`setvars.sh` は最新バージョンのフォルダーにある `env/vars.sh` スクリプトを `source` します。

特定のコンポーネントを除外

コンポーネントを除外する構文は次のようになります。

```
<key>=exclude
```

例えば、インテル® IPP を除外して、2021.1.1 の インテル® oneAPI マス・カーネル・ライブラリー (oneMKL) を含めるには次のようにします。

```
Mkl=2021.1.1
ipp=exclude
```

この例は次のように作用します。

- `Setvars.sh` は、インテル® oneAPI マス・カーネル・ライブラリー (oneMKL) 2021.1.1 の `env/vars.sh` スクリプトを `source` します。
- `setvars.sh` は、インテル® IPP の `env/vars.sh` スクリプトを `source` しません。
- `setvars.sh` は、そのほかのコンポーネントの最新バージョンの `env/vars.sh` スクリプトを `source` します。

特定のコンポーネントを含める

特定のコンポーネントの `env/vars.sh` スクリプトを `source` するには、最初にすべてのコンポーネントの `env/vars.sh` スクリプトを除外する必要があります。その後、`setvars.sh` で `source` するコンポーネントを追加し直します。次の行を定義して、すべてのコンポーネントの `env/vars.sh` スクリプトを `source` から除外します。

```
default=exclude
```

例えば、`setvars.sh` がインテル® oneAPI マス・カーネル・ライブラリー (oneMKL) とインテル® IPP コンポーネントの `env/vars.sh` スクリプトのみを `source` するようにするには、次の設定ファイルを使用します。

```
default=exclude
mkl=2021.1.1
ipp=latest
```

この例は次のように作用します。

- `setvars.sh` は、インテル® oneAPI マス・カーネル・ライブラリー (oneMKL) 2021.1.1 の `env/vars.sh` スクリプトを `source` します。
- `setvars.sh` は、インテル® IPP の最新バージョンの `env/vars.sh` スクリプト `source` します。
- `setvars.sh` は、そのほかのコンポーネントの `env/vars.sh` スクリプトを `source` しません。

3.7.9 Eclipse* で `servars.sh` スクリプトを自動化

`setvars.sh` スクリプトは、インテル® oneAPI ツールキットを使用するために必要な環境変数を設定します。このスクリプトは、コマンドライン開発向けに新しいターミナルウィンドウを開くたびに実行する必要があります。`setvars.sh` スクリプトは、Eclipse* の起動時に自動的に実行することもできます。`SETVARS_CONFIG` 環境変数を使用して、`setvars.sh` スクリプトにインテル® oneAPI ツール固有の設定を行うように指示できます。

`setvars.sh` で環境変数を設定する方法の詳細は、「Linux* で `setvars` および `oneapi-vars` スクリプトを使用」を参照してください。

3.7.10 SETVARS_CONFIG 環境変数の状態

SETVARS_CONFIG 環境変数を使用して、C/C++ 開発者向けの Eclipse* IDE インスタンスを起動したときにインテル® oneAPI 開発環境を自動的に設定できます。環境変数には 3 つの条件と状態があります。

- 未定義 (SETVARS_CONFIG 環境変数が存在しない)
- 定義されているが空 (値を含まないか空白である)
- setvars.sh 設定ファイルを示すように定義

SETVARS_CONFIG に値が設定されていない (空白のみが含まれる) 場合、Eclipse* の起動時に setvars.sh スクリプトは自動的に実行されます。この場合、setvars.sh スクリプトはシステムにインストールされている**すべて**の oneAPI ツールの環境を初期化します。setvars.sh スクリプト実行の詳細については、「Eclipse* 使用したサンプル・プロジェクトのビルドと実行」(英語) を参照してください。

SETVARS_CONFIG に setvars 設定ファイルへの絶対パスが定義されている場合、Eclipse* の起動時に setvars.sh スクリプトは自動的に実行されます。この場合、setvars.sh スクリプトは、setvars 設定ファイルで定義されるインテル® oneAPI ツールのみの環境を初期化します。setvars 設定ファイルを作成する方法の詳細は、「Linux* で setvars.sh および oneapi-vars スクリプトを使用」をご覧ください。

> **注:** Eclipse* でのデフォルトの SETVARS_CONFIG の動作は、Windows* の Visual Studio* で説明されている動作とは異なります。Eclipse* を起動すると、setvars.sh スクリプトは常に自動的に実行されます。Windows* で Visual Studio* を起動すると、SETVARS_CONFIG 環境変数が定義されている場合にのみ setvars.bat スクリプトは自動的に実行されます。

setvars 設定ファイルは任意の名前で作成でき、そのファイルが Eclipse* からアクセスおよび読み取り可能である限りハードディスク上のどこにでも保存できます (Linux* システムにインテル® oneAPI ツールをインストールしたときに Eclipse* に追加されたプラグインが SETVARS_CONFIG のアクションを実行するため、Eclipse* は setvars 設定ファイルにアクセスできる必要があります)。

setvars 設定ファイルを空のままにすると、setvars.sh スクリプトはシステムにインストールされている**すべて**のインテル® oneAPI ツールの環境を初期化します。これは、SETVARS_CONFIG 変数に空の文字列を定義するのと同じです。setvars 設定ファイルの作成方法については、「Linux* で setvars.sh および oneapi-vars スクリプトを使用」を参照してください。

3.7.11 SETVARS_CONFIG 環境変数の定義

SETVARS_CONFIG 環境変数はインストール中に自動的に定義されないため、Eclipse* を起動する前に (前述の方法で) 環境変数を追加する必要があります。SETVARS_CONFIG 環境変数には、以下を含むさまざまな場所を定義できます。

- /etc/environment
- /etc/profile
- ~/.bashrc

上記の例は、Linux* システムで環境変数を定義する一般的な場所です。SETVARS_CONFIG 環境変数に定義する場所は、システムとニーズによって異なります。

3.8 Linux* で modulefile を使用

modulefile で環境を設定すると、使用するコンポーネントの的確なバージョンを指定できます。

Linux* で環境を設定するには、次の 2 つの方法があります。

- このページで説明されているように modulefile を使用します。
- setvars.sh 設定ファイルを使用します。

ほとんどのインテル® oneAPI コンポーネントのフォルダーには、それぞれのコンポーネントに必要な環境変数を設定する modulefile のスクリプトが含まれています。modulefile を使用して、setvars.sh スクリプトに代わって開発環境を設定できます。modulefile では引数がサポートされていないため、複数の設定 (32 ビットや 64 ビットの設定など) をサポートするインテル® oneAPI ツールおよびライブラリーでは複数の modulefile を使用します。

注: modulefile は、コンポーネント・ディレクトリー・レイアウトの setvars.sh スクリプトでのみ使用できます。統合ディレクトリー・レイアウトでは oneapi-vars.sh を使用しますが、modulefile はサポートされません。レイアウトの詳細については、「Linux* で setvars および oneapi-vars スクリプトを使用」を参照してください。

注: インテル® oneAPI ツールキットで提供される modulefile は、Tcl 環境モジュール (Tmod) および Lua 環境モジュール (Lmod) と互換性があり、次のバージョンがサポートされます。

- Tmod 4.0
- Tcl バージョン 8.4
- Lmod バージョン 8.7.31

次のコマンドを使用して、システムにインストールされているバージョンを確認します。

```
$ module --version
```

各 modulefile は、実行時にシステムの Tcl バージョンが適切であるか自動的に確認します。

```
$ tclsh
$ puts $tcl_version
8.6
$ exit
```

各 modulefile は、実行時にシステムの Tcl バージョンを自動的に検出しますが、Tmod のバージョンは確認しません。

modulefile のバージョンがサポートされていない場合、回避策があります。詳細については、「インテル開発ツールにおける環境モジュールの利用」(英語) を参照してください。

oneAPI 2024.0 リリースでは、インテル® クラシック・コンパイラーが廃止されているため、icc の modulefile は削除されました。代替えとして、インテル® oneAPI C/C++ コンパイラー (icx および ifx) を使用してください。Ifort は引き続き使用できますが、インテル® oneAPI Fortran コンパイラー (ifx) を使用することを推奨します。

oneAPI の modulefile スクリプトは、それぞれのコンポーネント・ディレクトリーにある modulefiles フォルダーにあります (個々の vars スクリプトの配置方法と同様)。例えば、2024.0 以降の oneAPI ツールキットのデフォルトのインストールでは、コンパイラーの modulefile スクリプトは、/opt/intel/compiler/<component-version>/etc/modulefiles/ ディレクトリーにあります。2023 以前の oneAPI ツールキットでは、コンパイラーの modulefile スクリプトは /opt/intel/compiler/<component-version>/modulefiles/ ディレクトリーにあります。

注: <component-version> は、コンポーネント (ライブラリーまたはツール) のバージョン番号です。開発システム上に複数のコンポーネントが同時にインストールされている場合があります (例えば、compiler/2023.1、compiler/2023.2、compiler/2024.0 など)。コンポーネントが含まれるツールキットのバージョンは、そのツールキットで提供されるコンポーネントのバージョンとは異なることがあります。

インテル® oneAPI コンポーネントのフォルダーがどのように展開されているかにより、インテル® oneAPI の modulefile をインストール先で直接使用することが難しい場合があります。そのため、oneAPI インストール・フォルダーには特殊な modulefiles-setup.sh スクリプトが用意されており、インテル® oneAPI の modulefile を簡単に操作できます。デフォルトのインストールでは、設定スクリプトは /opt/intel/oneapi/modulefiles-setup.sh に配置されます。

modulefiles-setup.sh スクリプトは、インテル® oneAPI のインストールに含まれるすべての modulefile スクリプトを検索し、それらのフォルダーを単一ディレクトリーに保存します。

これらのバージョン管理された modulefile スクリプトは、それぞれ modulefiles-setup.sh スクリプトで配置された modulefile を指すシンボリック・リンクです。各コンポーネント・フォルダーには、少なくとも latest (最新) バージョンの modulefile が含まれ、バージョンを指定しなくてもデフォルトで最新のコンポーネントをロードできます。modulefiles-setup.sh スクリプトの実行時に --ignore-latest オプションを使用すると、module load コマンドでバージョンが指定されていない場合、最も高い semver バージョンの modulefile がロードされます。

2024.0 以降のツールキットのインストールでは、統合ディレクトリー・レイアウト内にツールキットのバージョンに関連するすべてのコンポーネントのモジュールファイルを含む modulefiles フォルダーを持ちます。例えば、インテル®

oneAPI ベース・ツールキット 2024.0 をデフォルトの場所にインストールすると、事前構成されたすぐに利用できる 2024.0 のベース・ツールキットのモジュールファイルのコレクション
が `/opt/intel/oneapi/2024.0/etc/modulefiles` フォルダーに配置されます。これは、このフォルダーを MODULEPATH 環境変数に追加するか、`module use` コマンドを使用して事前構成された `modulefile` フォルダーを指定して、`modulefiles` に追加できます。インテル® oneAPI HPC ツールキットの 2024.0 をインストールすると、2 つのツールキットのバージョンが同じであるため (ここでは 2024.0)、追加のモジュールファイルは同じ場所に配置されます。

事前構成されたツールキットの `modulefiles` フォルダー内には、そのツールキットのバージョンに含まれる 64 ビット・コンポーネントのモジュールファイルをロードする `oneapi` という名前のモジュールファイルがあります。`oneapi` モジュールファイルを使用すると、個別のモジュールファイルを使用する必要はありません。また、`oneapi` モジュールファイルは、`modulefiles-setup.sh` スクリプトで作成されるフォルダー内では期待するように動作しません。

2024.0 以降のツールキットで提供される modulefiles は、必要となる modulefiles を自動的にロードしないことに注意してください。代わりに、要求される modulefile は、適切な modulefile がロードされているかをチェックして、ロードされていない場合はエラーメッセージを表示します。必要となる modulefile の事前ロードは開発者に任せられます。oneap コンビニエンス modulefile は、すべての必要な modulefile を自動的にロードし、それらを必要とする modulefile をロードする前にロードします。

3.8.1 `modulefiles` ディレクトリーの作成

`modulefiles-setup.sh` スクリプトを実行します。

デフォルトでは、`modulefiles-setup.sh` スクリプトは、インテル® oneAPI ツールキットのインストール・フォルダーに `modulefiles` という名前のフォルダーを作成します。インストール・フォルダーが書き込み可能でない場合、`--outputdir=`<フォルダーのパス> オプションを使用して、書き込み可能な場所に `modulefiles` フォルダーを作成します。`modulefiles-setup.sh` スクリプトの詳細については、`modulefiles-setup.sh -help` を入力して確認できます。

`modulefiles-setup.sh` スクリプトが実行されると、次のような階層が最上位の `modulefiles` フォルダーに作成されます (`modulefiles` の正確なリストはインストールによって異なります)。この例では、インテル® Advisor 環境を設定する 1 つの `modulefile` と、コンパイラー環境を設定する 2 つの `modulefile` があります (コンパイラーの `modulefile` は、すべてのインテル® コンパイラーの環境を設定します)。最新のシンボリック・リンクをたどると、`semver` の規則に従って最上位バージョンの `modulefile` を指します。

```
|-- advisor
|   |-- 2021.2.0 -> /home/ubuntu/intel/oneapi/advisor/2021.2.0/modulefiles/advisor
|   `-- latest -> /home/ubuntu/intel/oneapi/advisor/latest/modulefiles/advisor
|-- ccl
|   |-- 2021.1.1 -> /home/ubuntu/intel/oneapi/ccl/2021.1.1/modulefiles/ccl
|   |-- 2021.2.0 -> /home/ubuntu/intel/oneapi/ccl/2021.2.0/modulefiles/ccl
|   `-- latest -> /home/ubuntu/intel/oneapi/ccl/latest/modulefiles/ccl
|-- clck
|   |-- 2021.1.1 -> /home/ubuntu/intel/oneapi/clck/2021.1.1/modulefiles/clck
|   `-- latest -> /home/ubuntu/intel/oneapi/clck/latest/modulefiles/clck
|-- compiler
|   |-- 2021.1.1 -> /home/ubuntu/intel/oneapi/compiler/2021.1.1/modulefiles/compiler
|   |-- 2021.2.0 -> /home/ubuntu/intel/oneapi/compiler/2021.2.0/modulefiles/compiler
|   `-- latest -> /home/ubuntu/intel/oneapi/compiler/latest/modulefiles/compiler
|-- compiler-rt
|   |-- 2021.1.1 -> /home/ubuntu/intel/oneapi/compiler/2021.1.1/modulefiles/compiler-rt
|   |-- 2021.2.0 -> /home/ubuntu/intel/oneapi/compiler/2021.2.0/modulefiles/compiler-rt
|   `-- latest -> /home/ubuntu/intel/oneapi/compiler/latest/modulefiles/compiler-rt
|-- compiler-rt32
|   |-- 2021.1.1 -> /home/ubuntu/intel/oneapi/compiler/2021.1.1/modulefiles/compiler-rt32
|   |-- 2021.2.0 -> /home/ubuntu/intel/oneapi/compiler/2021.2.0/modulefiles/compiler-rt32
|   `-- latest -> /home/ubuntu/intel/oneapi/compiler/latest/modulefiles/compiler-rt32
|-- compiler32
|   |-- 2021.1.1 -> /home/ubuntu/intel/oneapi/compiler/2021.1.1/modulefiles/compiler32
|   |-- 2021.2.0 -> /home/ubuntu/intel/oneapi/compiler/2021.2.0/modulefiles/compiler32
|   `-- latest -> /home/ubuntu/intel/oneapi/compiler/latest/modulefiles/compiler32
```

ここで、`MODULEFILESPATH` を更新して、`modulefiles-setup.sh` スクリプトで作成された新しい `modulefiles` フォルダーに含めるか、`moduleuse <folder_name>` コマンドを実行します。

3.8.2 システムに Tcl modulefile 環境をインストール

次の手順は、Ubuntu* で環境モジュール・ユーティリティーを実行する例を示しています。モジュール・ユーティリティーのインストールと設定の詳細については、http://modules.sourceforge.net/ (英語) を参照してください。

環境を設定

```
$ sudo apt update
$ sudo apt install tcl
$ sudo apt install environment-modules
```

`tclsh` のローカルコピーが新しいものであることを確認します (サポートされているバージョンについては、このページの最初を参照してください)。

```
$ echo 'puts [info patchlevel] ; exit 0' | tclsh 8.6.8
```

`module` のインストールをテストするには、`module` エイリアスを初期化します。

```
$ source /usr/share/modules/init/sh
$ module
```

注: POSIX* 互換シェルの初期化は、上記の `source` コマンドで機能するはずです。それ以外のシェル固有の `init` スクリプトは、`/usr/share/modules/init/` フォルダーにあります。詳細については、各フォルダーと `man module` の初期化セクションをご覧ください。module の init フォルダーは、Linux* ディストリビューションによって異なることに注意してください。

`init` スクリプト (`.../modules/init/sh`) で `module` エイリアスを `source` して、`module` コマンドが利用できるようにします。これにより、システムは次のセクションに示す `module` コマンドを使用できます。

3.8.3 `modulefiles-setup.sh` スクリプトの使用

次のことが前提とされています。

- Linux* 開発システムに `tclsh` がインストールされている
- システムに環境モジュール・ユーティリティー (`module` など) がインストールされている
- `init` コマンドで `.../modules/init/sh` (または等価なシェル) が `source` されている
- oneAPI 開発に必要なインテル® oneAPI ツールキットがインストールされている

インストール・ディレクトリーに移動して tbb をロードします。

```
$ cd <oneapi-root-folder>  # oneapi_root インストール・ディレクトリーに移動する
$ ./modulefiles-setup.sh   # modulefile 設定スクリプトを実行する
$ module use modulefiles   # 上記で作成した modulefiles フォルダーを使用する
$ module avail             # tbb/X.Y などが表示される
$ module load tbb          # tbb/X.Y モジュールをロード
$ module list              # ロードした tbb/X.Y モジュールをリスト
$ module unload tbb        # 環境から tbb/X.Y の変更を削除
$ module list              # tbb/X.Y 環境変数モジュールがリストされなくなる
```

アンロードの前に、`env` コマンドを使用して環境を検証し、ロードした `modulefile` で変更された部分を探します。

```
$ env | grep -i "intel"
```

例えば、次のコマンドを実行すると、ロードした `tbb modulefile` で変更された環境の一部分が表示されます (`modulefile` を調べてすべての変更を確認します)。

tbb をアンロード

```
$ module unload tbb              # tbb/X.Y の変更を環境から削除する
$ module list                    # tbb/X.Y env var モジュールをリストしないようにします
```

注: modulefile はスクリプトですが、ユーザーがインストールおよび保守する module インタープリターによってロードおよび解釈されるため、x (実行可能) 権限を設定する必要はありません。インテル® oneAPI ツールキットのインストールには、modulefile インタープリターは含まれていないため、個別にインストールする必要があります。同様に、modulefile には w 権限は必要ありませんが、読み取り可能である必要があります (r 権限をすべてのユーザーに設定します)。

3.8.4 バージョン管理

インテル® oneAPI ツールキットのインストーラーは、バージョンフォルダーを使用して、インテル® oneAPI ツールとライブラリーの共存を可能にしています。modulefiles-setup.sh スクリプトはバージョン管理されたコンポーネント・フォルダーを使用して、バージョン管理された modulefile を作成します。modulefiles 出力フォルダーに [<modulefile 名>/バージョン] としてシンボリック・リンクが作成されるため、module コマンドの使用時にそれぞれの modulefile をバージョン別に参照できます。

```
$ module avail
---------------- modulefiles -----------------
ipp/1.1 ipp/1.2 compiler/1.0 compiler32/1.0
```

3.8.5 複数の modulefile

ツールやライブラリーは、modulefiles フォルダー内に複数の modulefile を持つ場合があります。それぞれがロード可能モジュールになります。展開されたコンポーネント・フォルダーごとにバージョンが割り当てられます。

3.8.6 oneAPI で使用する modulefile の記述法を理解する

modulefiles-setup.sh スクリプトはシンボリック・リンクを使用して、すべての modulefile を modulefiles フォルダーに集約します。実際の modulefile は移動または変更されません。そのため、${ModulesCurrentModulefile} 変数には、それぞれのインストール・フォルダーにある実際の modulefile ではなく、各 modulefile への symlink が格納されます。各 modulefile は、次のような形式を使用して、それぞれのインストール・ディレクトリーにある元の modulefile への参照を取得します。

```
[ file readlink ${ModulesCurrentModulefile} ]
```

これは、実際のインストール場所はカスタマイズできるため、実行時に不明であり推測する必要があるためです。実際の modulefile はインストールされた場所以外に移動することはできません。そうしないと、構成に必要なライブラリーやアプリケーションへの絶対パスを検出できなくなります。

さらに詳しく理解するには、インストールに含まれる modulefile を確認してください。ほとんどの場合、実際のファイルへの symlink を解決する方法のコメントと、バージョン番号 (およびバージョン・ディレクトリー) の解析が含まれます。また、インストールされた TCL が適切なバージョンであることを確認するチェックも含まれています。

3.8.7 `modulefiles` による `module load` コマンドの使用

> **注:** この節で説明する内容は、2023.2 ツールキット以前の modulefile に対してのみ適用されます。2024.0 以降のツールキットで提供される modulefile は、必要となる modulefile を自動でロードしません。前提条件のある modulefile は、適切な modulefile がロードされているかチェックし (modulefile prereq コマンドを使用)、ロードされていない場合はエラーメッセージを出力します。依存関係のある modulefile の事前ロードは開発者に任されています。oneapi コンビニエンス modulefile は、すべての前提条件 modulefile を自動的にロードします。

`modulefile` のいくつかは、依存関係のあるモジュールがロードされるように `module load` コマンドを使用します。依存関係のある `modulefile` のバージョンチェックは行われません。これは、モジュールをロードする前に、その依存関係モジュールを手動で事前ロードできることを意味します。依存関係モジュールを事前ロードしない場合、最新バージョンがロードされます。

これは、環境を柔軟にコントロールできるようにするための設計です。例えば、以前のバージョンのコンパイラーで更新されたバージョンのライブラリーをテストするためインストールすると仮定します。更新されたライブラリーにはバグ修正があり、更新されたライブラリー内の他のモジュールには興味がないという可能性もあります。依存する `modulefile` が、ライブラリーの特定のバージョンを使用するようにハードコーディングされている場合、このテストは機能しません。

依存関係の `module load` 条件を満たせない場合、モジュールのロードは終了し環境は変更されません。

3.8.8 関連情報

`modulefile` の詳細については、以下をご覧ください。

- http://www.admin-magazine.com/HPC/Articles/Environment-Modules (英語)
- https://www.chpc.utah.edu/documentation/software/modules-advanced.php (英語)
- https://modules.readthedocs.io/en/latest/ (英語)
- https://lmod.readthedocs.io/en/latest/ (英語)

3.9 oneAPI アプリケーションで CMake* を使用

インテル® oneAPI 製品で提供される CMake* パッケージを使用すると、CMake* プロジェクトで Windows* または Linux* 上の oneAPI ライブラリーを簡単に利用できます。このパッケージを使用することで、ほかのシステム・ライブラリーが CMake* プロジェクトと統合する場合と同様の体験ができます。CMake* プロジェクトのターゲットに応じて、依存関係が生じたり、ほかのビルド変数が必要になることがあります。

次のコンポーネントが CMake* をサポートします。

- インテル® oneAPI DPC++ コンパイラー - Linux*、Windows*
- インテル® インテグレーテッド・パフォーマンス・プリミティブ (インテル® IPP) およびインテル® インテグレーテッド・パフォーマンス・プリミティブ・クリプトグラフィー (インテル® IPP Cryptgraphy) - Linux*、Windows*
- インテル® MPI ライブラリー - Linux*、Windows*
- 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 (oneCCL) - Linux*、Windows*
- インテル® oneAPI データ・アナリティクス・ライブラリー (oneDAL) - Linux*、Windows*
- インテル® oneAPI ディープ・ニューラル・ネットワーク・ライブラリー (oneDNN) - Linux*、Windows*
- インテル® oneAPI DPC++ ライブラリー (oneDPL) - Linux*、Windows*
- インテル® oneAPI マス・カーネル・ライブラリー (oneMKL) - Linux*、Windows*
- インテル® oneAPI スレッディング・ビルディング・ブロック (oneTBB) - Linux*、Windows*
- インテル® ビデオ・プロセシング・ライブラリー (oneVPL) - Linux*、Windows*

> **注:** 2024 リリース以降では、macOS* はインテル® oneAPI ツールキットおよびコンポーネントでサポートされなくなりました。oneAPI スレッディング・ビルディング・ブロック (oneTBB) やインテル® Implicit SPMD Program Compiler など、いくつかのオープンソース・プロジェクトは、引き続き Apple シリコン上の macOS* をサポートします。これらのツール開発の貢献と協力は歓迎します。

CMake* 設定を提供するライブラリーは、以下で識別できます。

- Linux* または macOS*:

  システム: `/usr/local/lib/cmake`
  ユーザー: `~/lib/cmake`

- Windows*: `HKEY_LOCAL_MACHINE\Software\Kitware\CMake\Packages\`

CMake* パッケージを使用するには、ほかのシステム・ライブラリーと同様にインテル® oneAPI ライブラリーを使用します。例えば、`find_package(tbb)` を使用すると、アプリケーションの CMake* パッケージはインテル® oneAPI スレッディング・ビルディング・ブロック (oneTBB) パッケージを使用します。

4 oneAPI プログラムのコンパイルと実行

この章では、CPU、GPU、および FPGA を対象とするダイレクト・プログラミングと API ベースのプログラミングにおける oneAPI のコンパイル手順について詳しく説明します。また、コンパイルの各ステージで使用されるツールについても詳しく説明します。

4.1 単一ソースのコンパイル

oneAPI プログラミング・モデルは、単一ソースのコンパイルをサポートします。単一ソースのコンパイルには、ホストとデバイスコードを個別にコンパイルする場合と比較して多くの利点があります。oneAPI プログラミング・モデルは、一部のユーザーの要望に答え、ホストコードとデバイスコードの個別コンパイルもサポートすることを覚えておいてください。利用可能な単一ソースのコンパイルには次のモデルがあります:

- 利便性 - 開発者は作成するファイル数を最小限に抑え、ホストコードの呼び出し元の直後にデバイスコードを定義できます。
- 安全性 - 単一ソースでは、単独のコンパイラーがホストとデバイス間の境界にあるコードを判断することができ、ホスト・コンパイラーによって生成される仮引数がデバイス・コンパイラーによって生成されるカーネルの実引数と一致します。
- 最適化 - デバイス・コンパイラーは、カーネルが起動されるコンテキストを知ることができるため、さらなる最適化を行うことができます。例えば、コンパイラーはいくつかの定数を伝搬したり、関数呼び出し全体でポインターのエイリアシング情報を推測することができます。

4.2 コンパイラーの起動

インテル® oneAPI DPC++/C++ コンパイラーは、コマンドラインでコンパイラーを起動する複数のコンパイラー・ドライバーを提供します。次の例は、C++ および SYCL* のオプションを示します。ドライバーオプションの詳細については、「各種コンパイラーと ドライバーの一覧」(英語) を参照してください。

コンパイラーの起動に関する詳細は、『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』の「コンパイラーの起動」(英語) を参照してください。

C++ アプリケーションをコンパイルする際に OpenMP* を有効にするには、次のコマンドでコンパイラーを起動します。

```
$ icpx -fiopenmp -fopenmp-targets=<arch> (Linux*)
$ icx /Qiopenmp /Qopenmp-targets:<arch> (Windows*)
```

SYCL* アプリケーションをコンパイルする際に OpenMP* を有効にするには、次のコマンドでコンパイラーを起動します。

```
$ icpx -fsycl -fiopenmp -fopenmp-targets=<arch> (Linux*)
$ icx-cl -fsycl /Qiopenmp /Qopenmp-targets:<arch> (Windows*)
```

オプションの詳細については、『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』の「コンパイラー・オプション」(英語) を参照してください。

コンパイラー・ドライバーは、OS ホストごとに互換性が異なります。Linux* では、GCC スタイルの `icpx -fsycl` コマンドライン・オプションが提供され、Windows* では Microsoft* Visual Studio* の Microsoft Visual C++* 互換の `icx-cl` が提供されます。

- GCC 形式のコマンドライン・オプション ("-" で始まる) を認識し、複数のオペレーティング・システムでビルドシステムを共有するプロジェクトに役立ちます。
- Windows* コマンドライン・オプション ("/" で始まる) を認識し、Microsoft* Visual Studio* ベースのプロジェクトで使用できます。

4.3 インテル® oneAPI DPC++/C++ コンパイラーの標準オプション

インテル® oneAPI DPC++/C++ コンパイラーの完全なオプションリストは、『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』に記載されています。

- 「オフロード向けのコンパイルオプションと OpenMP* オプションおよび並列処理オプション」(英語) には、SYCL* と OpenMP* オフロードに固有のオプションが示されています。
- 利用可能なすべてのオプションと簡単な説明は、「コンパイラー・オプションのリスト (アルファベット順)」(英語) にあります。

4.4 コンパイル例

oneAPI アプリケーションは、ダイレクト・プログラミング、利用可能な oneAPI ライブラリーを使用する API ベース、およびそれらを組み合わせて記述することができます。API ベースのプログラミングでは、ライブラリーの機能を使用してデバイスへのオフロードを行います。これにより、開発者はアプリケーションを開発する時間を節約できます。一般に、API ベースのプログラミングから始め、ニーズに対応できない場合に SYCL* または OpenMP* オフロード機能を導入するのが最も容易であると考えられます。

次のセクションでは、API ベースのコードと SYCL* を使用したダイレクト・プログラミングの例を紹介します。

4.4.1 API ベースのコード

次のコードは、インテル® oneAPI マス・カーネル・ライブラリー (oneMKL) の `oneapi::mkl::blas::axpy` 関数を使用して、浮動小数点数のベクトル全体に対して `a` と `x` を乗算して `y` を加算する API 呼び出し `(a * x + y)` の使用法を示しています。このサンプルコードは、oneAPI プログラミング・モデルを利用して、アクセラレーターで加算を実行します。

```
1.  #include <vector> // std::vector()
2.  #include <cstdlib> // std::rand()
3.  #include <CL/sycl.hpp>
4.  #include "oneapi/mkl/blas.hpp"
5.
6.  int main(int argc, char* argv[]) {
7.
8.    double alpha = 2.0;
9.    int n_elements = 1024;
10.
11.
12.   int incx = 1;
13.   std::vector<double> x;
14.   x.resize(incx * n_elements);
15.   for (int i=0; i<n_elements; i++)
16.     x[i*incx] = 4.0 * double(std::rand()) / RAND_MAX - 2.0;
17.     // -2.0 から 2.0 の範囲の rand 値
18.
19.   int incy = 3;
20.   std::vector<double> y;
21.   y.resize(incy * n_elements);
22.   for (int i=0; i<n_elements; i++)
23.     y[i*incy] = 4.0 * double(std::rand()) / RAND_MAX - 2.0;
24.     // -2.0 から 2.0 の範囲の rand 値
25.
26.   cl::sycl::device my_dev;
27.   try {
28.     my_dev = cl::sycl::device(cl::sycl::gpu_selector());
29.   } catch (...){
30.     std::cout << "Warning, failed at selecting gpu device.Continuing on default(host)
    device.\n";
31.   }
32.
33.   // 非同期例外をキャッチ
34.   auto exception_handler = [] (cl::sycl::exception_list
35.     exceptions) {
36.     for (std::exception_ptr const& e : exceptions) {
37.       try {
38.         std::rethrow_exception(e);
39.       } catch(cl::sycl::exception const& e) {
40.         std::cout << "Caught asynchronous SYCL exception:\n";
41.         std::cout << e.what() << std::endl;
42.       }
43.     }
44.   };
45.
46.   cl::sycl::queue my_queue(my_dev, exception_handler);
47.
48.
49.   cl::sycl::buffer<double, 1> x_buffer(x.data(), x.size());
50.   cl::sycl::buffer<double, 1> y_buffer(y.data(), y.size());
```

```
51.
52.   // y = alpha*x + y を計算
53.   try {
54.     oneapi::mkl::blas::axpy(my_queue, n_elements, alpha, x_buffer,
55.     incx, y_buffer, incy);
56.   }
57.
58.   catch(cl::sycl::exception const& e) {
59.     std::cout << "\t\tCaught synchronous SYCL exception:\n"
60.               << e.what() << std::endl;
61.   }
62.
63.   std::cout << "The axpy (y = alpha * x + y) computation is complete!"<< std::endl;
64.
65.
66.   // y_buffer をプリント
67.   auto y_accessor = y_buffer.template
68.     get_access<cl::sycl::access::mode::read>();
69.   std::cout << std::endl;
70.   std::cout << "y" << " = [ " << y_accessor[0] << " ]\n";
71.   std::cout << "    [ " << y_accessor[1*incy] << " ]\n";
72.   std::cout << "    [ " << "... ]\n";
73.   std::cout << std::endl;
74.
75.   return 0;
76. }
```

アプリケーション (axpy.cpp として保存) をコンパイルするには、次の操作を行います。

1. MKLROOT 環境変数が適切に設定されていることを確認します。

   Linux*: `echo ${MKLROOT}`
   Windows*: `echo %MKLROOT%`

   正しく設定されていない場合、setvars.sh、oneapi-vars.sh スクリプトを source するか (Linux*)、setvars.bat、oneapi-vars.bat (Windows*) スクリプトを実行、または lib および include サブディレクトリーを含むディレクトリー・パスを変数に設定します。

   setvars および oneAPI-vars スクリプトの詳細については、「oneAPI 開発環境の設定」を参照してください。

2. 次のコマンドでアプリケーションをビルドします。

   Linux*:

```
$ icpx -fsycl -I${MKLROOT}/include -c axpy.cpp -o axpy.o
```

   Windows*:

```
$ icpx -fsycl -I${MKLROOT}/include /EHsc -c axpy.cpp /Foaxpy.obj
```

3. 次のコマンドでアプリケーションをリンクします。

Linux*:

```
$ icpx -fsycl axpy.o -fsycl-device-code-split=per_kernel \
"${MKLROOT}/lib/intel64"/libmkl_sycl.a -Wl,-export-dynamic -Wl,--start-group\
"${MKLROOT}/lib/intel64"/libmkl_intel_ilp64.a \
"${MKLROOT}/lib/intel64"/libmkl_sequential.a \
"${MKLROOT}/lib/intel64"/libmkl_core.a -Wl,--end-group -lsycl -lOpenCL \
-lpthread -lm -ldl -o axpy.out
```

Windows*:

```
$ icpx -fsycl axpy.obj -fsycl-device-code-split=per_kernel ^
"${MKLROOT}/lib/intel64"/mkl_sycl.lib ^
"${MKLROOT}/lib/intel64"/mkl_intel_ilp64.lib ^
"${MKLROOT}/lib/intel64"/mkl_sequential.lib ^
"${MKLROOT}/lib/intel64"/mkl_core.lib ^
sycl.lib OpenCL.lib -o axpy.exe
```

4. 次のコマンドでアプリケーションを実行します。

Linux*:

```
$ ./axpy.out
```

Windows*:

```
$ axpy.exe
```

4.4.2 ダイレクト・プログラミング

ここでは、「ベクトル加算のサンプルコード」(英語) を使用します。このサンプルコードは、oneAPI プログラミング・モデルを利用して、アクセラレーターで加算を実行します。

次のコマンドで、実行形式をコンパイルしてリンクします。

```
$ icpx -fsycl vector_add.cpp
```

コマンドとオプションのコンポーネントと関数は、「API ベースのコード」セクションで説明したものと類似しています。

このコマンドを実行すると、実行時にベクトルの加算を実行する実行ファイルが作成されます。

4.5 コンパイルの手順

オフロードを行うプログラムを作成する場合、コンパイラーはホストとデバイス向けの両方のコードを生成する必要があります。oneAPI は、この複雑な作業を開発者から見えないようにします。開発者は、DPC++ コンパイラー (`icpx -fsycl`) を使用して SYCL* アプリケーションをコンパイルするだけで (一度のコンパイルコマンドで)、ホストとデバイス向けのコードを生成できます。

注: システム要件に示されるインテル® プロセッサーに加え、AMD* および NVIDIA* GPU もターゲットとなる場合があります (Linux* のみ)。

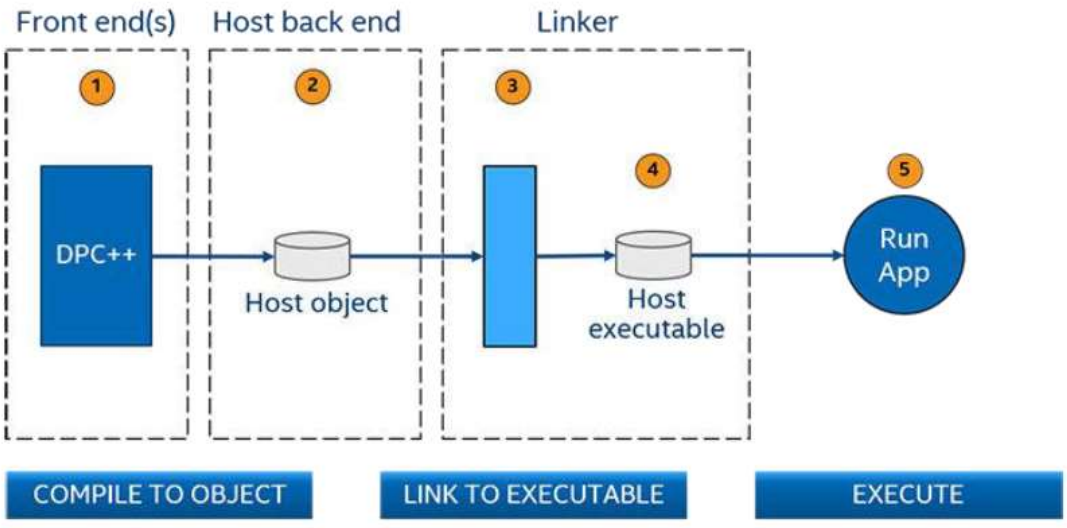
- インテル® oneAPI DPC++ コンパイラーで AMD* GPU を使用するには、Codeplay から oneAPI for AMD* GPU プラグインを入手してインストールします。
- インテル® oneAPI DPC++ コンパイラーで NVIDIA* GPU を使用するには、Codeplay から oneAPI for NVIDIA* GPU プラグインを入手してインストールします。

デバイスコードの実行には、Just-in-Time (JIT) コンパイルと Ahead-of-Time (AOT) コンパイルの 2 つのオプションがありますが、JIT がデフォルトです。このセクションでは、ホストコードのコンパイル方法と、デバイスコードを生成する 2 つの方法を説明します。詳しくは、『Data Parallel C++』(英語) 書籍の 13 章をご覧ください。

4.5.1 従来のコンパイル手順 (ホストのみのアプリケーション)

従来のコンパイル手順は、C、C++、またはそのほかの言語で利用される標準のコンパイル方法です。デバイスへのオフロードがない場合に使用されます。コンパイル手順を図に示します。

従来のコンパイル手順



1. フロントエンドは、ソースを中間表現に変換し、バックエンドに渡します。

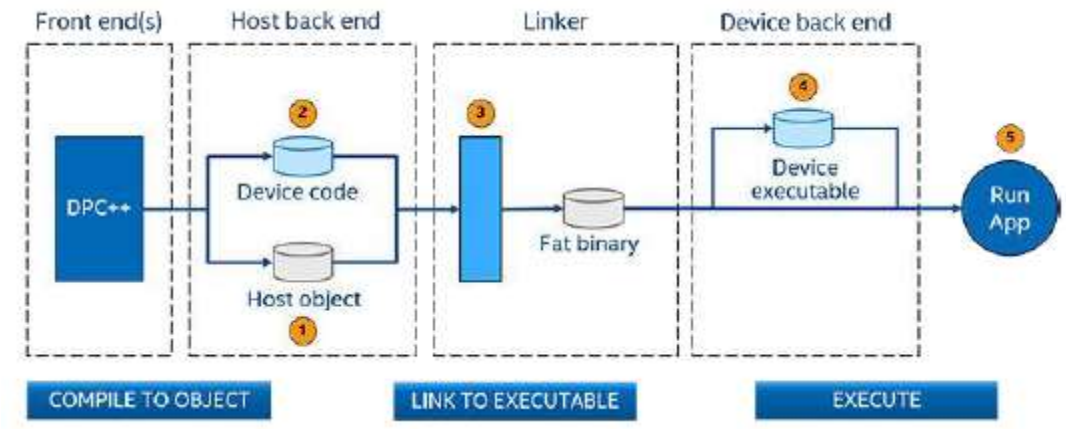
2. バックエンドは、中間表現をオブジェクト・コードに変換してオブジェクト・ファイル (Windows* では `.obj`、Linux* では `.o`) を出力します。
3. 1 つ以上のオブジェクト・ファイルがリンカーに渡されます。
4. リンカーは実行ファイルを生成します。
5. これで、アプリケーションを実行できます。

4.5.2 SYCL* オフロードコードのコンパイル手順

SYCL* オフロードコードのコンパイル手順には、従来のコンパイル手順にデバイスコードの JIT および AOT オプションを追加します。この手順では、開発者は `icpx -fsycl` を使用して、SYCL* アプリケーションをコンパイルし、出力としてホストコードとデバイスコードの両方を含む実行可能ファイルを作成します。

SYCL* オフロードコードの基本コンパイル手順を次に示します。

SYCL* オフロードコードの基本コンパイル手順



1. ホストコードは、バックエンドでオブジェクト・コードに変換されます。
2. デバイスコードは SPIR-V* 形式またはデバイスバイナリーに変換されます。
3. リンカーは、ホスト・オブジェクト・コードとデバイスコード (SPIR-V* またはデバイスバイナリー) を組み合わせた、(デバイスコードが埋め込まれた) ホスト実行コードを含むファットバイナリーを生成します。
4. バイナリーが起動されると、オペレーティング・システムはホスト・アプリケーションを実行します。オフロードが行われる場合、ランタイムはデバイスコードをロードします (必要に応じて SPIR-V* をデバイスバイナリーに変換します)。
5. アプリケーションは、ホストと利用可能なデバイスで実行されます。

4.5.3 JIT のコンパイル手順

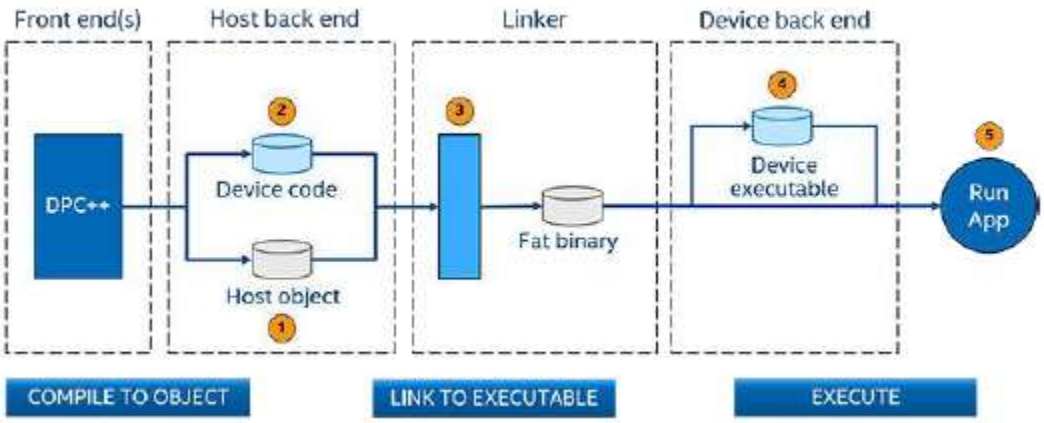
JIT コンパイルの手順では、デバイスコードはバックエンドで SPIR-V* 形式の中間コードに変換され、SPIR-V* としてファットバイナリーに組み込まれ、ランタイムによって SPIR-V* からデバイスバイナリーに変換されます。アプリケーションが起動されると、ランタイムは利用可能なデバイスを判別してそのデバイス固有のコードを生成します。これにより、AOT (事前) コンパイル手順よりも、アプリケーションの実行環境とパフォーマンスの柔軟性が高まります。しかし、アプリケーションの実行時にコンパイル (JIT) が行われるため、アプリケーションの実行時間が増加する可能性があります。大量のデバイスコードを持つ大規模なアプリケーションでは、パフォーマンスへの影響が顕著に表れることがあります。

ヒント: JIT コンパイル手順は、ターゲットデバイスが不明である場合に役立ちます。

注: JIT コンパイラーは、FPGA デバイスではサポートされません。

コンパイル手順を次の図に示します。

JIT コンパイル手順



1. ホストコードは、バックエンドでオブジェクト・コードに変換されます。
2. デバイスコードは SPIR-V* 形式に変換されます。
3. リンカーは、ホスト・オブジェクト・コードとデバイス SPIR-V* を組み合わせた (SPIR-V* が埋め込まれた)、ホスト実行コードを含むファットバイナリーを生成します。
4. 実行されると次のように処理されます。

   (a) ホスト上のデバイスランタイムは、デバイスの SPIR-V* をデバイスのバイナリーに変換します。

   (b) 変換されたデバイスバイナリーはデバイスへロードされます。

5. アプリケーションは、実行時に利用可能なホストとデバイスで実行されます。

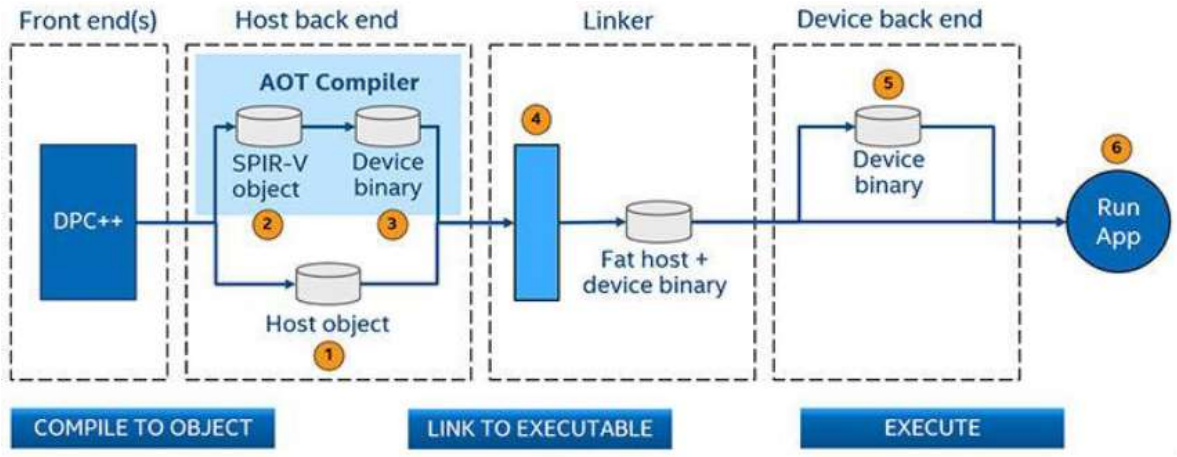
4.5.4 AOT のコンパイル手順

AOT (事前) コンパイルでは、デバイスコードが SPIR-V* に変換されてから、ホスト・バックエンドのデバイスコードに変換され、最終的に生成されたデバイスコードがファットバイナリーに組み込まれます。しかし、実行ファイルの起動時間は JIT 手順よりも短くなります。

> **ヒント:** AOT 手順は、ターゲットとするデバイスが明確に判明している場合に適しています。AOT 手順では、デバッグサイクルが高速化されるため、アプリケーションをデバッグする際の利用が推奨されます。

コンパイル手順を次の図に示します。

AOT コンパイル手順



1. ホストコードは、バックエンドでオブジェクト・コードに変換されます。
2. デバイスコードは SPIR-V* 形式に変換されます。
3. デバイスの SPIR-V* は、コマンドラインでユーザーが指定したデバイス向けのデバイス・コード・オブジェクトに変換されます。
4. リンカーは、ホスト・オブジェクト・コードとデバイス・オブジェクト・コードを組み合わせた、デバイスバイナリーが埋め込まれたホスト実行コードを含むファットバイナリーを生成します。
5. 実行時に、デバイスバイナリーはデバイスへロードされます。
6. アプリケーションは、ホストと指定されたデバイスで実行されます。

4.5.5 ファットバイナリー

ファットバイナリーは、JIT と AOT コンパイル手順から生成されます。デバイスコードが埋め込まれたホストバイナリーです。デバイスコード自体は、コンパイル手順によって異なります。

ファットバイナリー

- ホストコードは、ELF (Linux*) または PE (Windows*) 形式の実行ファイルです。
- デバイスコードは、JIT 手順では SPIR-V*、AOT 手順ではデバイスバイナリー (実行可能) です。実行ファイルは次のいずれかの形式です。
  - CPU: ELF (Linux*)、PE (Windows*)
  - GPU: ELF (Windows*、Linux*)
  - FPGA: ELF (Linux*)、PE (Windows*)

4.6 CPU 手順

CPU はコンピューターの頭脳とも呼ばれ、分岐予測、メモリーの仮想化、命令のスケジュールなどを含む複雑な回路/アルゴリズムで構成されています。このような複雑性を考慮し、CPU は幅広いタスクを処理するように設計されています。

SYCL* および OpenMP* オフロードを利用するプログラミング・モデルにより、異種 CPU および GPU システムでのアプリケーションの実装が可能になります。SYCL* および OpenMP* オフロードにおける「デバイス」は、CPU と GPU の両方を指す場合があります。

最新の CPU は、並列計算に利用可能なハイパースレッディングと広い SIMD 幅を備えた複数のコアを搭載しています。ワークロードに計算集約型で並列実行可能な領域が存在する場合、そのような領域は GPU や FPGA などのコプロセッサーではなく CPU へオフロードすることを推奨します。これはまた、データを PCIe* を介してオフロードする必要がないため (コプロセッサーや GPU とは異なり)、データ転送のオーバーヘッドを最小限に抑えてレイテンシーを軽減します。

CPU でアプリケーションを実行するには、CPU で直接実行される従来型の CPU フローと、CPU デバイスで実行される CPU オフロードの 2 つのオプションがあります。CPU オフロードは、SYCL* または OpenMP* オフロード・アプリケーションで使用できます。OpenMP* オフロード・アプリケーションと SYCL* オフロード・アプリケーションは、どちらも OpenCL* ランタイムとインテル® oneAPI スレッディング・ビルディング・ブロック (oneTBB) を使用して、CPU デバイスで実行できます。

ヒント: ワークロードが CPU、GPU、または FPGA に最適であるか不明な場合は、「各種 oneAPI ワークロードに対する CPU、GPU、および FPGA の利点の比較」(英語) を参照してください。

4.6.1 従来の CPU 向け手順

従来の CPU ワークフローは、ランタイムなしで CPU 上で動作します。コンパイルフローは、C、C++ またはほかの言語で使用されるようなデバイスへのオフロードを行わない標準的なコンパイルです。

従来のワークロードでは、コンパイルフローの概要で説明されているように、従来のコンパイルフロー (ホスト専用アプリケーション) 手順を使用して、ホスト上でコンパイルおよび実行されます。

コンパイルコマンドの例:

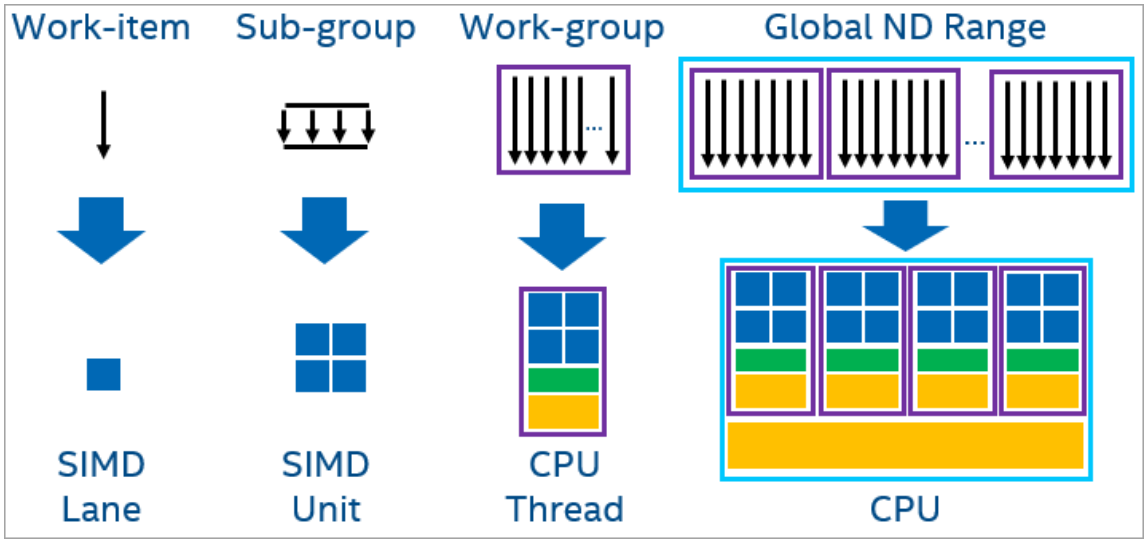
```
$ icpx -g -o matrix_mul_omp src/matrix_mul_omp.cpp
```

4.6.2 CPU オフロードの手順

デフォルトでは、CPU デバイスへオフロードする場合、OpenCL* ランタイムを使用します。OpenCL* ランタイムは、並列処理にインテル® oneAPI スレッディング・ビルディング・ブロック (oneTBB) も活用します。

CPU にオフロードする場合、ワークグループは異なる論理コアに割り当てられ、これらのワークグループは並列に実行できます。ワークグループ内のワーク項目は、CPU の SIMD レーンにマップできます。ワーク項目 (サブグループ) は、SIMD 方式で同時に実行されます。

CPU ワークグループ



CPU 実行の詳細については、「各種 oneAPI コンピューティング・ワークロードに対する CPU、GPU、および FPGA の利点の比較」(英語) を参照してください。

4.6.2.1. CPU オフロード向けの設定

1. `setvars` または `oneapi-vars` スクリプトの実行を含む、「oneAPI 開発環境の設定」セクションのすべての手順を実行したことを確認します。
2. `sycl-ls` コマンドを実行して、必要な OpenCL* ランタイムが CPU に関連付けられていることを確認します。

   例:

```
$ sycl-ls
CPU : OpenCL 2.1 (Build 0)[ 2020.11.12.0.14_160000 ]
GPU : OpenCL 3.0 NEO [ 21.33.20678 ]
GPU : 1.1[ 1.2.20939 ]
```

3. 次のサンプルコードを使用して、コードが CPU で実行されていることを確認します。サンプルコードは、整数の大きなベクトルにスカラーを加算し、結果を検証します。

SYCL*

SYCL* では CPU で実行するための組込みデバイスセレクターが用意されています。これは、`device_selector` 基本クラスを使用し `cpu_selector` で CPU デバイスを選択できます。

または、`default_selector` を使用して、実装で定義されたヒューリスティックに従って次の環境変数によって実行時にデバイスを選択することもできます。

```
$ export ONEAPI_DEVICE_SELECTOR=cpu
```

SYCL* サンプルコード:

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

using namespace sycl;
using namespace std;
constexpr size_t array_size = 10000;
int main(){
  constexpr int value = 100000;
  try{
    cpu_selector d_selector;
    queue q(d_selector);
    int *sequential = malloc_shared<int>(array_size, q);
    int *parallel = malloc_shared<int>(array_size, q);
    // シーケンシャル iota
    for (size_t i = 0; i < array_size; i++) sequential[i] = value + i;

    // SYCL* の並列 iota
    auto e = q.parallel_for(range{array_size}, [=](auto i) {
      parallel[i] = value + i;
```

```
21.     });
22.     e.wait();
23.     // 2 つの結果が等しいか検証
24.     for (size_t i = 0; i < array_size; i++) {
25.       if (parallel[i] != sequential[i]) {
26.         cout << "Failed on device.\n";
27.         return -1;
28.       }
29.     }
30.     free(sequential, q);
31.     free(parallel, q);
32.   }catch (std::exception const &e) {
33.     cout << "An exception is caught while computing on device.\n";
34.     terminate();
35.   }
36.   cout << "Successfully completed on device.\n";
37.   return 0;
38. }
```

次のコマンドでサンプルコードをコンパイルします。

```
$ dpcpp simple-iota-dp.cpp -o simple-iota.
```

追加のコマンドはサンプルの CPU コマンドから取得できます。

生成したバイナリーを実行します。

```
$ ./simple-iota
Running on device: Intel® Core™ i7-8700 CPU @ 3.20GHz
Successfully completed on device.
```

OpenMP*

OpenMP* のサンプルコード:

```
1.  #include<iostream>
2.  #include<omp.h>
3.  #define N 1024
4.  int main(){
5.    float *a = (float *)malloc(sizeof(float)*N);
6.
7.    for(int i = 0; i < N; i++)
8.      a[i] = i;
9.    #pragma omp target teams distribute parallel for simd map(tofrom: a[:N])
10.   for(int i = 0; i < 1024; i++)
11.     a[i]++;
12.
13.   std::cout<<”successfully completed on device.\n";
14.   return 0;
15. }
```

次のコマンドでサンプルコードをコンパイルします。

```
$ icpx simple-ompoffload.cpp -fiopenmp -fopenmp-targets=spir64 -o simple-ompoffload
```

CPU 上でオフロード領域を実行するには、バイナリーを実行する前に次の環境変数を設定します。

```
$ export LIBOMPTARGET_DEVICETYPE=cpu
$ export LIBOMPTARGET_PLUGIN=opencl
```

生成したバイナリーを実行します。

```
$ ./simple-ompoffload
Successfully completed on device
```

4.6.3 CPU へコードをオフロード

アプリケーションをオフロードする場合、ボトルネックとオフロードの利点が得られるコード領域を特定することが重要です。計算集約型、または高度なデータ並列カーネルコードがある場合、そのコード領域をオフロードすることを検討してください。

オフロードするコード領域を特定するには、· (英語) が役立ちます。

4.6.3.1. オフロードされたコードのデバッグ

以下のリストには、オフロードされるコードの基本的なデバッグのヒントが示されています。

- ホストターゲットをチェックしてコードが正しいことを確認します。
- `printf` を使用して、アプリケーションをデバッグします。SYCL* と OpenMP* オフロードでは、どちらもカーネルコードで `printf` がサポートされます。環境変数を設定して詳細なログ情報を取得します。
  - SYCL* では、次のデバッグ環境変数を利用できます。すべての環境変数については GitHub* (英語) から入手できます。

SYCL* で推奨されるデバッグ環境変数

環境変数	値	説明
`ONEAPI_DEVICE_SELECTOR`	`backend:device_type:device_num`	GitHub* (英語) の説明を参照してください。
`SYCL_PI_TRACE`	`1|2|-1`	1: SYCL*/DPC++ ランタイムプラグインの基本トレースログを出力します 2: SYCL*/DPC++ ランタイムプラグインのすべての API トレースを出力します -1: 2 のすべてと追加のデバッグ

		情報を出力します。

OpenMP* では、次のデバッグ環境変数が推奨されます。利用可能なすべての環境変数については、「LLVM/OpenMP* ドキュメント」(英語) を参照してください。

OpenMP* で推奨されるデバッグ環境変数

環境変数	値	説明
`LIBOMPTARGET_DEVICETYPE`	`cpu|gpu|host`	デバイスを選択します。
`LIBOMPTARGET_DEBUG`	`1`	詳細なデバッグ情報を出力します。
`LIBOMPTARGET_INFO`	LLVM/OpenMP* のドキュメントで利用可能な値 (英語)	ユーザーがさまざまなタイプのランタイム情報を `libomptarget` から取得できるようにします。

- 事前 (AOT) コンパイルを使用して、ジャストインタイム (JIT) コンパイルを AOT コンパイルに移行します。詳細については、「CPU アーキテクチャー向けの事前コンパイル」を参照してください。

oneAPI で利用可能なデバッグ手法とデバッグツールの詳細については、「SYCL* および OpenMP* オフロード処理のデバッグ」を参照してください。

4.6.4 CPU コードの最適化

CPU オフロードコードのパフォーマンスに影響する可能性がある多くの要因があります。ワーク項目、ワークグループ、および実行されるワーク量は、CPU コア数によって異なります。

- コアで実行されるワーク量が計算集約型でない場合、パフォーマンスが低下する可能性があります。これは、スケジュールのオーバーヘッドとスレッドのコンテキスト切り替えが原因です。
- CPU では、PCIe* を介したデータ転送が不要であり、オフロード領域がデータを長時間待機する必要がないため、レイテンシーが低くなります。
- アプリケーションの性質に基づいて、スレッド・アフィニティーは CPU のパフォーマンスに影響を与える可能性があります。詳細については、「マルチコアにおけるバイナリーの実行制御」を参照してください。
- デフォルトでは JIT コンパイルが使用されますが、代わりに AOT コンパイル (オフラインコンパイル) を使用して、特定の CPU アーキテクチャーをターゲットにしてコードをコンパイルします。詳細については、「CPU アーキテクチャー向けの最適化オプション」を参照してください。

「オフロード・パフォーマンスの最適化」で追加の推奨事項が提供されています。

4.6.5 CPU コマンドの例

次のコマンドは、デバイスコードの一部が静的ライブラリーにある実装を想定しています。

注: 動的ライブラリーとのリンクはサポートされていません。

デバイスコードを使用してファット・オブジェクトを生成します。

```
$ icpx -fsycl -c static_lib.cpp
```

ar ツールを使用して、静的ファット・ライブラリーを作成します。

```
$ ar rc r libstlib.a static_lib.o
```

アプリケーションのソースをコンパイルします。

```
$ icpx -fsycl -c a.cpp
```

静的ライブラリーとアプリケーションをリンクします。

```
$ icpx -fsycl -foffload-static-lib=libstlib.a a.o -o a.exe
```

4.6.6 CPU アーキテクチャー向けの事前 (AOT) コンパイル

事前 (AOT) コンパイルモードでは、最適化オプションを使用して、特定の CPU アーキテクチャーでの実行を改善するコードを生成できます。

```
$ icpx -fsycl -fsycl-targets=spir64_x86_64 -Xs "-device <CPU 最適化フラグ>" a.cpp b.cpp -o
app.out
```

次の CPU 最適化オプションがサポートされます。

```
-march=<instruction_set_arch> ターゲットの命令セット・アーキテクチャーを設定:
'sse42' インテル® ストリーミング SIMD 拡張命令 4.2
'avx2' インテル® アドバンスト・ベクトル・エクステンション 2
'avx512' インテル® アドバンスト・ベクトル・エクステンション 512
```

注: サポートされる最適化オプションは、将来のリリースで変更される可能性があります。

4.6.7 複数の CPU コア上でバイナリーの実行をコントロール

4.6.7.1. 環境変数

次の環境変数は、プログラムの実行中に複数の CPU コアへ SYCL* または OpenMP* スレッドの配置をコントロールします。OpenCL* ランタイム CPU デバイスを使用して CPU にオフロードする場合、これらの変数を使用します。

SYCL* または OpenMP* 環境変数

環境変数	説明
DPCPP_CPU_CU_AFFINITY	CPU へスレッド・アフィニティーを設定します。以下を指定できます。 • close - スレッドは利用可能な CPU コアに連続して配置されます。 • spread - スレッドは利用可能なコアに分散されます。 • master - スレッドはプライマリー・スレッドと同じコアに配置されます。DPCPP_CPU_CU_AFFINITY が設定されているとプライマリー・スレッドも固定されます (未設定では固定されません)。 この環境変数は、OpenMP* の OMP_PROC_BIND 環境変数に似ています。 **デフォルト**: 未設定
DPCPP_CPU_SCHEDULE	スケジューラーが work-group をスケジュールするアルゴリズムを指定します。現在、SYCL* ランタイムはインテル® oneAPI スレッディング・ビルディング・ブロック (oneTBB) のスケジューラーを使用しています。この値は、oneTBB スケジューラーが使用するパーティショナーを選択します。以下を指定できます。 • dynamic - oneTBB の auto_partitioner。負荷を分散するため適切な分割を行います。 • affinity - oneTBB の affinity_partitioner。subrange をワーカースレッドにマッピングすることで、auto_partitioner のキャッシュ・アフィニティーを向上させます。 • static - oneTBB の static_partitioner。range の反復をワーカースレッド間でできるだけ均等に分散します。oneTBB のパーティショナーは、grainsize を使用してチャンクの大きさを制御します。デフォルトの grainsize は 1 であり、すべての work-group を独立して実行できることを示します。 **デフォルト**: dynamic
DPCPP_CPU_NUM_CUS	カーネルの実行に使用するスレッド数を設定します。 オーバーサブスクリプション状態を回避するには、DPCPP_CPU_NUM_CUS の最大値をハードウェア・スレッド数にする必要があります。DPCPP_CPU_NUM_CUS が 1 である場合、すべての work-group は単一のスレッドで順番に実行されます。これはデバッグ時に役立ちます。 この環境変数は、OpenMP* の OMP_NUM_THREADS に似ています。 **デフォルト**: 未設定。oneTBB によって決定されます。

環境変数	説明
DPCPP_CPU_PLACES	アフィニティーを設定する場所を指定します。 値は、{ sockets | numa_domains | cores | threads } のいずれかです。 この環境変数は、OpenMP* の OMP_PLACES 環境変数に似ています。 numa_domains が選択されると、oneTBB の NUMA API が使用されます。これは、OpenMP* 5.1 の OMP_PLACES=numa_domains に似ています。oneTBB の task arena は numa ノードにバインドされ、SYCL* nd-range は task arena に均一に分散されます。 DPCPP_CPU_PLACES は、DPCPP_CPU_CU_AFFINITY とともに使用することを推奨します。 **デフォルト**: cores

サポートされる環境変数の詳細については、『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) をご覧ください。

4.6.7.2. 例 1: インテル® ハイパースレッディング・テクノロジー有効

2 ソケットで、ソケットごとに 4 つの物理コアがあり、それぞれの物理コアには 2 つのハイパースレッドがあるマシンを想定します。

- S<num> は、リストで指定される 8 つのコアを持つソケット番号を示します
- T<num> は、oneTBB のスレッド番号を示します。
- "-" は未使用のコアを意味します。

```
DPCPP_CPU_NUM_CUS=16
   export DPCPP_CPU_PLACES=sockets
   DPCPP_CPU_CU_AFFINITY=close:S 0:[T0 T1 T2 T3 T4 T5 T6 T7] S1:[T8 T9 T10 T11 T12 T13 T14
T15]
   DPCPP_CPU_CU_AFFINITY=spread: S0:[T0 T2 T4 T6 T8 T10 T12 T14] S1:[T1 T3 T5 T7 T9 T11 T13
T15]
   DPCPP_CPU_CU_AFFINITY=master: S0:[T0 T1 T2 T3 T4 T5 T6 T7] S1:[T8 T9 T10 T11 T12 T13 T14
T15]


   export DPCPP_CPU_PLACES=cores
   DPCPP_CPU_CU_AFFINITY=close: S0:[T0 T8 T1 T9 T2 T10 T3 T11] S1:[T4 T12 T5 T13 T6 T14 T7
T15]
   DPCPP_CPU_CU_AFFINITY=spread: S0:[T0 T8 T2 T10 T4 T12 T6 T14] S1:[T1 T9 T3 T11 T5 T13 T7
T15]
   DPCPP_CPU_CU_AFFINITY=master: S0:[T0 T1 T2 T3 T4 T5 T6 T7]  S1:[T8 T9 T10 T11 T12 T13 T14
T15]


   export DPCPP_CPU_PLACES=threads
```

```
   DPCPP_CPU_CU_AFFINITY=close: S0:[T0 T1 T2 T3 T4 T5 T6 T7] S1:[T8 T9 T10 T11 T12 T13 T14
T15]
   DPCPP_CPU_CU_AFFINITY=spread: S0:[T0 T2 T4 T6 T8 T10 T12 T14] S1:[T1 T3 T5 T7 T9 T11 T13
T15]
   DPCPP_CPU_CU_AFFINITY=master: S0:[T0 T1 T2 T3 T4 T5 T6 T7] S1:[T8 T9 T10 T11 T12 T13 T14
T15]


export DPCPP_CPU_NUM_CUS=8
   DPCPP_CPU_PLACES=sockets, cores and threads have the same bindings:
   DPCPP_CPU_CU_AFFINITY=close close:  S0:[T0 - T1 - T2 - T3 -] S1:[T4 - T5 - T6 - T7 -]
   DPCPP_CPU_CU_AFFINITY=close spread: S0:[T0 - T2 - T4 - T6 -] S1:[T1 - T3 - T5 - T7 -]
   DPCPP_CPU_CU_AFFINITY=close master: S0:[T0 T1 T2 T3 T4 T5 T6 T7] S1:[]
```

4.6.7.3. 例 2: インテル® ハイパースレッディング・テクノロジー無効

2 ソケットで、ソケットごとに 4 つの物理コアがあり、それぞれの物理コアには 2 つのハイパースレッドがあるマシンを想定します。

- `S<num>` は、リストで指定される 8 つのコアを持つソケット番号を示します。
- `T<num>` は、oneTBB のスレッド番号を示します。
- `"-"` は未使用のコアを意味します。

```
export DPCPP_CPU_NUM_CUS=8
DPCPP_CPU_PLACES=sockets, cores and threads have the same bindings:
DPCPP_CPU_CU_AFFINITY=close:   S0:[T0 T1 T2 T3]       S1:[T4 T5 T6 T7]
DPCPP_CPU_CU_AFFINITY=spread:  S0:[T0 T2 T4 T6]       S1:[T1 T3 T5 T7]
DPCPP_CPU_CU_AFFINITY=master:  S0:[T0 T1 T2 T3]       S1:[T4 T5 T6 T7]

export DPCPP_CPU_NUM_CUS=4
DPCPP_CPU_PLACES=sockets, cores and threads have the same bindings:
DPCPP_CPU_CU_AFFINITY=close:   S0:[T0 - T1 - ]        S1:[T2 - T3 - ]
DPCPP_CPU_CU_AFFINITY=spread:  S0:[T0 - T2 - ]        S1:[T1 - T3 - ]
DPCPP_CPU_CU_AFFINITY=master:  S0:[T0 T1 T2 T3]       S1:[ -  -  -  - ]
```

4.7 GPU 手順

GPU は、アプリケーションの計算集約型領域の負荷を軽減する用途で使用できる特殊な計算デバイスです。通常、GPU は多数の小規模なコアで構成され、大量のスループットをもたらします。タスクには、CPU に適したものと GPU に適したものがあります。

注: システム要件に示されるインテル® プロセッサーに加え、AMD* および NVIDIA* GPU もターゲットとなる場合があります (Linux* のみ)。

- インテル® oneAPI DPC++ コンパイラーで AMD* GPU を使用するには、Codeplay から oneAPI for AMD* GPU プラグインを入手してインストールします。

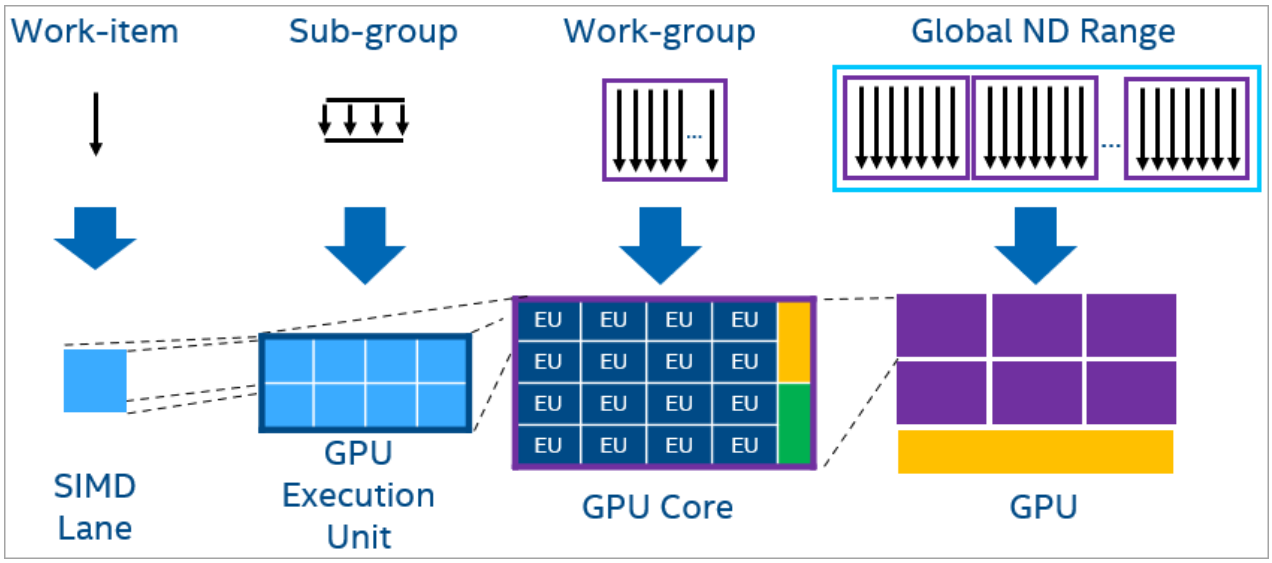
- インテル® oneAPI DPC++ コンパイラーで NVIDIA* GPU を使用するには、Codeplay から oneAPI for NVIDIA* GPU プラグインを入手してインストールします。

ヒント: ワークロードが CPU、GPU、または FPGA に最適であるか不明な場合は、「各種 oneAPI ワークロードに対する CPU、GPU、および FPGA の利点の比較」(英語) を参照してください。

4.7.1 GPU オフロードの手順

プログラムを GPU にオフロードすると、デフォルトでレベルゼロのランタイムが利用されます。OpenCL* ランタイムに切り替えるオプションも用意されています。SYCL* および OpenMP* オフロードでは、各ワーク項目は SIMD レーンにマップされます。サブグループは並列に実行されるワーク項目で形成される SIMD 幅に分割され、サブグループは GPU の EU スレッドにマップされます。ローカルデータを同期または共有するワーク項目を含むワークグループは、計算ユニット (ストリーミング·マルチプロセッサーまたは X^e コア – サブスライスとも呼ばれます) での実行に割り当てられます。最後に、ワーク項目のグローバル ND-Range 全体が GPU 全体にマップされます。

RPG インターフェイスの GPU ワークグループ



GPU 実行の詳細については、「各種 oneAPI コンピューティング·ワークロードに対する CPU、GPU、および FPGA の利点の比較」(英語) を参照してください。

4.7.1.1. GPU オフロード向けの設定

1. `setvars` または `oneapi-vars` スクリプトの実行を含む、「oneAPI 開発環境の設定」セクションのすべての手順を実行したことを確認します。

2. ドライバーをインストールして GPU システムを構成し、ユーザーを video グループに追加します。詳細については、導入ガイドを参照してください。

   - インテル® oneAPI ベース・ツールキット導入ガイド (Linux* | Windows*) (英語)
   - インテル® oneAPI HPC ツールキット導入ガイド (Linux* | Windows*) (英語)

3. `sycl-ls` コマンドを使用して、サポートされている GPU と必要なドライバーがインストールされていることを確認します。次の例では、OpenCL* およびレベルゼロドライバーがインストールされている場合、GPU に関連付けられたランタイムごとに 2 つのエントリーが表示されています。

```
CPU : OpenCL 2.1 (Build 0)[ 2020.11.12.0.14_160000 ]
GPU : OpenCL 3.0 NEO [ 21.33.20678 ]
GPU : 1.1[ 1.2.20939 ]
```

4. 次のサンプルコードを使用して、コードが GPU で実行されていることを確認します。サンプルコードは、整数の大きなベクトルにスカラーを加算し、結果を検証します。

SYCL*

SYCL* では GPU で実行するための組込みデ スセレクターが用意されています。これは、`device_selector` 基本クラスを使用し `gpu_selector` で GPU デバイスを選択できます。独自のカスタムセレクターを作成することもできます。詳細については、『Data Parallel C++』書籍の「Choosing Devices (デバイスの選択)」(英語) を参照してください。

SYCL* サンプルコード:

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

using namespace sycl;
using namespace std;
constexpr size_t array_size = 10000;
int main(){
  constexpr int value = 100000;
  try{
  //
  // デフォルトのデバイスセレクターは、最もパフォーマンスが高いデバイスを選択します
    default_selector d_selector;
    queue q(d_selector);

    // USM を使用した共有メモリー割り当て
    int *sequential = malloc_shared<int>(array_size, q);
    int *parallel = malloc_shared<int>(array_size, q);
    // シーケンシャル iota
    for (size_t i = 0; i < array_size; i++) sequential[i] = value + i;

    // SYCL* の並列 iota
    auto e = q.parallel_for(range{array_size}, [=](auto i) { parallel[i] = value + i; });
    e.wait();
    // 2 つの結果が等しいか検証
    for (size_t i = 0; i < array_size; i++) {
      if (parallel[i] != sequential[i]) {
```

```
28.          cout << "Failed on device.\n";
29.          return -1;
30.        }
31.      }
32.      free(sequential, q);
33.      free(parallel, q);
34.    }catch (std::exception const &e) {
35.      cout << "An exception is caught while computing on device.\n";
36.      terminate();
37.    }
38.    cout << "Successfully completed on device.\n";
39.    return 0;
40. }
```

次のコマンドでサンプルコードをコンパイルします。

```
$ icpx -fsycl simple-iota-dp.cpp -o simple-iota
```

生成したバイナリーを実行します。

```
$ ./simple-iota
Running on device: Intel® UHD Graphics 630 [0x3e92]
Successfully completed on device.
```

OpenMP*

OpenMP* のサンプルコード:

```
1.  #include <stdlib.h>
2.  #include <omp.h>
3.  #include <iostream>
4.  constexpr size_t array_size = 10000;
5.
6.  #pragma omp requires unified_shared_memory
7.  int main(){
8.    constexpr int value = 100000;
9.    // デフォルトのターゲットデバイスを返します
10.   int deviceId = (omp_get_num_devices() > 0) ? omp_get_default_device() :
    omp_get_initial_device();
11.   int *sequential = (int *)omp_target_alloc_host(array_size, deviceId);
12.   int *parallel = (int *)omp_target_alloc(array_size, deviceId);
13.
14.   for (size_t i = 0; i < array_size; i++)
15.     sequential[i] = value + i;
16.
17.   #pragma omp target parallel for
18.   for (size_t i = 0; i < array_size; i++)
19.     parallel[i] = value + i;
20.
21.   for (size_t i = 0; i < array_size; i++) {
22.     if (parallel[i] != sequential[i]) {
23.       std::cout << "Failed on device.\n";
24.       return -1;
25.     }
26.   }
27.
28.   omp_target_free(sequential, deviceId);
```

```
29.   omp_target_free(parallel, deviceId);
30.
31.   std::cout << "Successfully completed on device.\n";
32.   return 0;
33. }
```

次のコマンドでサンプルコードをコンパイルします。

```
$ icpx -fsyclsimple-iota-omp.cpp -fiopenmp -fopenmp-targets=spir64 -o simple-iota
```

生成したバイナリーを実行します。

```
$ ./simple-iota
Successfully completed on device.
```

注: オフロード領域が存在し、アクセラレーターがない場合、`OMP_TARGET_OFFLOAD=mandatory` 環境変数が指定されない限り、カーネルは従来のホストコンパイル (OpenCL* ランタイムなし) にフォールバックします。

4.7.1.2. GPU へコードをオフロード

どの GPU ハードウェアで、どのコード領域をオフロードするか決定するには、「GPU 最適化ワークフロー・ガイド」(英語) を参照してください。

オフロードするコード領域を特定するには、インテル® Advisor のオフロードのモデル化 (英語) が役立ちます。

4.7.1.2.1. GPU コードのデバッグ

以下のリストには、オフロードされるコードの基本的なデバッグのヒントが示されています。

- CPU またはホスト/ターゲットをチェック、またはランタイムを OpenCL* に切り替えてコードが正しいことを確認します。
- `printf` を使用して、アプリケーションをデバッグします。SYCL* と OpenMP* オフロードでは、どちらもカーネルコードで `printf` がサポートされます。
- 環境変数を設定して詳細なログ情報を取得します。

SYCL では、次のデバッグ環境変数を利用できます。すべての環境変数については GitHub* (英語) をご覧ください。

オフロードコードのデバッグのヒント

環境変数	値	説明
ONEAPI_DEVICE_SELECTOR	backend:device_type:device_num	GitHub* の説明を参照してください。
SYCL_PI_TRACE	1|2|-1	1: DPC++ ランタイムプラグインの基本トレースログを出力します。 2: DPC++ ランタイムプラグインのすべての API トレースを出力します。 -1: 2 のすべてと追加のデバッグ情報を出力します。
ZE_DEBUG	任意の値で定義された変数 (有効)	この環境変数は、DPC++ ランタイムが使用された際にレベルゼロ・バックエンドからのデバッグ出力を有効にします。以下が報告されます。 • レベルゼロ API の呼び出し • レベルゼロイベント情報

OpenMP* では、次のデバッグ環境変数が推奨されます。利用可能なすべての環境変数については、「LLVM/OpenMP* ドキュメント」(英語) を参照してください。

OpenMP* デバッグで推奨される環境変数

環境変数	値	説明
LIBOMPTARGET_DEVICETYPE	cpu | gpu	デバイスを選択します。
LIBOMPTARGET_DEBUG	1	詳細なデバッグ情報を出力します。
LIBOMPTARGET_INFO	LLVM/OpenMP* のドキュメントで利用可能な値 (英語)	ユーザーがさまざまなタイプのランタイム情報を libomptarget から取得できるようにします。

事前 (AOT) コンパイルを使用して、ジャストインタイム (JIT) コンパイルを AOT コンパイルに移行します。

4.7.1.2.2. CL_OUT_OF_RESOURCES エラー

CL_OUT_OF_RESOURCES エラーは、エミュレーターがデフォルトでサポートする __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カーネルが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)
$
```

または、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量とハードウェアの制限を確認するには、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)
$
```

または、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)
$
```

oneAPI で利用可能なデバッグ手法とデバッグツールの詳細については、「DPC++ および OpenMP* オフロード処理のデバッグ」を参照してください。

4.7.1.3. GPU コードの最適化

オフロードコードを最適化するにはいくつかの方法があります。次の表には、最適化のヒントが示されています。詳細については、「oneAPI GPU 最適化ガイド」を参照してください。

- ホストとデバイス間のメモリー転送のオーバーヘッドを削減します。
- コアをビジー状態に維持し、データ転送のオーバーヘッドのコストを軽減するため十分な量のワークを実行します。
- GPU キャッシュ、共有ローカルメモリーなど GPU メモリー階層を活用して、メモリーアクセスを高速化します。
- JIT コンパイルの代わりに AOT コンパイル (オフラインコンパイル) を使用します。事前コンパイルでは、コードを特定の GPU アーキテクチャーをターゲットにできます。詳細については、「GPU 向けの事前 (AOT) コンパイル」を参照してください。
- インテル® GPU Occupancy Calculator (英語) を使用すると、特定のカーネルおよびワークグループの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。

「オフロード・パフォーマンスの最適化」で追加の推奨事項が提供されています。

4.7.2 GPU コマンドの例

次の例は、Linux* でデバイスコードを使用して静的ライブラリーを作成し、使用する方法を示します。

> **注:** 動的ライブラリーとのリンクはサポートされていません。

デバイスコードを使用してファット・オブジェクトを生成します。

```
$ icpx -fsycl -c static_lib.cpp
```

ar ツールを使用して、静的ファット・ライブラリーを作成します。

```
$ ar cr libstlib.a static_lib.o
```

アプリケーションのソースをコンパイルします。

```
$ icpx -fsycl -c a.cpp
```

静的ライブラリーとアプリケーションをリンクします。

```
$ icpx -fsycl -foffload-static-lib=libstlib.a a.o -o a.exe
```

4.7.3 GPU アーキテクチャー向けの事前 (AOT) コンパイル

次のコマンドは、特定の GPU ターゲット向けに `app.out` を生成します。

DPC++:

```
$ icpx -fsycl-targets=spir64_gen -Xs "-device <device name>" a.cpp b.cpp -o app.out
```

OpenMP* オフロード:

```
$ icpx -fiopenmp -fopenmp-targets=spir64_gen -Xopenmp-target-backend "-device <device name>"
a.cpp b.cpp -o app.out
```

デバイス名に利用できる値の一覧は、『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) から入手できます。

4.8 FPGA 手順

フィールド・プログラマブル・ゲートアレイ (FPGA) は、任意の機能を実行するように繰り返しプログラム可能な集積回路です。FPGA は、空間コンピューティング・アーキテクチャーとして分類され、CPU や GPU のような固定命令セット・アーキテクチャー (ISA) デバイスとは異なり、従来のアクセラレーター・デバイスとは異なる最適化のトレードオフが求められます。

CPU、GPU、または FPGA 向けに SYCL* コードをコンパイルできますが、FPGA 開発用のコンパイル手順は、CPU や GPU 開発におけるコンパイル手順とは多少異なります。

FPGA 手順の詳細については、インテル® oneAPI FPGA ハンドブック (英語) を参照してください。

ヒント: GitHub の FPGA 向け oneAPI サンプル (英語) を確認して、FPGA デバイスの SYCL* プログラミングを学習できます。

FPGA プログラミングの詳細は、https://link.springer.com/chapter/10.1007/978-1-4842-5574-2_17 (英語) から入手できる『Data Parallel C++』書籍からも学ぶことができます。

4.8.1 FPGA 向けのコンパイルが特殊である理由

FPGA はいくつかの点で CPU や GPU とは異なります。CPU や GPU と比較した大きな違いは、FPGA ハードウェア専用のデバイスバイナリーが必要なことです。これはほとんどの場合、計算集約型で時間がかかる処理です。FPGA のコンパイルが完了するまで数時間かかるのは通常の動作です。そのため、FPGA 開発向けに事前 (オフライン) カーネル・コン

パイル・モードのみがサポートされます。FPGA ハードウェアのコンパイルには時間がかかるため、ジャストインタイム (オンライン) コンパイルは実用的ではありません。

コンパイル時間が長くなるほど、開発者の生産性は悪化します。インテル® oneAPI DPC++/C++ コンパイラーは、FPGA をターゲットとする開発設計を素早く反復できるメカニズムを提供します。可能な限り完全な FPGA コンパイルにかかる手順を回避することで、CPU および GPU 開発に近い感覚で高速コンパイル時間の恩恵を受けられます。

FPGA 手順の詳細については、インテル® oneAPI FPGA ハンドブック (英語) を参照してください。

4.8.2 SYCL* FPGA コンパイルの種別

SYCL* はアクセラレーター全般をサポートします。インテル® oneAPI DPC++/C++ コンパイラーは、FPGA コードの開発を支援するため、FPGA 固有のサポートを提供します。このドキュメントでは、インテル® oneAPI ベース・ツールキットがサポートするさまざまな FPGA コンパイル手順について説明します。

以下の表に、FPGA コンパイルの要約を示します。

FPGA コンパイルの種別

デバイス・イメージ・タイプ	コンパイル時間	説明
FPGA エミュレーター	数秒	FPGA デバイスコードは CPU 向けにコンパイルされます。OpenCL* ソフトウェア向けインテル® FPGA エミュレーション・プラットフォームを使用して、SYCL* コードが正しく機能することを確認します。
最適化レポート	数分	FPGA デバイスコードは部分的にハードウェア向けにコンパイルされます。コンパイラーは、FPGA で生成される構造とパフォーマンスのボトルネックを特定し、リソース利用率を推測して最適化レポートを生成します。コンパイルで FPGA デバイスファミリーまたは製品番号をターゲットにする場合、このステージでコード内の IP コンポーネントの RTL ファイルも作成されます。その後、インテル® Quartus® Prime 開発ソフトウェアを使用して、IP コンポーネントをさらに大きな設計に統合できます。
FPGA シミュレーター	数分	FPGA デバイスコードは CPU にコンパイルされます。Questa*-インテル® FPGA Edition Software のシミュレーターを使用して、コードをデバッグします。
FPGA ハードウェア・イメージ	数時間	ターゲットの FPGA プラットフォームで実行する FPGA ビットストリームを生成します。コンパイルで FPGA デバイスファミリーまたは製品番号をターゲットにする場合、このステージでコード内の IP コンポーネントの RTL ファイルも作成されます。その後、インテル® Quartus® Prime 開発ソフトウェアを使用して、IP コンポーネントをさらに大きな設計に統合できます。

一般的な FPGA 開発ワークフローでは、エミュレーション、最適化レポート、およびシミュレーション・ステージを繰り返して、それぞれのステージからもたらされるフィードバックを反映してコードを改良します。可能な限りエミュレーションと FPGA 最適化レポートを活用することを推奨します。

IP コンポーネントの開発時にこれらのステージがどのように適用されるかは、「FPGA IP オーサリング手順」を参照してください。

ヒント:

FPGA エミュレーション向けにコンパイルしたり、FPGA 最適化レポートを生成するには、インテル® oneAPI ベース・ツールキットのインテル® oneAPI DPC++/C++ コンパイラーのみが必要です。FPGA ハードウェアのコンパイルには、インテル® Quartus® Prime 開発ソフトウェアを別途インストールする必要があります。ボードをターゲットにする場合、ボード用の BSP をインストールする必要があります。

詳細については、『インテル® oneAPI ツールキット・インストール・ガイド』(英語)とインテル® FPGA 開発手順のページ(英語)をご覧ください。

また、IP コンポーネントの RTL コードを生成するには、インテル® oneAPI ベース・ツールキットに含まれる、インテル® oneAPI DPC++/C++ コンパイラーのみが必要です。ただし、IP コンポーネントをハードウェア設計でシミュレーションまたは統合するには、インテル® Quartus® Prime 開発ソフトウェア・プロ・エディションをインストールする必要があります。

4.8.2.1. FPGA エミュレーター

FPGA エミュレーター (OpenCL* ソフトウェア向けインテル® FPGA エミュレーション・プラットフォーム) は、コードの正当性を検証する最速の手法です。CPU 上で SYCL* デバイスコードを実行します。FPGA エミュレーターは、SYCL* ホストデバイスに似ていますが、ホストデバイスとは異なり、FPGA パイプや `fpga_reg` など FPGA 拡張機能がサポートされます。詳細については、『インテル® oneAPI ツールキット向け FPGA 最適化ガイド』の「パイプ拡張」(英語) と「カーネル変数」(英語) を参照してください。

次に FPGA エミュレーターを使用する際に覚えておくべき重要事項を示します。

- **パフォーマンスは典型的なものではない。**

  FPGA エミュレーターは、FPGA デバイスの動作そのものを再現するものではないため、パフォーマンスの評価に使用しないでください。例えば、FPGA で 100 倍のパフォーマンス向上をもたらす最適化は、エミュレーターのパフォーマンスには影響しないか、多少の増減を示すことがあります。

- **未定義の動作は異なる可能性があります。**

  FPGA エミュレーターと FPGA ハードウェア向けにコンパイルしたコードの結果が異なる場合、コードに未定義の動作が含まれる可能性があります。未定義の動作は言語仕様では指定されておらず、ターゲットごとに振る舞いが異なる可能性があります。

フルスタック・アクセラレーション・カーネルのエミュレーションの詳細については、「カーネルのエミュレート」を参照してください。

IP コンポーネントのエミュレーションの詳細については、「IP コンポーネントのエミュレーションとデバッグ」を参照してください。

4.8.2.2. FPGA 最適化レポート

完全な FPGA コンパイルは次のステージで行われ、最適化レポートは両方のステージの後に出力されます。

FPGA 最適化レポート

ステージ	説明	最適化レポートの情報
FPGA 初期イメージ (コンパイルには数分かかります)	SYCL* デバイスコードは最適化され、Verilog レジスター転送レベル (RTL) (FPGA の低レベルの設計入力言語) で指定される FPGA 設計に変換されます。これにより、実行形式ファイではない FPGA 初期イメージが生成されます。 このステージでは静的な最適化レポートが生成されます。	これには、コンパイラーが SYCL* デバイスコードから FPGA 設計に変換した手法に関する重要な情報が含まれています。レポートには次の情報が含まれます。 • FPGA で生成された構造の可視化情報 • パフォーマンスと予期されるパフォーマンスのボトルネックに関する情報 • リソースの利用状況を推測 FPGA の最適化レポートについては、『インテル® oneAPI FPGA ハンドブック』(英語) を参照してください。
FPGA ハードウェア・イメージ (コンパイルには数時間かかります)	設計回路のトポロジーを指定する Verilog RTL は、インテル® Quartus® Prime 開発ソフトウェア・プロ・エディションソフトウェアによって FPGA のプリミティブ・ハードウェア・リソースにマッピングされます。結果は、FPGA ハードウェア・バイナリー (ビットストリーム) です。	リソースと f_{MAX} 数に関する正確な情報が含まれます。レポートの解析に関する詳細は、『インテル® oneAPI ツールキット向け FPGA 最適化ガイド』の「設計の解析」(英語) を参照してください。 FPGA の最適化レポートについては、『インテル® oneAPI FPGA 最適化ハンドブック』(英語) を参照してください。

コンパイルで FPGA デバイスまたは製品番号をターゲットにする場合、このステージでコード内の IP コンポーネントの RTL ファイルも作成されます。その後、インテル® Quartus® Prime 開発ソフトウェアを使用して、IP コンポーネントをさらに大きな設計に統合できます。

4.8.2.3. FPGA シミュレーター

シミュレーション手順では、Questa*-インテル® FPGA Edition Software のシミュレーターを使用して、合成されたカーネルの正確な動作をシミュレートできます。エミュレーションと同様に、ターゲットの FPGA ボードが装着されていないシステムでシミュレーションを実行できます。シミュレーターは、エミュレーションよりも正確にカーネルをモデル化できますが、エミュレーターよりもかなり低速です。

シミュレーション手順は、サイクル精度とビット精度を目的とし、カーネルのデータパスの動作と浮動小数点データ型の操作結果を正確にモデル化します。ただし、シミュレーションでは、可変レイテンシー・メモリーやその他の外部インターフェイスを正確にモデル化することはできません。シミュレーションは FPGA ハードウェアやエミュレーターよりもかなり低速であるため、小規模な入力データセットで設計をシミュレーションすることを推奨します。

シミュレーション手順をプロファイルと組み合わせて利用することで、設計に関する追加情報を取得できます。プロファイルの詳細については、『インテル® oneAPI FPGA 最適化ハンドブック』の「DPC++ 向けインテル® FPGA ダイナミック・プロファイラー」(英語) を参照してください。

注: GNU* Project Debugger (GDB)、Microsoft* Visual Studio* または通常のソフトウェア・デバッガーを使用して、シミュレーション向けにコンパイルされたカーネルコードをデバッグすることはできません。

シミュレーション手順の詳細については、『インテル® oneAPI FPGA 最適化ハンドブック』のシミュレーションによるカーネルの評価を参照してください。

4.8.2.4. FPGA ハードウェア

FPGA ハードウェアのコンパイルには、インテル® Quartus® Prime 開発ソフトウェア (英語) を別途インストールする必要があります。これは、次のいずれかをターゲットにする FPGA ハードウェア・イメージの完全なコンパイルステージです。

- インテル® FPGA デバイスファミリー
- 特定のインテル® FPGA デバイスの製品番号
- サポートされる BSP を持つカスタムボード
- インテル® FPGA プログラマブル・アクセラレーション・カード (インテル® FPGA PAC) 非推奨

ターゲットの詳細については、「インテル® oneAPI DPC++/C++ コンパイラーのシステム要件」(英語) を参照してください。インテル® FPGA PAC またはカスタムボードの使用に関する詳細は、「FPGA BSP とボード」の節と、『インテル® oneAPI ツールキット・インストール・ガイド (Linux*)』(英語) を参照してください。

5 API ベースのプログラミング

最適化されたアプリケーション向けの特殊 API を提供することで、プログラミング・プロセスを簡素化するインテル® oneAPI ツールキットのライブラリーを利用できます。この章では、サンプルコードを含むライブラリーの基本的な情報と、特定の利用ケースでどのライブラリーが最も有効であるか判断するのに役立つ情報を提供します。利用可能な API など、それぞれのライブラリーの詳細は、ライブラリーのドキュメントをご覧ください。

oneAPI ツールキットのライブラリー

ライブラリー	使用法
インテル® oneAPI DPC++ ライブラリー (oneDPL)	ハイパフォーマンスの並列アプリケーションで使用します。
インテル® oneAPI マス・カーネル・ライブラリー (oneMKL)	高度に最適化および並列化された数学ルーチンをアプリケーションで利用できます。
インテル® oneAPI スレッディング・ビルディング (oneTBB)	マルチコア CPU でのインテル® TBB ベースの並列処理と、アプリケーションでの SYCL* デバイス高速並列処理を組み合わせます。
インテル® データ・アナリティクス・ライブラリー (oneDAL)	ビッグデータ解析アプリケーションと分散計算を高速化します。
インテル® コレクティブ・コミュニケーション・ライブラリー (oneCCL)	ディープラーニングとマシンラーニングのワークロードを処理するアプリケーションで使用します。
インテル® oneAPI ディープ・ニューラル・ネットワーク・ライブラリー (oneDNN)	インテル® アーキテクチャー・ベースのプロセッサーおよびインテル® プロセッサー・グラフィックス向けに最適化された、ニューラル・ネットワークを使用するディープラーニング・アプリケーションで使用します。

注: oneAPI サンプルコードの完全なリストは、GitHub* にある oneAPI サンプルカタログ (英語) をご覧ください。これらのサンプルは、CPU、GPU、FPGA をターゲットとしたマルチアーキテクチャー・アプリケーションの開発、オフロード、最適化を支援するように作成されています。

5.1 インテル® oneAPI DPC++ ライブラリー (インテル® oneDPL)

インテル® oneAPI DPC++ ライブラリー (oneDPL) は、インテル® oneAPI DPC++/C++ コンパイラーと連携して生産性の高い API を提供します。これらの API を使用して、ハイパフォーマンスな並列アプリケーション向けに、デバイス全体で SYCL* プログラミングの労力を最小限に抑えることができます。

oneDPL は次のコンポーネントで構成されます。

- Parallel STL
  - Parallel STL の使用手順
  - マクロ
- ライブラリー・クラスと関数の追加セット (このドキュメントでは**拡張 API** と呼びます)。

  - 並列アルゴリズム
  - イテレーター
  - 関数オブジェクト・クラス
  - 範囲ベースの API
- テスト済みの標準 C++ API
- 乱数ジェネレーター

5.1.1 インテル® oneDPL ライブラリーの使い方

oneDPL を使用するには、インテル® oneAPI ベース・ツールキット (英語) をインストールします。

Patallel STL と API 拡張を使用するには、必要なヘッダーファイルをソースコードでインクルードします。すべての oneDPL ヘッダーファイルは `oneapi/dpl` ディレクトリーにあります。それらをインクルードするには、`#include <oneapi/dpl/…>` を使用します。oneDPL には、大部分のクラスと関数対する名前空間 `oneapi::dpl` があります。

テスト済みの C++ 標準 API を使用するには、対応する C++ ヘッダーファイルをインクルードし、`std` 名前空間を使用する必要があります。

5.1.2 インテル® oneDPL サンプルコード

oneDPL のサンプルコードは、https://github.com/oneapi-src/oneAPI-samples/tree/master/Libraries/oneDPL (英語) から入手できます。それぞれのサンプルには、ビルド手順が説明された Readme が含まれています。それぞれのサンプルには、ビルド手順が説明された `Readme` が含まれています。

5.2 インテル® oneAPI マス・カーネル・ライブラリー (インテル® oneMKL)

インテル® oneAPI マス・カーネル・ライブラリー (oneMKL) は、最大限のパフォーマンスを必要とするアプリケーション向けに最適化され、広範囲に並列化されたルーチンの数学計算ライブラリーです。oneMKL には、C/Fortran プログラミング言語インターフェイスを備えた CPU アーキテクチャー向けのインテル® マス・カーネル・ライブラリー (インテル® MKL) のハイパフォーマンスな最適化が含まれており、各種 CPU アーキテクチャーとインテル® グラフィックス・テクノロジーでパフォーマンスを高める SYCL* インターフェイスが追加されています。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、その他の機能を提供します

OpenMP* オフロードを使用して、インテル® GPU で標準の oneMKL 計算を実行できます。詳細については、「C インターフェイスの OpenMP* オフロード」(英語) および「Fortran インターフェイスの OpenMP* オフロード」(英語) を参照してください。

CPU と GPU アーキテクチャー向けに最適化された新しい SYCL* インターフェイスには、次のような計算機能が追加されています。

- BLAS と LAPACK 密線形代数ルーチン
- スパース BLAS スパース線形代数ルーチン
- 乱数生成器 (RNG)
- ベクトルの数学演算用に最適化されたベクトル数学 (VM) ルーチン
- 高速フーリエ変換 (FFT)

機能一覧、ドキュメント、サンプルコード、ダウンロードについては、oneMKL のウェブサイト (英語) をご覧ください。oneMKL をインテル® oneAPI ベース・ツールキット (英語) の一部として利用する場合、有償オプションとして優先サポート (英語) を利用できます。インテルのコミュニティー・サポートについては、oneMKL フォーラム (英語) を参照してください。コミュニティーがサポートするオープンソース・バージョンについては、oneMKL GitHub* (英語) のページを参照してください。

次の表に、oneMKL サイトの違いを示します。

oneMKL の oneAPI 仕様 (英語)	パフォーマンスに優れた数学ライブラリー関数の DPC++ インターフェイスを定義します。oneMKL 仕様は、oneMKL 実装よりも頻繁に更新されます。
oneAPI マス・カーネル・ライブラリー (oneMKL) インターフェイス・プロジェクト (英語)	oneMKL 仕様のオープンソース実装です。このプロジェクトは、oneMKL 仕様で文書化された DPC++ インターフェイスをあらゆる数学ライブラリーに実装し、各種ターゲット・ハードウェアでの動作を実証することを目標としています。ここで提供される実装は、まだ完全ではない可能性があり、時間をかけて完全な仕様を構築することを目標としています。複数のハードウェア・ターゲットやほかの数学ライブラリーにサポートを拡張するため、コミュニティーにこのプロジェクトへの貢献を呼び掛けています。
インテル® oneAPI マス・カーネル・ライブラリー (oneMKL) プロジェクト (英語)	oneMKL 仕様のインテル製品実装 (DPC++ インターフェイスを使用) および同等の機能を提供する C と Fortran インターフェイスは、インテル® oneAPI ベース・ツールキットの一部として提供されます。インテル® CPU およびインテル® GPU ハードウェア向けに高度に最適化されています。

5.2.1 インテル® oneMKL の使い方

SYCL* インターフェイスを使用する場合、考慮すべきことがあります。

- oneMKL は、インテル® oneAPI DPC++/C++ コンパイラーおよび oneDPL に依存しています。アプリケーションは、インテル® oneAPI DPC++/C++ コンパイラーでビルドされ、SYCL* ヘッダーを利用しており、DPC++ リンカーを使用して oneMKL とリンクされている必要があります。
- oneMKL の SYCL* インターフェイスは、入力データ (ベクトル、行列など) にデバイスのアクセスが可能な統合共有メモリー (USM) ポインターを使用します。
- oneMKL の一部の SYCL* インターフェイスは、入力データ向けのデバイスへのアクセスが可能な USM ポインターのほかに、`sycl::buffer` オブジェクトのアクセスもサポートしています。

- oneMKL の SYCL* インターフェイスは、浮動小数点タイプに基づいてオーバーロードされます。標準ライブラリー・タイプ `std::complex<float>`、`std::complex<double>` を使用する、単精度実数引数 (`float`)、倍精度実数引数 (`double`)、半精度実数引数 (`half`)、および異なる精度の複素数引数を受け入れる汎用行列乗算 API があります。
- oneMKL の 2 レベルの名前空間構造が SYCL* インターフェイスに追加されます。

oneMKL の 2 レベルの名前空間

名前空間	説明
`oneapi::mkl`	oneMKL の各種ドメイン間の共通要素が含まれます。
`oneapi::mkl::blas`	密ベクトル-ベクトル、行列-ベクトル、および行列-行列の低レベル操作が含まれます。
`oneapi::mkl::lapack`	行列因数分解や固有値ソルバーなど高レベルの密行列演算が含まれます。
`oneapi::mkl::rng`	各種確率密度関数の乱数生成器が含まれます。
`oneapi::mkl::stats`	単精度および倍精度の多次元データセットの基本的な統計予測値が含まれます。
`oneapi::mkl::vm`	ベクトル数学ルーチンが含まれます。
`oneapi::mkl::dft`	高速フーリエ変換操作が含まれます。
`oneapi::mkl::sparse`	スパース行列ベクトル乗算、スパース三角ソルバーなどのスパース行列演算が含まれます。

5.2.2 インテル® oneMKL サンプルコード

SYCL* インターフェイスと oneMKL の一般的なワークフローを示すため、次のサンプルコードは、GPU デバイスで倍精度の行列-行列乗算を行います。

注: 次のコード例では、行中のコメントで示されるように、コンパイルと実行に追加のコードが必要です。

```
1.  // 標準 SYCL* ヘッダー
2.  #include <CL/sycl.hpp>
3.  // STL クラス
4.  #include <exception>
5.  #include <iostream>
6.  // oneMKL の SYCL*/DPC++ API の宣言
7.  #include "oneapi/mkl.hpp"
8.  int main(int argc, char *argv[]) {
9.      //
10.     // ユーザーは、m、n、k、ldA、ldB、ldC の設定と A、B、C 行列のデータを取得
11.     //
12.     // A、B、および C は、data() と size() メンバー関数を含む std::vector などの
13.     //    コンテナに格納します
14.     //
15.
16.     // GPU デバイスを作成
17.     sycl::device my_device;
18.     try {
19.         my_device = sycl::device(sycl::gpu_selector());
20.     }
```

```
21.     catch (...) {
22.         std::cout << "Warning: GPU device not found! Using default device instead." <<
    std::endl;
23.     }
24.     // キューにアタッチする非同期例外ハンドラーを作成
25.     // 必須ではありませんが、システムが適切に構成されていない場合に参照できる有用な情報が提供されます
26.     auto my_exception_handler = [](sycl::exception_list exceptions) {
27.         for (std::exception_ptr const& e : exceptions) {
28.             try {
29.                 std::rethrow_exception(e);
30.             }
31.             catch (sycl::exception const& e) {
32.                 std::cout << "Caught asynchronous SYCL exception:\n"
33.                     << e.what() << std::endl;
34.             }
35.             catch (std::exception const& e) {
36.                 std::cout << "Caught asynchronous STL exception:\n"
37.                     << e.what() << std::endl;
38.             }
39.         }
40.     };
41.     // 例外ハンドラーがアタッチされた gpu デバイスで実行キューを作成
42.     sycl::queue my_queue(my_device, my_exception_handler);
43.     // デバイスとホスト間のオフロード向けに行列データの SYCL* バッファーを作成
44.     sycl::buffer<double, 1> A_buffer(A.data(), A.size());
45.     sycl::buffer<double, 1> B_buffer(B.data(), B.size());
46.     sycl::buffer<double, 1> C_buffer(C.data(), C.size());
47.     // add oneapi::mkl::blas::gemm を実行キューに追加し、同期例外をキャッチ
    exceptions
48.     try {
49.         using oneapi::mkl::blas::gemm;
50.         using oneapi::mkl::transpose;
51.         gemm(my_queue, transpose::nontrans, transpose::nontrans, m, n, k, alpha,
    A_buffer, ldA, B_buffer,
52.             ldB, beta, C_buffer, ldC);
53.     }
54.     catch (sycl::exception const& e) {
55.         std::cout << "\t\tCaught synchronous SYCL exception during GEMM:\n"
56.             << e.what() << std::endl;
57.     }
58.     catch (std::exception const& e) {
59.         std::cout << "\t\tCaught synchronous STL exception during GEMM:\n"
60.             << e.what() << std::endl;
61.     }
62.     // 続行する前に、キャッチされた非同期例外を処理
63.     my_queue.wait_and_throw();
64.     //
65.     // 後処理の結果
66.     //
67.     // C バッファーからデータをアクセスし、C 行列の一部を出力
68.     auto C_accessor = C_buffer.template get_access<sycl::access::mode::read>();
69.     std::cout << "\t" << C << " = [ " << C_accessor[0] << ", "
70.         << C_accessor[1] << ", ... ]\n";
71.     std::cout << "\t    [ " << C_accessor[1 * ldC + 0] << ", "
72.         << C_accessor[1 * ldC + 1] << ",  ... ]\n";
73.     std::cout << "\t    [ " << "... ]\n";
74.     std::cout << std::endl;
75.
76.     return 0;
```

```
77. }
```

(倍精度値) 行列 A (サイズ m * k)、B (サイズ k * n)、C (サイズ m * n) は、ホストマシンの配列 (次元 ldA、ldB、ldC) に格納されると想定します。スカラー (倍精度) alpha と beta を指定し、行列-行列乗算 (mkl::blas::gemm) を計算します。

```
C = alpha * A * B + beta * C
```

標準 SYCL* ヘッダーと対象の mkl::blas::gemm API 宣言を含む oneMKL SYCL*/DPC++ 固有ヘッダーをインクルードします。

```
// 標準 SYCL ヘッダー
#include <CL/sycl.hpp>
// STL クラス
#include <exception>
#include <iostream>
// インテル® oneAPI マス・カーネル・ライブラリーの SYCL/SYCL ++API の宣言
#include "oneapi/mkl.hpp"
```

次に、通常のようにホストマシンで行列データをロードまたはインスタンス化し、GPU デバイスを作成して非同期例外ハンドラーを作成し、最後に例外ハンドラーでデバイスキューを作成します。ホストで発生する例外は、標準 C++ 例外メカニズムでキャッチできます。ただし、デバイスで発生する例外は非同期エラーとして見なされ、例外リストに保存されて、ユーザー定義の例外ハンドラーによって処理されます。

```
// GPU デバイスを作成
sycl::device my_device;
try {
    my_device = sycl::device(sycl::gpu_selector());
}
catch (...) {
    std::cout << "Warning: GPU device not found! Using default device instead."<< std::endl;
}
// キューにアタッチする非同期例外ハンドラーを作成
// 必須ではありませんが、システムが適切に構成されていない場合に参照できる有用な情報が提供されます
auto my_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:\n"
                << e.what() << std::endl;
        }
        catch (std::exception const& e) {
            std::cout << "Caught asynchronous STL exception:\n"
                << e.what() << std::endl;
        }
    }
};
```

これで、行列データが SYCL* バッファーにロードされ、ターゲットのデバイスにオフロードして、完了後にホストに戻すことができます。最後に、mkl::blas::gemm API がすべてのバッファー、サイズ、および転置操作で呼び出され、行列乗算カーネルとデータをターゲットにキューに追加します。

```
// 例外ハンドラーがアタッチされた gpu デバイスで実行キューを作成
sycl::queue my_queue(my_device, my_exception_handler);
// デバイスとホスト間のオフロード向けに行列データの SYCL バッファーを作成
sycl::buffer<double, 1> A_buffer(A.data(), A.size());
sycl::buffer<double, 1> B_buffer(B.data(), B.size());
sycl::buffer<double, 1> C_buffer(C.data(), C.size());
// add oneapi::mkl::blas::gemm を実行キューに追加し、同期例外をキャッチ
try {
    using oneapi::mkl::blas::gemm;
    using oneapi::mkl::transpose;
    gemm(my_queue, transpose::nontrans, transpose::nontrans, m, n, k, alpha, A_buffer, ldA,
B_buffer,
       ldB, beta, C_buffer, ldC);
}
catch (sycl::exception const& e) {
    std::cout << "\t\tCaught synchronous SYCL exception during GEMM:\n"
        << e.what() << std::endl;
}
catch (std::exception const& e) {
    std::cout << "\t\tCaught synchronous STL exception during GEMM:\n"
        << e.what() << std::endl;
}
```

gemm カーネルがキューに登録された後、実行されます。キューは、すべてのカーネルの実行を待機し、キャッチされた非同期例外を例外ハンドラーに渡してスローするように要求します。ランタイムは、ホストと GPU デバイス間のバッファーのデータ転送を処理します。C_buffer のアクセサーが作成されるまでに、バッファーのデータはホストマシンに暗黙的に転送されます。この場合、アクセサーは 2 x 2 の C_buffer のサブ行列を出力するために使用されます。

```
// C バッファーからデータをアクセスし、C 行列の一部を出力
auto C_accessor = C_buffer.template get_access<sycl::access::mode::read>();
std::cout << "\t" << C << " = [ " << C_accessor[0] << ", "
    << C_accessor[1] << ", ...]\n";
std::cout << "\t    [ " << C_accessor[1 * ldC + 0] << ", "
    << C_accessor[1 * ldC + 1] << ",  ...]\n";
std::cout << "\t    [ " << "...]\n";
std::cout << std::endl;

return 0;
```

結果データは C_buffer オブジェクトにあり、明示的にどこかにコピーしない限り (元の C コンテナに戻すなど)、C_buffer がスコープ外になるまでアクセサーを介してのみ利用できます。

5.3 インテル® oneAPI スレッディング・ビルディング・ブロック (インテル® oneTBB)

インテル® oneAPI スレッディング・ビルディング (oneTBB) は、ホスト上でタスクベースの、共有メモリー並列プログラミングを可能にする、広く使用されている C++ ライブラリーです。このライブラリーは、SYCL* および ISO C++ で利用可能な機能のほかに、CPU 上での並列プログラミング向けに次のような機能を提供します。

- 汎用並列アルゴリズム
- コンカレント・コンテナ
- スケラーブル・メモリー・アロケーター
- ワークスチール・タスク・スケジューラー
- 低レベル同期プリミティブ

oneTBB はコンパイラーに依存せず、さまざまなプロセッサーとオペレーティング・システムで利用できます。CPU 向けのマルチスレッド並列処理を実現するため、ほかの oneAPI ライブラリー (インテル® oneAPI マス・カーネル・ライブラリー、インテル® oneAPI ディープ・ニューラル・ネットワーク・ライブラリーなど) で使用されています。

機能一覧、ドキュメント、サンプルコード、ダウンロードについては、インテル® oneAPI スレッディング・ビルディングのウェブサイト (英語) をご覧ください。oneTBB をインテル® oneAPI ベース・ツールキット (英語) の一部として利用する場合、有償オプションとして優先サポート (英語) を利用できます。インテルのコミュニティー・サポートについては、oneTBB (英語) を参照してください。コミュニティーがサポートするオープンソース・バージョンについては、oneTBB GitHub* (英語) のページを参照してください。

5.3.1 インテル® oneTBB の使い方

oneTBB は、ほかの C++ コンパイラーでもインテル® oneAPI DPC++ コンパイラーと同じ方法で使用できます。詳細は、「oneTBB ドキュメント」(英語) をご覧ください。

現在、oneTBB はアクセラレーターを直接サポートしません。ただし、DPC++ 言語、OpenMP* オフロードのほかの oneAPI ライブラリーを組み合わせて、利用可能なすべてのハードウェア・リソースを効率良く使用するプログラムを構築できます。

5.3.2 インテル® oneTBB サンプルコード

2 つの基本的な oneTBB サンプルコードが、https://github.com/oneapi-src/oneAPI-samples/tree/master/Libraries/oneTBB (英語) で入手できます。これらのサンプルコードは、CPU と GPU 向けに記述されています。

- `tbb-async-sycl` は、oneTBB フローグラフ非同期ノードと機能ノードを使用し、計算カーネルを分割して CPU と GPU 間で実行する方法を示します。フローグラフ非同期ノードは、SYCL* を使用して機能ノードが計算の CPU 部分を実行する間に、GPU で実装された計算を実行します。

- `tbb-task-sycl` は、2 つの oneTBB タスクが同じ計算カーネルを実行する方法を示します。1 つのタスクが SYCL* コードを実行し、もう一方は oneTBB コードを実行します。
- `tbb-resumable-tasks-sycl` は、oneTBB 再開タスクと `parallel_for` を使用して、計算カーネルを分割して CPU と GPU で実行する方法を示します。再開可能なタスクは SYCL* を使用して GPU で計算を実装し、`parallel_for` は計算の CPU 部分を実行します。

5.4 インテル® oneAPI データ・アナリティクス・ライブラリー (インテル® oneDAL)

インテル® oneAPI データ・アナリティクス・ライブラリー (oneDAL) は、データ解析のすべてのステージ (前処理、変換、解析、モデリング、検証、意思決定) で、バッチ、オンライン、および分散処理モードで計算を実行する、高度に最適化されたアルゴリズムのビルディング・ブロックを提供することで、ビッグデータ解析の高速化を支援するライブラリーです。

アルゴリズムの計算だけでなくデータの取り込みを最適化し、スループットとスケーラビリティーを向上します。C++、および Java* API に加えて、Spark* や Hadoop* などの一般的なデータソースへのコネクターが含まれます。oneDAL の Python* ラッパーは、インテル® ディストリビューションの Python* (英語) に含まれます。

さらに従来の機能に加えて、oneDAL は従来の C++ インターフェイスに DPC++ API 拡張機能を提供し、一部のアルゴリズムで GPU も利用できるようにします。

このライブラリーは、分散計算では特に有用です。通信レイヤーから独立した分散アルゴリズムのビルディング・ブロックの完全なセットを提供します。これにより、ユーザーは利用したい通信基盤を使って、高速でスケーラブルな分散アプリケーションを構築できます。

機能一覧、ドキュメント、サンプルコード、ダウンロードについては、oneDAL のウェブサイト (英語) をご覧ください。oneDAL をインテル® oneAPI ベース・ツールキット (英語) の一部として利用する場合、有償オプションとして優先サポート (英語) を利用てきます。インテルのコミュニティー・サポートについては、oneDAL フォーラム (英語) を参照してください。コミュニティーがサポートするオープンソース・バージョンについては、oneDAL GitHub* (英語) のページを参照してください。

5.4.1 インテル® oneDAL の使い方

アプリケーションをビルドして oneDAL とリンクするのに必要な依存関係に関する情報は、「oneDAL のシステム要件」(英語) を参照してください。

oneDAL ベースのアプリケーションは、適切なデバイスセレクターを選択することで、CPU または GPU でアルゴリズムをシームレスに実行できます。新機能により次のことが可能になります。

- 数値テーブルから SYCL* バッファーを抽出してカスタムカーネルに渡す。
- SYCL* バッファーから数値テーブルを作成する。

アルゴリズムは、SYCL* バッファーを再利用して GPU データを保持し、GPU と CPU 間でデータを繰り返しコピーする過負荷を排除するように最適化されています。

5.4.2 インテル® oneDAL サンプルコード

oneDAL のサンプルコードは GitHub* リポジトリーから入手できます。次のサンプルコードは、oneDAL 固有の機能を示す分かりやすい例です。

https://github.com/oneapi-src/oneDAL/tree/master/examples/oneapi/dpc/source/svm (英語)

5.5 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)

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 (oneCCL) は、ディープラーニング (DL)、およびマシンラーニング (ML) ワークロード向けのスケーラブルでハイパフォーマンスな通信ライブラリーです。インテル® Machine Learning Scaling Library に由来するアイデアを発展させて、新しい機能と利用ケースを実現するため設計と API を拡張しています。

oneCCL は次の機能を備えています。

- 低レベルの通信ミドルウェア上に構築された MPI と `libfabrics`。
- 通信パフォーマンスに対する計算の生産的なトレードオフを可能にすることで、通信パターンのスケーラビリティーを促進する最適化。
- 優先順位、永続的な操作、アウトオブオーダー実行など、一連の DL 固有の最適化。
- CPU や GPU など各種ハードウェア・ターゲットで実行する DPC++ API。
- さまざまなインターコネクトで動作: インテル® Omni-Path アーキテクチャー (インテル® OPA)、InfiniBand*、イーサネット。

機能一覧、ドキュメント、サンプルコード、ダウンロードについては、oneCCL のウェブサイト (英語) をご覧ください。oneCCL をインテル® oneAPI ベース・ツールキット (英語) の一部として利用する場合、有償オプションとして優先サポート (英語) を利用できます。コミュニティーがサポートするオープンソース・バージョンについては、oneCCL GitHub* ページ (英語) を参照してください。

5.5.1 インテル® oneCCL の使い方

MPI やインテル® oneAPI DPC++/C++ コンパイラーなど、ハードウェアとソフトウェアの依存関係に関する完全なリストについては、「oneCCL のシステム要件」(英語) を参照してください。

SYCL* 対応 API は、oneCCL のオプション機能です。oneCCL ストリーム・オブジェクトを作成する場合、CPU と SYCL* バックエンドのどちらかを選択できます。

- CPU バックエンド: 最初の引数に `ccl_stream_host` を指定します。
- SYCL* バックエンド: デバイスタイプに応じて `ccl_stream_cpu` または `ccl_stream_gpu` を指定します。
- SYCL* ストリームで動作する集合操作
  - C API では、oneCCL は通信バッファーが `void*` にキャストされた `sycl::buffer` オブジェクトであると想定します。
  - C++ API では、oneCCL は通信バッファーが参照渡しされることを想定します。

使用法に関する詳しい説明は、https://oneapi-src.github.io/oneCCL/ (英語) から入手できます。

5.5.2 インテル® oneCCL サンプルコード

oneCCL のサンプルコードは、https://github.com/oneapi-src/oneAPI-samples/tree/master/Libraries/oneCCL (英語) から入手できます。

コードをビルドして実行する手順を含む入門サンプルは、同じ GitHub* リポジトリーから入手できます。

5.6 インテル® oneAPI ディープ・ニューラル・ネットワーク・ライブラリー (インテル® oneDNN)

インテル® oneAPI ディープ・ニューラル・ネットワーク・ライブラリー (oneDNN) は、ディープラーニング・アプリケーション向けのオープンソースのパフォーマンス・ライブラリーです。このライブラリーには、インテル® アーキテクチャー・ベースのプロセッサーおよびインテル® プロセッサー・グラフィックス向けに最適化されたニューラル・ネットワークの基本的な構成要素が含まれます。oneDNN は、インテル® アーキテクチャー・ベースのプロセッサーとインテル® プロセッサー・グラフィックス上でアプリケーションのパフォーマンス向上に注目するディープラーニング・アプリケーションおよびフレームワーク開発者を対象としています。ディープラーニングでは、oneDNN を利用するアプリケーションを使用すべきです。

oneDNN は、インテル® oneAPI DL フレームワーク・デベロッパー・ツールキット、インテル® oneAPI ベース・ツールキットの一部として配布され、`apt` または `yum` チャネルから入手できます。

oneDNN は、C および C++ インターフェイス、OpenMP*、oneTBB、OpenCL* ランタイムなど、DNNL で現在サポートされる機能を引き続きサポートします。oneDNN は、oneAPI プログラミング・モデルの DPC++ API とランタイムサポートを導入します。

機能一覧、ドキュメント、サンプルコード、ダウンロードについては、oneDNN ウェブサイト (英語) をご覧ください。oneDNN をインテル® oneAPI ベース・ツールキット (英語) の一部として利用する場合、有償オプションとして優先サ

ポート (英語) を利用できます。コミュニティーがサポートするオープンソース・バージョンについては、oneDNN GitHub* (英語) のページを参照してください。

5.6.1 インテル® oneDNN の使い方

oneDNN は、インテル® 64 アーキテクチャーまたは互換プロセッサーをベースにするシステムをサポートします。サポートされる CPU およびグラフィックス・ハードウェアのリストは、インテル® oneAPI ディープ・ニューラル・ネットワーク・ライブラリーのシステム要件を参照してください。

oneDNN は実行時に命令セット・アーキテクチャー (ISA) を検出し、オンライン生成機能を使用して、サポートされる最新の ISA 向けに最適化されたコードを展開します。

アプリケーションが使用する CPU または GPU ランタイム・ライブラリーとの相互運用性を保証するため、オペレーティング・システムごとにいくつかのパッケージが提供されます。

オペレーティング・システムごとのパッケージ

設定	依存関係
`cpu_dpcpp_gpu_dpcpp`	DPC++ ランタイム
`cpu_iomp`	インテルの OpenMP* ランタイム
`cpu_gomp`	GNU* OpenMP* ランタイム
`cpu_vcomp`	Microsoft* Visual C++* OpenMP* ランタイム
`cpu_tbb`	インテル® oneAPI スレッディング・ビルディング

パッケージには必要なライブラリーが含まれていないため、ビルド時にインテル® oneAPI ツールキットまたはサードパーティーのツールを使用してアプリケーションで依存関係を解決する必要があります。

SYCL* 環境では、oneDNN は DPC++ SYCL* ランタイムを介して CPU または GPU ハードウェアと対話します。oneDNN は SYCL* を使用するほかのコードを使用することもできます。これを可能にするため、oneDNN は基盤となる SYCL* オブジェクトと相互作用する API 拡張機能を提供します。

そのような状況の 1 つとして、oneDNN が提供しないカスタム操作を行うため SYCL* カーネルを実行する場合が考えられます。その場合、oneDNN はカーネルをシームレスに送信するのに必要な API を提供し、同じデバイスキューを使用して実行コンテキストを oneDNN と共有します。

相互運用性を提供する API は、次の 2 つのシナリオを想定します。

- 既存の SYCL* オブジェクトをベースとする oneDNN オブジェクトの構築
- 既存の oneDNN オブジェクト向けの SYCL* オブジェクトへのアクセス

次の表に oneDNN オブジェクトと SYCL* オブジェクトのマッピングを示します。

oneDNN と SYCL* オブジェクトのマッピング 1

oneDNN オブジェクト	SYCL* オブジェクト
エンジン	`cl::sycl::device and cl::sycl::context`
ストリーム	`cl::sycl::queue`
メモリー	`cl::sycl::buffer<uint8_t, 1>` または統合共有メモリー (USM) ポインター

注: 内部的にライブラリー・メモリー・オブジェクトは、1D `uint8_t` SYCL* バッファーを使用しますが、異なるタイプの SYCL* バッファーを介してメモリーを初期化およびアクセスできます。この場合、バッファーは異なるタイプの `cl::sycl::buffer<uint8_t, 1>` に再解釈されます。

oneDNN と SYCL* オブジェクトのマッピング 2

oneDNN オブジェクト	SYCL* オブジェクトからの再構成
エンジン	`dnnl::sycl_interop::make_engine(sycl_dev, sycl_ctx)`
ストリーム	`dnnl::sycl_interop::make_stream(engine, sycl_queue)`
メモリー	USM ベース: `dnnl::memory(memory_desc, engine, usm_ptr)` バッファーベース: `dnnl::sycl_interop::make_memory(memory_de sc, engine, sycl_buf)`

oneDNN と SYCL* オブジェクトのマッピング 3

oneDNN オブジェクト	SYCL* オブジェクトの抽出
エンジン	`dnnl::sycl_interop::get_device(engine)` `dnnl::sycl_interop::get_context(engine)`
ストリーム	`dnnl::sycl_interop::get_queue(stream)`
メモリー	USM ポインター: `dnnl::memory::get_data_handle()` バッファー: `dnnl::sycl_interop::get_buffer(memory)`

注:

- oneDNN でアプリケーションをビルドするには、コンパイラーが必要です。インテル® oneAPI DPC++/C++ コンパイラーは、インテル® oneAPI ベース・ツールキットに含まれています。
- SYCL* 相互運用性 API を有効にするには、`dnnl_sycl.hpp` をインクルードする必要があります。
- OpenMP* はランタイム・オブジェクトの受け渡しに依存しないため、oneDNN と連携する相互運用 API は必要ありません。

5.6.2 インテル® oneDNN サンプルコード

oneDNN のサンプルコードは、https://github.com/oneapi-src/oneAPI-samples/tree/master/Libraries/oneDNN (英語) から入手できます。導入向けのサンプルは新規ユーザーを対象としており、ビルドコマンドと実行コマンドの例を含む `readme` ファイルが含まれています。

5.7 その他のライブラリー

その他のライブラリーは、各種インテル® oneAPI ツールキットに含まれます。各ライブラリーの詳細については、それぞれのライブラリーの公式ドキュメントをご覧ください。

- インテル® インテグレーテッド・パフォーマンス・プリミティブ (インテル® IPP)
- インテル® MPI ライブラリー
- インテル® オープン・ボリューム・カーネル・ライブラリー

6 ソフトウェア開発プロセス

oneAPI プログラミング・モデルを使用したソフトウェア開発プロセスは、標準の開発プロセスをベースにしています。プログラミング・モデルは、アクセラレーターを使用してパフォーマンスを向上するため、この節ではその作業に固有の手順を説明します。これには以下のものがあります。

- パフォーマンス・チューニング・サイクル
- コードのデバッグ
- ほかのアクセラレーター向けのコードの移行
- コードのコンポーザビリティー

6.1 SYCL* と DPC++ へのコードの移行

C++ または OpenCL* などほかのプログラミング言語で記述されたコードは、複数のデバイスで使用するため DPC++ コンパイラーでコンパイルされる SYCL* コードへ移行できます。移行手順は、元の言語によって異なります。

6.1.1 C++ から SYCL* への移行

SYCL* は、C++ をベースとする「単一ソース」スタイルのプログラミング・モデルです。C++17 と C++20 の機能を基に構築され、ヘテロジニアス・プログラミングにおけるオープン、マルチベンダー、およびマルチアーキテクチャーのソリューションをサポートします。

DPC++ コンパイラー・プロジェクトは、SYCL* を LLVM C++ コンパイラーに導入し、複数のベンダーとアーキテクチャーに対応したハイパフォーマンスな実装を実現しています。

既存の C++ アプリケーションを高速化する場合、大部分の C++ コードは変更する必要がないため、SYCL* はシームレスな統合を可能にします。デバイス側のコンパイルを可能にする SYCL* の構造については、「oneAPI プログラミング・モデル」をご覧ください。

6.1.2 DPC++ コンパイラーを使用した CUDA* から SYCL* への移行

インテル® DPC++ 互換性ツール (インテル® DPCT) は、インテル® oneAPI ベース・ツールキットに含まれます。このツールの目的は、NVIDIA* CUDA* で記述された既存のプログラムから、DPC++ コンパイラーでコンパイルされる SYCL* で記述されたプログラムへの移行を支援することです。このツールは、可能な限り SYCL* コードを自動生成します。しかし、すべてのコードが自動的に移行されるとは限らず、手動での変更が必要な場合があります。このツールには、IDE プラグインヘルプと、DPC++ への移行を完了するためのユーザーガイド (英語) を含まれます。手動での変更が完了したら、DPC++ コンパイラーを使用して実行可能ファイルを作成します。

インテル® DPC++ 互換性ツールを使用した CUDA* から SYCL* への移行

- 移行されたコード例、ツールのダウンロード手順などの説明は、インテル® DPC++ 互換性ツールのウェブサイト (英語) でご覧いただけます。
- ツールの詳しい使用方法は、『インテル® DPC++ 互換性ツール・ユーザーガイド』(英語) を参照してください。

6.1.3 OpenCL* コードから SYCL* への移行

現在の DPC++ プロジェクトの SYCL* ランタイムは、並列処理を実現するため OpenCL* コードを採用しています。通常、SYCL* ではカーネルを実装するコードの行数と必要な API 関数やメソッドの呼び出しは少なくなります。ホストのソースコード行にデバイスのソースコードを埋め込むことで、OpenCL* プログラムを作成できます。

ほとんどの OpenCL* アプリケーション開発者は、デバイスへのカーネルオフロードに伴うセットアップ・コードの必要性を認識しているでしょう。SYCL* を使用すると、OpenCL* C コードに関連する大部分のセットアップ・コードなしで、シンプルで現代的な C++ ベースのアプリケーションを開発できます。これにより、習得の労力が軽減され、並列化の実装に集中できます。

さらに、OpenCL* アプリケーションの機能は、SYCL* API を介して引き続き利用できます。更新されたコードは、必要に応じて SYCL* インターフェイスを使用できます。

6.1.4 CPU、GPU、および FPGA 間の移行

DPC++ コンパイラーを使用した SYCL* では、プラットフォームは CPU、GPU、FPGA、またはその他のアクセラレーターやプロセッサーなどのデバイス (なくてもかまいません) に接続されたホストデバイスで構成されます。

プラットフォームに複数のデバイスが存在する場合、一部または大部分のワークをデバイスへオフロードするようにアプリケーションを設計します。oneAPI プログラミング・モデルで複数のデバイスにワークを分散するには、いくつかの方法があります。

1. デバイスセレクターの初期化 - SYCL* ではセレクターと呼ばれるクラスが提供され、プラットフォーム内のデバイスを手動で選択するか、oneAPI ランタイムのヒューリスティックがデバイスで利用可能な計算機能を判断してデフォルトデバイスを選択できるようになっています。

2. データセットの分割 - データに依存関係がない高い並列性を持つアプリケーションでは、データセットを明示的に分割して異なるデバイスで利用します。次のサンプルコードは、複数のデバイスにワークロードを分配する例です。`icpx -fsycl snippet.cpp` コマンドでコードをコンパイルします。

```
1.  int main() {
2.    int data[1024];
3.    for (int i = 0; i < 1024; i++)
4.      data[i] = i;
5.    try {
6.      cpu_selector cpuSelector;
7.      queue cpuQueue(cpuSelector);
8.      gpu_selector gpuSelector;
9.      queue gpuQueue(gpuSelector);
10.     buffer<int, 1> buf(data, range<1>(1024));
11.     cpuQueue.submit([&](handler& cgh) {
12.       auto ptr =
13.       buf.get_access<access::mode::read_write>(cgh);
14.       cgh.parallel_for<class divide>(range<1>(512),
15.         [=](id<1> index) {
16.           ptr[index] -= 1;
17.       });
18.     });
19.     gpuQueue.submit([&](handler& cgh1) {
20.       auto ptr =
21.       buf.get_access<access::mode::read_write>(cgh1);
22.       cgh1.parallel_for<class offset1>(range<1>(1024),
23.         id<1>(512), [=](id<1> index) {
24.           ptr[index] += 1;
25.       });
26.     });
27.     cpuQueue.wait();
28.     gpuQueue.wait();
29.   }
30.   catch (exception const& e) {
31.     std::cout <<
32.     "SYCL exception caught: " << e.what() << '\n';
33.     return 2;
34.   }
35.   return 0;
36. }
```

3. デバイス間で複数のカーネルをターゲットにする - アプリケーションに複数の独立したカーネルで並列処理可能なスコープがある場合、ターゲットデバイスごとに異なるキューを使用します。SYCL* でサポートされるプラットフォームとプラットフォームごとのデバイスのリストは、`get_platforms()` と `platform.get_devices()` を呼び出すことで取得できます。すべてのデバイスが特定されたら、デバイスごとにキューを作成し、異なるカーネルを異なるキューに配置します。次のサンプルコードは、複数の SYCL* デバイスにカーネルを配置する方法を示します。

```
1.  #include <stdio.h>
2.  #include <vector>
3.  #include <CL/sycl.hpp>
4.  using namespace cl::sycl;
5.  using namespace std;
6.  int main()
7.  {
8.    size_t N = 1024;
9.    vector<float> a(N, 10.0);
10.   vector<float> b(N, 10.0);
11.   vector<float> c_add(N, 0.0);
```

```
12.   vector<float> c_mul(N, 0.0);
13.   {
14.     buffer<float, 1> abuffer(a.data(), range<1>(N),
15.       { property::buffer::use_host_ptr() });
16.     buffer<float, 1> bbuffer(b.data(), range<1>(N),
17.       { property::buffer::use_host_ptr() });
18.     buffer<float, 1> c_addbuffer(c_add.data(), range<1>(N),
19.       { property::buffer::use_host_ptr() });
20.     buffer<float, 1> c_mulbuffer(c_mul.data(), range<1>(N),
21.       { property::buffer::use_host_ptr() });
22.   try {
23.     gpu_selector gpuSelector;
24.     auto queue = cl::sycl::queue(gpuSelector);
25.     queue.submit([&](cl::sycl::handler& cgh) {
26.       auto a_acc = abuffer.template
27.         get_access<access::mode::read>(cgh);
28.       auto b_acc = bbuffer.template
29.         get_access<access::mode::read>(cgh);
30.       auto c_acc_add = c_addbuffer.template
31.         get_access<access::mode::write>(cgh);
32.       cgh.parallel_for<class VectorAdd>
33.         (range<1>(N), [=](id<1> it) {
34.           //int i = it.get_global();
35.           c_acc_add[it] = a_acc[it] + b_acc[it];
36.         });
37.     });
38.     cpu_selector cpuSelector;
39.     auto queue1 = cl::sycl::queue(cpuSelector);
40.     queue1.submit([&](cl::sycl::handler& cgh) {
41.       auto a_acc = abuffer.template
42.         get_access<access::mode::read>(cgh);
43.       auto b_acc = bbuffer.template
44.         get_access<access::mode::read>(cgh);
45.       auto c_acc_mul = c_mulbuffer.template
46.         get_access<access::mode::write>(cgh);
47.       cgh.parallel_for<class VectorMul>
48.         (range<1>(N), [=](id<1> it) {
49.           c_acc_mul[it] = a_acc[it] * b_acc[it];
50.         });
51.     });
52.   }
53.   catch (cl::sycl::exception e) {
54.     /* In the case of an exception being throw, print the
55.        error message and
56.        * return 1. */
57.     std::cout << e.what();
58.     return 1;
59.   }
60.   }
61.   for (int i = 0; i < 8; i++) {
62.     std::cout << c_add[i] << std::endl;
63.     std::cout << c_mul[i] << std::endl;
64.   }
65.   return 0;
66. }
```

6.2 コンポーザビリティー

oneAPI プログラミング・モデルは、開発ツールチェーン全体をサポートするエコシステムを実現します。これには、CPU、GPU、FPGA など複数のアクセラレーターをサポートするコンパイラーとライブラリー、デバッガーと解析ツールが含まれます。

6.2.1 C/C++ OpenMP* および SYCL* のコンポーザビリティー

oneAPI プログラミング・モデルは、OpenMP* オフロードをサポートする LLVM/Clang ベースの統合コンパイラーを提供します。これにより、OpenMP* 構造を使用してホスト側のアプリケーションを並列化するか、ターゲットデバイスにオフロードするシームレスな統合が可能になります。インテル® oneAPI ベース・ツールキットで利用可能なインテル® oneAPI DPC++/C++ コンパイラーは、制限付きで OpenMP* と SYCL* の互換性を提供します。単一のアプリケーションでは、OpenMP* `target` 領域または SYCL* 構造を使用して、さまざまな関数やコードセグメントなどのコード領域の実行をデバイスにオフロードできます。

OpenMP* と SYCL* オフロード構造は、別々のファイル、同じファイル、または同じ関数 (制限付き) で使用できます。OpenMP* および SYCL* オフロードコードは、実行可能ファイル、静的ライブラリー、またはさまざまな組み合わせで一緒にバンドルできます。

> **注:** DPC++ 向けの SYCL* は CPU でデバイスコードを実行する場合、oneTBB のランタイムを使用します。そのため、CPU で OpenMP* と SYCL* の両方を使用すると、スレッドのオーバーサブスクリプションが発生する可能性があります。システムで実行されるワークロードのパフォーマンスを解析することで、この問題が生じているか確認することができます。

6.2.1.1. 制限事項

同じアプリケーションで OpenMP* と SYCL* 構造を混在する場合、いくつかの考慮すべき制限があります。

- OpenMP* ディレクティブは、デバイスで実行される SYCL* カーネル内では使用できません。同様に、SYCL* コードは、OpenMP* `target` 領域内では使用できません。ただし、ホスト CPU で実行される OpenMP* コード内で SYCL* 構造を使用することはできます。
- プログラム内の OpenMP* および SYCL* デバイス領域は相互依存関係を持つことができません。例えば、デバイスコードの SYCL* 領域で定義された関数は、同じデバイスで実行される OpenMP* コードから呼び出すことはできません (その逆も同様)。OpenMP* デバイス領域と SYCL* デバイス領域は個別にリンクされ、個別のバイナリーとしてコンパイラーによって生成されるファットバイナリーを構成します。
- 現時点では、OpenMP* と SYCL* ランタイム・ライブラリー間での直接的な相互作用はサポートされていません。例えば、OpenMP* API で生成されたデバイス・メモリー・オブジェクトは、SYCL* コードからはアクセスできません。OpenMP* で生成されたデバイス・メモリー・オブジェクトを SYCL* コードで使用すると、未定義の動作となります。

6.2.1.2. 例

次のコードは、SYCL* と OpenMP* オフロード構造を同じアプリケーションで使用する例を示します。

```
1.  #include <CL/sycl.hpp>
2.  #include <array>
3.  #include <iostream>
4.
5.
6.  float computePi(unsigned N) {
7.    float Pi;
8.  #pragma omp target map(from : Pi)
9.  #pragma omp parallel for reduction(+ : Pi)
10.   for (unsigned I = 0; I < N; ++I) {
11.     float T = (I + 0.5f) / N;
12.     Pi += 4.0f / (1.0 + T * T);
13.   }
14.   return Pi / N;
15. }
16.
17.
18. void iota(float *A, unsigned N) {
19.   cl::sycl::range<1> R(N);
20.   cl::sycl::buffer<float, 1> AB(A, R);
21.   cl::sycl::queue().submit([&](cl::sycl::handler &cgh) {
22.     auto AA = AB.template get_access<cl::sycl::access::mode::write>(cgh);
23.     cgh.parallel_for<class Iota>(R, [=](cl::sycl::id<1> I) {
24.       AA[I] = I;
25.     });
26.   });
27. }
28.
29.
30. int main() {
31.   std::array<float, 1024u> Vec;
32.   float Pi;
33.
34.
35. #pragma omp parallel sections
36.   {
37. #pragma omp section
38.     iota(Vec.data(), Vec.size());
39. #pragma omp section
40.     Pi = computePi(8192u);
41.   }
42.
43.
44.   std::cout << "Vec[512] = " << Vec[512] << std::endl;
45.   std::cout << "Pi = " << Pi << std::endl;
46.   return 0;
47. }
```

サンプルコードをコンパイルするには次のコマンドを使用します。

```
$ icpx -fsycl -fiopenmp -fopenmp-targets=spir64 offloadOmp_dpcpp.cpp
```

説明:

- `-fsycl` オプションは SYCL* を有効にします。
- `-fiopenmp -fopenmp-targets=spir64` オプションは OpenMP* オフロードを有効にします。

次はサンプルコードからの出力例となります。

```
$ ./a.out
Vec[512] = 512
Pi = 3.14159
```

注: コードに OpenMP* オフロードが含まれておらず、通常の OpenMP* コードのみが含まれる場合、-fopenmp-targets を省略した次のコマンドを使用します。

```
$ icpx -fsycl -fiopenmp omp_dpcpp.cpp
```

6.2.2 OpenCL* コードの相互運用性

oneAPI プログラミング・モデルでは、開発者は SYCL *API のさまざまな部分からすべての OpenCL* コードの機能を引き続き利用できます。SYCL* が提供する OpenCL* コードの相互運用性は、SYCL* が持つ高度なプログラミング・モデル・インターフェイスを利用しながら、従来の OpenCL* コードを再利用するのに役立ちます。この相互運用性モードには 2 つに主要機能があります。

1. OpenCL* コードのオブジェクトから SYCL* オブジェクトを生成する。例えば、SYCL* バッファーは、OpenCL* `cl_mem` から、または `cl_command_queue` の SYCL* キューから構築できます。
2. SYCL* オブジェクトから OpenCL* コードのオブジェクトを取得する。例えば、SYCL* アクセサーに関連付けられた暗黙の `cl_mem` を使用する OpenCL* カーネルを起動します。

6.3 DPC++ と OpenMP* オフロード処理のデバッグ

ホスト・プラットフォーム向けにコードを記述、デバッグ、および最適化する場合、コードを改善する手順はシンプルです。デバッガーで実行中のビルド、キャッチ、およびクラッシュや不正な結果の原因となる言語レベルのエラーに対処し、プロファイル・ツールを使用してパフォーマンスの問題を特定して修正します。

コードの一部分が DPC++ または OpenMP* オフロードにより別のデバイスで実行されるアプリケーションでは、コードの改善はかなり複雑になる可能性があります。

- DPC++ または OpenMP* オフロードの誤った使い方は、接続されたオフロードデバイスでプログラムが実行される際に、ジャストインタイム (JIT) コンパイルされるまで認識できないことがあります (この問題は、事前コンパイル (AOT) で確認できることがあります)。

- プログラムの論理的なエラーによるクラッシュは、ホスト、オフロードデバイス、または各種計算デバイスを連携されるソフトウェア・スタックで予期しない動作として現れる可能性があります。原因を究明するには以下が必要です。
  - インテル® ディストリビューションの GDB などの標準デバッガーを使用して、ホスト上のコードで何が起こっているかデバッグします。
  - デバイス固有のデバッガーを使用してオフロードデバイスの問題をデバッグします。ただし、デバイスのアーキテクチャー、計算スレッドを表現する規則、またはアセンブリーがホストとは異なることに注意してください。
  - カーネルとデータがデバイスとのやり取りを行う場合にのみ中間ソフトウェア・スタックで現れる問題をデバッグするには、デバイスとホスト間の通信、および処理中に報告されるエラーを監視する必要があります。
- ホストとオフロードデバイスで発生する可能性がある一般的なパフォーマンスの問題に加えて、ホストとオフロードデバイスが連携するパターンは、アプリケーションのパフォーマンスに大きな影響を与えることがあります。これは、ホストとオフロードデバイス間の通信を監視する必要があるもう 1 つのケースです。

ここでは、オフロードプログラムの実行中に利用できる各種デバッグ、およびパフォーマンスに解析ツールとその用法について説明します。

リソースをオフロードする拡張機能を備えた OpenMP* または SYCL* API を使用するアプリケーションのトラブルシューティングは、「高度な並列アプリケーションのトラブルシューティング」(英語) のチュートリアルを参照してください。

6.3.1 SYCL* と OpenMP* 開発向けの oneAPI デバッグツール

次に示すツールは、SYCL* および OpenMP* オフロード処理のデバッグに役立ちます。

SYCL* および OpenMP* オフロード処理をデバッグするツール

ツール	使用方法
環境変数	環境変数を使用して、プログラムを変更することなく、プログラムの実行時に OpenMP* および SYCL* ランタイムから診断情報を収集できます。
GPU 向けのプロファイル・ツールである ze_tracer ツール (GPU 向け PTI)	SYCL* よび OpenMP* オフロードで oneAPI レベルゼロと OpenCL* バックエンドを使用する場合、このツールを利用してバックエンドのエラーをデバッグし、ホストとデバイスの両方でパフォーマンスのプロファイルを実行できます。 **関連資料:** • Onetrace ツールの GitHub* (英語) • GPU 向け PTI の GitHub* (英語) • 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」
OpenCL* アプリケーションのインターセプト・レイヤー	SYC*L および OpenMP* オフロードで OpenCL* バックエンドを使用する場合、このライブラリーを利用してバックエンドのエラーをデバッグし、ホストとデバイスの両方でパフォーマンスのプロファイルを実行できます。
インテル® ディストリビューションの GDB	通常ホストおよびデバイス (CPU、GPU、FPGA エミュレーション) で、論理的なバグを調査する際にアプリケーションのソースレベルのデバッグで使用されます。

ツール	使用方法
インテル® Inspector	このツールは、オフロードの失敗につながる問題を含め、メモリーとスレッドの問題を検出してデバッグするのに役立ちます。 **注:** インテル® Inspector は、インテル® oneAPI HPC ツールキットに含まれます。
アプリケーション・デバッグ	これらのツールとランタイムベースのアプローチに加えて、開発者は別のアプローチから問題を見つけることもできます。次に例を示します。 • カーネルの出力と期待される出力を比較 • デバッグ用途の変数を作成して中間結果を確認 • カーネル内で結果をプリント **注:** SYCL* と OpenMP* では、どちらもオフロード領域内からの `stdout` への出力をサポートします。どの SIMD レーンまたはスレッドが出力しているか確認してください。
SYCL* 例外ハンドラー	一部の DPC++ プログラミングのエラーは、プログラムの実行中に SYCL* ランタイムから例外として返されます。これらは、実行時にフラグを使用するコード内のエラーを診断するのに役立ちます。詳細とサンプルについては、「SYCL* 例外」を参照してください。SYCL* 例外のサンプルについては、以下を参照してください。 • ガイド付き行列乗算の例外 • ガイド付き行列乗算の無効なコンテキスト • ガイド付き行列乗算の競合状態
インテル® Advisor	Fortran、C、C++、OpenCL* および SYCL* アプリケーションが最新のプロセッサーで最大限のパフォーマンスを発揮するのを支援します。
インテル® VTune™ プロファイラー	ネイティブシステムまたはリモートシステムでパフォーマンス・データを収取して解析します。
OpenMP* オフロード・ディレクティブ	「インテルツールによる OpenMP* アプリケーションのオフロードと最適化」(英語) では、OpenMP* ディレクティブを使用してアプリケーションに並列処理を追加する方法を説明しています。

6.3.1.1. デバッグ環境変数

OpenMP* と SYCL* のオフロードランタイム、およびレベルゼロ、OpenCL*、シェーダーコンパイラーは、ホストとオフロードデバイス間の通信を監視する環境変数を提供します。この環境変数を使用して、オフロード計算向けに選択されたランタイムを検出または制御できます。

6.3.1.1.1. OpenMP* オフロード環境変数

OpenMP* オフロードがどのように動作するかを理解し、バックエンドの制御に利用できるいくつかの環境変数が用意されています。

注: OpenMP* は FPGA デバイスではサポートされません。

OpenMP* オフロード環境変数

環境変数	説明
`LIBOMPTARGET_DEBUG=<Num>`	デバッグ情報の出力を制御します。詳細は、「ランタイム」(英語) を参照してください。この環境設定は、OpenMP* オフロードランタイムからのデバッグ出力を有効にします。以下が報告されます。 • 検出および使用された利用可能なランタイム (1、2) • 選択されたランタイムが開始および停止されたタイミング (1、2) • 使用されるオフロードデバイスの詳細 (1、2) • ロードされたサポート・ライブラリー (1、2) • すべてのメモリー割り当てと割り当て解除のサイズとアドレス (1、2) • デバイス間とのデータコピーに関する情報、また統合共有メモリーではデバイスマッピング (1、2) • カーネルの起動と起動時の詳細情報 (引数、SIMD 幅、グループ情報など) (1、2) • 呼び出されたゼロレベル/OpenCL* API 関数 (関数名、引数/パラメーター) (2) 値: `<Num> = 0`: 無効 `<Num> = 1`: デバイス検出、カーネルコンパイル、メモリコピー操作、カーネル呼び出し、およびその他のプラグイン依存アクションなどのプラグインアクションからの基本的なデバッグ情報を表示します。 `<Num> = 2`: `<Num>=1` の出力に加えて、どの GPU ランタイム API 関数がどの引数/パラメーターで呼び出されたかを表示します。 **デフォルト:** 0

環境変数	説明
`LIBOMPTARGET_INFO=<Num>`	この環境変数は、オフロードランタイムからの基本的なオフロード情報の表示を制御します。以下を表示します。ユーザーが `libomptarget` にさまざまなランタイム情報を要求できるようにします。詳細は、「ランタイム」(英語) を参照してください。 • OpenMP* デバイスカーネルが受け取ったすべてのデータ引数を出力します (1) • マップされたアドレスがデバイス・マッピング・データ・テーブルにすでに存在することを示します (2) • ターゲットのオフロードが失敗した場合、デバイス・ポインター・マップの内容をダンプします (4) • デバイス・マッピング・テーブルでエントリーが変更されたことを示します (8) • データがデバイス間でコピーされたことを示します (32) 値: (0, 1, 2, 4, 8, 32) **デフォルト:** 0

`LIBOMPTARGET_PLUGIN_PROFILE=<Num>[,<Unit>]`	この環境変数は、オフロードされた OpenMP* コードのパフォーマンス・データの表示を有効にします。以下を表示します。
	• 合計データ転送時間 (リードとライト)
	• データ割り当て回数
	• モジュールのビルド時間 (ジャストインタイム・コンパイル)
	• 各カーネルの実行時間。
	値:
	• `F` - 無効化
	• `T` - ミリ秒単位のタイミングで有効化
	• `T,usec` - マイクロ秒単位のタイミングで有効化
	デフォルト: `F`
	例: `export LIBOMPTARGET_PLUGIN_PROFILE=T,usec`
	`<Enable>:= 1 | T`
	`<Unit> := usec | unit_usec`
	基本的なプラグイン・プロファイルを有効にし、プログラムの終了時に結果を出力します。`<Unit>` を指定しない場合、マイクロ秒がデフォルトの単位になります。

環境変数	説明

LIBOMPTARGET_PLUGIN=<Name>	この環境変数を使用して OpenMP* オフロードの実行に使用するバックエンドを選択できます。
	
	注: レベルゼロのバックエンドは GPU デバイスでのみサポートされます。
	
	`<Name>:=`
	`LEVEL0 | OPENCL | CUDA | X86_64 | NIOS2 |`
	`level0 | opencl | cuda | x86_64 | nios2 |`
	
	使用するオフロードのプラグイン名を指定します。このオプションが指定されると、オフロードランタイムは他の RTL をロードしません。
	
	値:
	
	• LEVEL0 または LEVEL_ZERO - レベルゼロ・バックエンドを使用します
	• OPENCL - OpenCL* バックエンドを使用します
	
	デフォルト:
	
	• GPU オフロードデバイス: LEVEL0
	• CPU または FPGA オフロードデバイス: OPENCL

LIBOMPTARGET_PROFILE=<FileName>	libomptarget 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。詳細は、「ランタイム」(英語) を参照してください。

環境変数	説明
LIBOMPTARGET_DEVICES=<DeviceKind>	<DeviceKind> := DEVICE | SUBDEVICE | SUBSUBDEVICE | ALL | device | subdevice | subsubdevice | all サブデバイスをユーザーに公開する方法を制御します。 DEVICE/device: 最上位レベルのデバイスのみが OpenMP* デバイスとしてレポートされ、subdevice 節がサポートされます。 SUBDEVICE/subdevice: 第 1 レベルのサブデバイスのみが OpenMP* デバイスとしてレポートされ、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* デバイスとしてレポートされ、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。 ALL/all: 最上位のデバイスとそれらのサブデバイスが OpenMP* デバイスとしてレポートされ、subdevice 節は無視されます。これは、インテル® GPU ではサポートされておらず、今後廃止される予定です。 **デフォルト:** <DeviceKind>=device に相当
LIBOMPTARGET_LEVEL0_MEMORY_POOL=<Option>	<Option> := 0 | <PoolInfoList> <PoolInfoList> := <PoolInfo>[,<PoolInfoList>] <PoolInfo> := <MemType>[,<AllocMax>[,<Capacity>[,<PoolSize>]]] <MemType> := all | device | host | shared <PoolSize> := 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) 再利用可能なメモリープールの構成を制御します。プールは、合計サイズが <PoolSize> を超えない、1 ブロックから最大 <AllocMax> サイズの <Capacity> 割り当てが可能なメモリーブロックのリストです。 **デフォルト:** <Option>=device,1,4,256,host,1,4,256,shared,8,4,256 に相当
LIBOMPTARGET_LEVEL0_STAGING_BUFFER_SIZE=<Num>	ステージ・バッファー・サイズを <Num> KB に設定します。ステージバッファーホストとデバイス間のコピー操作において、2 段階のコピー操作の一時ストレージとして使用されます。バッファーはディスクリート・デバイスでのみ使用されます。 **デフォルト:** 16

環境変数	説明
LIBOMPTARGET_LEVEL_ZERO_COMMAND_BATCH=<Value>	<Value> := <Type>[,<Count>] <Type> := none | NONE | copy | COPY | compute | COMPUTE <Count> := バッチ処理するコマンドの最大数 ターゲット領域のコマンドバッチ処理を有効にします。 <Type>=none|NONE: コマンドバッチ処理を無効にします。 <Type>=copy|COPY: データ転送でターゲット領域のコマンドバッチ処理を有効にします。 <Type>=compute|COMPUTE: データ転送と計算にターゲット領域のコマンドバッチ処理を有効にし、コピーエンジンの利用を無効にします。 <Type> が copy または compute (有効) のいずれかで、<Count> が指定されていないと、ターゲット領域ですべてのコマンドに対してバッチ処理が行われます。 **デフォルト:** <Type>=none (無効)
LIBOMPTARGET_LEVEL_ZERO_USE_IMMEDIATE_COMMAND_LIST=<Bool>	<Bool> := 1 | T | t | 0 | F | f 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。 **デフォルト:** 無効
OMP_TARGET_OFFLOAD=MANDATORY	これは OpenMP* 仕様で定義されています。 https:// www.openmp.org/spec-html/5.1/openmpse74.html#x340-515000617 (英語)

6.3.1.1.2. SYCL* と DPC++ 環境変数

DPC++ コンパイラーは、すべての標準 SYCL* 環境変数をサポートします。すべての環境変数については GitHub* (英語) をご覧ください。デバッグで注目すべきは、次の SYCL* 環境変数と追加のレベルゼロ環境変数です。

SYCL* と DPC++ 環境変数

環境変数	説明
ONEAPI_DEVICE_SELECTOR	この複雑な環境変数を使用すると、ランタイムで使用されるランタイム、計算デバイス ID を利用可能なすべての組み合わせのサブセットに制御できます。 計算デバイス ID は、SYCL* API、clinfo、または sycl-ls (0 から始まる番号) によって返される ID に対応します。その ID を持つデバイスが特定のタイプであったり、特定のランタイムをサポートするかには関連性がありません。プログラムが特定のセレクター gpu_selector など) を使用して、ONEAPI_DEVICE_SELECTOR のフィルターで除外されたデバイスを要求すると、例外がスローされます。 詳細については、GitHub* にある環境変数の説明をご覧ください。 https://github.com/intel/llvm/blob/sycl/sycl/doc/EnvironmentVariables.md (英語) 次のような値を含みます。 • opencl:cpu - 利用可能なすべての CPU デバイスで OpenCL* ランタイムのみを使用します • opencl:gpu - 利用可能なすべての GPU デバイスで OpenCL* ランタイムのみを使用します • opencl:gpu:2 - GPU である 3 番目のデバイスでのみ OpenCL* ランタイムのみを使用します • level_zero:gpu:1 - GPU である 2 番目のデバイスでのみレベルゼロランタイムのみを使用します • opencl:cpu,level_zero - CPU デバイスでは OpenCL* ランタイムのみを使用し、サポートされる計算デバイスではレベルゼロランタイムを使用します **デフォルト:** 利用可能なすべてのランタイムとデバイスを使用します
ONEAPI_DEVICE_SELECTOR	このデバイス選択環境変数によって、SYCL* ベースのアプリケーションの実行時に使用するデバイスの選択を制御できます。デバイスを特定のタイプ (GPU やアクセラレーター) またはバックエンド (レベルゼロや OpenCL*) に制限するのに役立ちます。このデバイス選択のメカニズムは、ONEAPI_DEVICE_SELECTOR を置き換えるものです。ONEAPI_DEVICE_SELECTOR の構文は OpenMP* と共有されており、デバイスを可能にします。詳しい説明は、インテル® oneAPI DPC++ コンパイラーのドキュメント (英語) を参照してください。
SYCL_PI_TRACE	この環境設定は、ランタイムからのデバッグ出力を有効にします。 値: • 1 - SYCL* プラグインとデバイスが検出され使用されたことを報告します • 2 - 引数と結果の値を含む SYCL* API 呼び出しを報告します • -1 - 利用可能なすべてのトレースを報告します **デフォルト:** 無効

環境変数	説明
ZE_DEBUG	この環境変数は、ランタイムが使用された際にレベルゼロ・バックエンドからのデバッグ出力を有効にします。以下が報告されます。 • レベルゼロ API の呼び出し • レベルゼロイベント情報 値: 任意の値で定義された変数 (有効) **デフォルト:** 無効

6.3.1.1.3. サポート向けの診断情報を生成する環境変数

レベルゼロ・バックエンドは、動作を制御して診断を支援するいくつかの環境変数を提供します。

- レベルゼロ仕様、コア・プログラミング・ガイド: https://spec.oneapi.com/level-zero/latest/core/PROG.html#environment-variables (英語)
- レベルゼロ仕様、ツール・プログラミング・ガイド: https://spec.oneapi.com/level-zero/latest/tools/PROG.html#environment-variables (英語)

デバッグ情報の追加リソースは、実行時または事前コンパイル(AOT) 時に、レベルゼロまたは OpenCL* バックエンド (OpenMP* オフロードと SYCL*/DPC++ ランタイムで使用される) によって呼び出されるインテル® グラフィックス・コンパイラーから提供されます。インテル® グラフィックス・コンパイラーは、ターゲットのオフロードデバイス向けの実行形式コードを生成します。環境変数の完全なリストは、https://github.com/intel/intel-graphics-compiler/blob/master/documentation/configuration_flags.md (英語) を参照してください。パフォーマンスの問題をデバッグする際に必要となるのは、次の 2 つです。

- `IGC_ShaderDumpEnable=1` (デフォルトは `0`) によって、インテル® グラフィックス・コンパイラーが生成するすべての LLVM、アセンブリー、および ISA コードが `/tmp/IntelIGC/<application_name>` に書き込まれます。
- `IGC_DumpToCurrentDir=1` (デフォルトは `0`) は、`IGC_ShaderDumpEnable` によって生成されるすべてのファイルを `/tmp/IntelIGC/<application_name>` に書き込みます。多数のファイルが生成される可能性があるため、一時ディレクトリーを作成することを推奨します。

インテル® oneAPI の異なるバージョン間で OpenMP* オフロードや SYCL* オフロード・アプリケーションのパフォーマンスの問題が生じる場合、異なるコンパイラー・オプションを使用したり、デバッガーを使用するなど、`IGC_ShaderDumpEnable` を有効にして結果ファイルを提供する必要があります。互換性の詳細については、「oneAPI ライブラリーの互換性」を参照してください。

6.3.1.2. オフロード・インターセプト・ツール

オフロード・ソフトウェア自身に組み込まれているデバッガーと診断情報に加えて、オフロード・パイプラインを介して送信される API 呼び出しとデータを監視するのは有益です。レベルゼロでは、アプリケーションが onetrace または ze_tracer ツールの引数として実行される場合、アプリケーションのレベルゼロのさまざまな動作がインターセプトされ

報告されます。OpenCL* では、OpenCL* 呼び出しをインターセプトして報告するライブラリーを `LD_LIBRARY_PATH` に追加して、環境変数を設定しファイルへの診断情報生成を制御できます。また、onetrace または cl_tracer を使用して、アプリケーションの OpenCL* API 呼び出しのさまざまな情報をレポートすることもできます。ここでも、アプリケーションは onetrace や cl_tracer ツールへの引数として実行されます。

6.3.1.2.1. OpenCL* アプリケーションのインターセプト・レイヤー

このライブラリーは、SYCL* または OpenMP* オフロードのバックエンドとして OpenCL* が使用される場合にデバッグおよびパフォーマンス・データを収集します。OpenCL* が SYCL* または OpenMP* オフロードのバックエンドである場合、このツールはバッファーのオーバーライト、メモリーリーク、ポインターの不一致を検出し、ランタイム・エラー・メッセージのより詳しい情報を提供します (CPU、FPGA、または GPU のいずれかが計算デバイスである場合にこれらの問題を診断できます)。OpenCL* バックエンドを使用するプログラムで onetrace を利用する場合や、レベルゼロ・バックエンドを使用するプログラムで OpenCL* アプリケーション・ライブラリーのインターセプト・レイヤーを利用する場合には、あまり役立たないことに注意してください。

追加のリソース:

- OpenCL* アプリケーションのインターセプト・レイヤーのビルドと使用法に関する各種情報は、https://github.com/intel/opencl-intercept-layer (英語) から入手できます。

  注: 最良の結果を得るには、次のフラグを使用して cmake を実行します。

  ```
  -DENABLE_CLIPROF=TRUE -DENABLE_CLILOADER=TRUE
  ```

- 同様のツール (CLIntercept) に関する情報は、https://github.com/gmeeker/clintercept (英語) および https://sourceforge.net/p/clintercept/wiki/Home/ (英語) で入手できます。
- OpenCL* アプリケーションのインターセプト・レイヤーの制御に関連する情報は、https:// github.com/intel/opencl-intercept-layer/blob/master/docs/controls.md (英語) にあります。
- GPU の最適化に関する情報は、『oneAPI GPU 最適化ガイド』(英語 | 日本語)から入手できます。

6.3.1.2.2. GPU 向けのプロファイル・ツールのインターフェイス (onetrace、cl_tracer および ze_trace)

OpenCL* アプリケーションのインターセプト・レイヤーと同様に、これらツールは、レベルゼロが SYCL* または OpenMP* バックエンドである場合にデバッグおよびパフォーマンス・データを収集します。レベルゼロは、GPU で実行される計算のバックエンドとしてのみ利用できることに注意してください (現時点で、CPU と FPGA 向けのレベルゼロ・バックエンドはありません)。onetrace ツールは、https://github.com/intel/pti-gpu (英語) にある GPU 向けのプロファイル・ツール・インターフェイス (GPU 向け PTI) プロジェクトの一部です。このプロジェクトには、レベルゼロまたは OpenCL* オフロード・バックエンドからのアクティビティーのみをトレースする ze_tracer や cl_tracer ツールも含まれています。ze_tracer および cl_tracer ツールは、ほかのバックエンドを使用するアプリケーションで使用されると、出力を生成しませんが、onetrace は使用するオフロード・バックエンドにかかわりなく出力できます。

onetrace ツールはソースで配布されています。ツールのビルドに関する手順は、https://github.com/intel/pti-gpu/tree/master/tools/onetrace (英語) を参照してください。このツールは次の機能を提供します。

- 呼び出しのログ: このモードでは、すべての標準レベルゼロ (L0) API 呼び出しとその引数、およびタイムスタンプ付きの戻り値をトレースできます。これにより、ホストプログラムが接続された計算デバイスを利用する際に発生する障害の補足情報が得られます。
- ホストとデバイスのタイミング: すべての API 呼び出しの存続期間、カーネルの存続期間、およびアプリケーション全体の実行時間を提供します。
- デバイス・タイムライン・モード: 各デバイス・アクティビティーのタイムスタンプを提供します。すべてのタイムスタンプは、同じ (CPU) 時間スケールで示されます。
- Chrome* 呼び出しのログモード: chrome://tracing ブラウザーツール (英語) で開くことができる JSON 形式への API 呼び出しをダンプします。

このデータはオフロードの障害やパフォーマンスの問題をデバッグするのに役立ちます。

追加のリソース:

- GPU 向けプロファイル・ツール・インターフェイス (GPU 向け PTI) GitHub* プロジェクト (英語)
- onetrace ツールの GitHub* (英語)

6.3.1.3. インテル® ディストリビューションの GDB

インテル® ディストリビューションの GDB は、プログラムの状態を調査および変更できるアプリケーション・デバッガーです。アプリケーションのホスト部分とデバイスにオフロードされたカーネルの両方を、同じデバッグセッションでシームレスにデバッグできます。このデバッガーは、CPU、GPU、および FPGA エミュレーション・デバイスをサポートします。主な機能を以下に示します。

- GPU デバイスに自動的に接続してデバッグイベントを監視
- デバッグ用に JIT コンパイルされた、または動的にロードされたカーネルバイナリーを自動検出
- プログラムの実行を停止するブレークポイントを設定 (カーネル内部と外部の両方)
- スレッドのリスト表示、現在のスレッド・コンテキストへの切り替え
- アクティブな SIMD レーンのリスト表示、現在の SIMD レーンのコンテキストをスレッドごとに切り替え
- 複数のスレッドと SIMD レーン・コンテキストの式値の評価と出力
- レジスター値の調査と変更
- マシン命令を逆アセンブル
- 関数呼び出しスタックの表示とナビゲート
- ソースと命令レベルのステップ実行
- 非停止およびすべて停止のデバッグモード
- インテル® プロセッサー・トレースを使用した実行の記録 (CPU のみ)

インテル® ディストリビューションの GDB の詳細とドキュメントへのリンクは、『インテル® ディストリビューションの GDB 導入ガイド (Linux* 版)』(英語) または『インテル® ディストリビューションの GDB 導入ガイド(Windows* 版)』(英語) をご覧ください。

6.3.1.4. オフロード向けインテル® Inspector

インテル® Inspector は、シリアルおよびマルチスレッド・アプリケーションを開発するユーザー向けの動的メモリーおよびスレッドエラーを検出するツールです。動的に生成されたオフロードコードだけでなく、アプリケーションのネイティブコードの正当性を検証するのに利用できます。

前述のツールや手法とは異なり、インテル® Inspector は GPU または FPGA と通信するオフロードコードのエラーを検出することはできません。インテル® Inspector では、CPU をターゲットとしてカーネルを実行するように SYCL* または OpenMP* ランタイムを設定する必要があります。解析を実行する前に、次の環境変数を設定する必要があります。

- CPU デバイスでカーネルを実行するように SYCL* アプリケーションを設定します。

```
$ export ONEAPI_DEVICE_SELECTOR=opencl:cpu
```

- CPU デバイスでカーネルを実行するように OpenMP* アプリケーションを設定します。

```
$ export OMP_TARGET_OFFLOAD=MANDATORY
$ export LIBOMPTARGET_DEVICETYPE=cpu
```

- JIT コンパイラーまたはランタイムでコード解析とトレース有効にします。

```
$ export CL_CONFIG_USE_VTUNE=True
$ export CL_CONFIG_USE_VECTORIZER=false
```

次のいずれかのコマンドを使用して、コマンドラインから解析を開始します。インテル® Inspector のグラフィカル・ユーザー・インターフェイスからも開始できます。

- メモリー: `inspxe-cl -c mi3 -- <app> [app_args]`
- スレッド化: `inspxe-cl -c ti3 -- <app> [app_args]`

次のコマンドを使用して解析結果を表示します。

```
$ inspxe-cl -report=problems -report-all
```

SYCL* や OpenMP* オフロードプログラムが OpenCL* バックエンドに不正なポインターを渡したり、誤ったスレッドからバックエンドに不正なポインターを渡す場合、インテル® Inspector は問題を警告します。これは、インターセプト・レイヤーやデバッガーを使用して問題を見つけるよりも容易な場合があります。

『インテル® Inspector ユーザーガイド (Linux* 版)』(英語) または『インテル® Inspector ユーザーガイド (Windows* 版)』(英語) で詳細を参照できます。

6.3.2 オフロード処理のトレース

GPU に計算をオフロードするプログラムが開始される場合、プログラムの実行に関連する多くのコンポーネントが動作します。マシン非依存のコードをマシン依存のコードにコンパイルし、データとバイナリーをデバイスにコピーして、結果を戻す必要があります。ここでは、「oneAPI デバッグツール」で説明したツールを使用して、すべてのアクティビティーをトレースする方法を示します。

6.3.2.1. カーネル設定時間

オフロードコードをデバイスで実行する前に、マシン非依存のカーネルをターゲットデバイス向けにコンパイルしてコードをデバイスにコピーする必要があります。このカーネルのセットアップ時間が考慮されていないと、ベンチマークを複雑にしたり歪めたりする可能性があります。ジャストインタイム・コンパイルは、オフロード・アプリケーションのデバッグに遅延を引き起こすことがあります。

OpenMP* オフロードプログラムでは、環境変数 `LIBOMPTARGET_PLUGIN_PROFILE=T[,usec]` を設定すると、オフロードコード `ModuleBuild` のビルドに必要な時間が報告されます。これをプログラムの実行時間全体と比較します。

SYCL* オフロードプログラムでは、カーネルのセットアップ時間を判断するのはさらに困難になります。

- レベルゼロまたは OpenCL* がバックエンドである場合、onetrace や ze_tracer よって返されるデバイスのタイミングとタイムラインから、カーネルのセットアップ時間を求められます。
- OpenCL* がバックエンドの場合、OpenCL* アプリケーションのインターセプト・レイヤーを使用して情報を取得する際に、`BuildLogging`、`KernelInfoLogging`、`CallLogging`、`CallLoggingElapsedTime`、`KernelInfoLogging`、`HostPerformanceTiming`、`HostPerformanceTimeLogging`、`ChromeCallLogging`、または `CallLoggingElapsedTime` フラグを設定できます。onetrace や cl_tracer よって返されるデバイスのタイミングおよびタイムラインから、カーネルのセットアップ時間を求めることもできます。

この方法により、`LIBOMPTARGET_PLUGIN_PROFILE=T` によって戻される情報を補足できます。

インテル® VTune™ プロファイラーがカーネルのセットアップ時間を解析する方法の詳細については、「Linux* カーネル解析を有効にする」を参照してください。

6.3.2.2. バッファーの作成、サイズ、およびコピーの監視

バッファーが作成された時期、作成されたバッファーの数、およびそれらが再利用されるか、また常に作成および破棄されるかを知ることは、オフロード・アプリケーションのパフォーマンスを最適化する上で重要な鍵となります。しかし、OpenMP* や SYCL* などの高レベル・プログラミング言語を使用する場合、それらは必ずしも明確でない可能性がありユーザーからバッファーの管理は隠匿されます。

高レベルでは、プログラムの実行時に `LIBOMPTARGET_DEBUG` および `SYCL_PI_TRACE` 環境変数を使用して、バッファーに関連するアクティビティーを追跡できます。`LIBOMPTARGET_DEBUG` は、`SYCL_PI_TRACE` よりも多くの情報

を提供します (作成されたバッファーアドレスとサイズをレポート)。SYCL_PI_TRACE は、API 呼び出しをレポートするだけであり、バッファーアドレスやサイズに関する情報は含まれません。

低レベルでは、レベルゼロのバックエンドを使用する場合、onetrace や ze_tracer の呼び出しログモードは、引数を含むすべてのレベルゼロ API 呼び出しに関する情報を提供します。例えば、バッファー作成呼び出し (zeMemAllocDevice など) で、デバイスとの間で渡される結果バッファーのサイズを取得できるため役立つことがあります。onetrace と ze_tracer は、デバイス・タイムライン・モードですべてのゼロレベルデバイス間とのアクティビティー (メモリー転送を含む) をダンプすることができます。アクティビティーごとに、追加 (コマンドリストへ)、送信 (キューへ)、開始と終了時間を取得できます。

OpenCL* がバックエンドの場合、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際に、CallLogging、CallLoggingElapsedTime、および ChromeCallLogging フラグを設定して同様の情報を取得できます。onetrace や ze_tracer の呼び出しログモードは、引数を含むすべての OpenCL* API 呼び出しに関する情報を提供します。上記と同様に、onetrace と ze_tracer を使用すると、デバイス・タイムライン・モードですべての OpenCL* デバイス間とのアクティビティー (メモリー転送を含む) をダンプすることができます。

6.3.2.3. 合計転送時間

合計データ転送時間をカーネル実行時間と比較することは、接続されているデバイスへの計算のオフロードが有益であるか判断することが重要になることがあります。

OpenMP* オフロードプログラムでは、LIBOMPTARGET_PLUGIN_PROFILE=T[,usec] を設定すると、ビルド (DataAlloc)、オフロードデバイスの読み取り (DataRead)、および書き込み (DataWrite) に必要な時間がレポートされます (ただし合計として)。

SYCL* を使用する C++ プログラムのデータ転送時間を判断するのはさらに困難になります。

- レベルゼロまたは OpenCL* がバックエンドである場合、onetrace や ze_tracer よって返されるデバイスのタイミングとタイムラインから、合計データ転送時間を求められます。
- OpenCL* がバックエンドの場合、onetrace または cl_tracer を使用するか、OpenCL* アプリケーションのインターセプト・レイヤーを使用して情報を取得する際に、BuildLogging、KernelInfoLogging、CallLogging、CallLoggingElapsedTime、KernelInfoLogging、HostPerformanceTiming、HostPerformanceTimeLogging、ChromeCallLogging、または CallLoggingElapsedTime フラグを設定できます。

インテル® VTune™ プロファイラーがカーネルのセットアップ時間を解析する方法の詳細については、『インテル® VTune™ プロファイラー・ユーザーズガイド』の「GPU オフロード解析」、「ホットスポット・レポート」、「GPU 計算/メディア・ホットスポット・ビュー」を参照してください。

6.3.2.4. カーネル実行時間

OpenMP* オフロードプログラムでは、LIBOMPTARGET_PLUGIN_PROFILE=T[,usec] を設定すると、オフロードされたすべてのカーネル (Kernel#...) の合計実行時間が報告されます。

SYCL* を使用するオフロードカーネル向け:

- レベルゼロ または OpenCL* バックエンドでは、onetrace や ze_tracer のデバイス・タイミング・モードを使用してすべてのカーネルのデバイスでの実行時間を取得できます。

- OpenCL* がバックエンドの場合、onetrace または cl_tracer を使用するか、OpenCL* アプリケーションのインターセプト・レイヤーを使用して情報を取得する際に、次のフラグを設定できます: `CallLoggingElapsedTime`、`DevicePerformanceTiming`、`DevicePerformanceTimeKernelInfoTracking`、`DevicePerformanceTimeLWSTracking`、`DevicePerformanceTimeGWSTracking`、`ChromePerformanceTiming`、`ChromePerformanceTimingInStages`。

インテル® VTune™ プロファイラーがカーネルの実行時間を解析する方法の詳細については、「アクセラレーター解析グループ」を参照してください。

6.3.2.5. デバイスカーネルが呼び出され、スレッドが生成される場合

場合によってには、オフロードカーネルが作成され、実行を開始するかなり前にデバイスへ転送されることがあります (通常は、カーネルの実行に必要なすべてのデータと制御が転送された後にのみ)。

インテル® ディストリビューションの GDB を使用してデバイスカーネルにブレークポイントを設定できます。そして、カーネル引数の照会、スレッドの生成と破棄の監視、コード内の現在のスレッドの位置とリスト表示 (`info thread` を使用) などが行えます。

6.3.3 オフロード処理のデバッグ

6.3.3.1. 異なるランタイムまたは計算デバイスで実行

オフロードプログラムが正常に実行されなかったか、生成された結果が正しくない場合、比較的容易な正当性の確認方法は、OpenMP* アプリケーションでは `LIBOMPTARGET_PLUGIN` と `OMP_TARGET_OFFLOAD` 環境変数を、また SYCL* アプリケーションでは `ONEAPI_DEVICE_SELECTOR` 環境変数を使用して、別のランタイム (OpenCL* とレベルゼロ) または計算デバイス (CPU と GPU) でアプリケーションを実行することです。異なるランタイム間で再現されるエラーは、ほとんどの場合、ランタイムの問題として排除できます。そしてデバイス間で再現されるエラーの大部分は、不良ハードウェアの問題を排除できます。

6.3.3.2. CPU 実行をデバッグ

オフロードコードを CPU で実行するには、ホスト実装と OpenCL* の CPU バージョンという 2 つのオプションがあります。ホスト実装は、オフロードコードのネイティブ実装であり、オフロードされないコードと同じようにデバッグできます。OpenCL* の CPU バージョンは、OpenCL* ランタイムとコード生成プロセスを通過しますが、最終的にインテル® oneAPI スレッディング・ビルディング・ブロック (oneTBB) ランタイムで実行される通常の並列コードになります。繰り返しますが、これにより慣れ親しんだアセンブリーと並列処理メカニズムのデバッグ環境が提供されています。ポイン

ターはスタック全体にアクセスでき、データを直接参照できます。また、オペレーティング・システム・プロセスの通常の上限を超えるメモリー制限はありません。

CPU オフロード実行のエラーを検出して修正すると、GPU オフロード実行で発生するエラーよりもはるかに少ない労力でエラーを解決でき、GPU やほかのアクセラレーターが接続されたシステムを利用する必要もなくなります。

OpenMP* アプリケーションでホスト実装を適用するには、target または device 構造を削除して、通常のホスト OpenMP* コードに置き換えます。`LIBOMPTARGET_PLUGIN=OPENCL` が設定され、GPU オフロードが無効になると、オフロードコードは OpenMP* ランタイムで実行され、インテル® oneAPI スレッディング・ビルディング・ブロック (oneTBB) が並列処理を行います。

SYCL* アプリケーションで `ONEAPI_DEVICE_SELECTOR=host` を設定するとホストデバイスはシングルスレッドで実行を行います。これは、データ競合やデッドロックなどスレッド化の問題が実行エラーの原因であるか判断するのに役立ちます。`ONEAPI_DEVICE_SELECTOR=opencl:cpu` に設定すると、CPU の OpenCL* ランタイムが使用され、インテル® oneAPI スレッディング・ビルディング・ブロック (oneTBB) が並列処理を行います。

6.3.3.3. インテル® ディストリビューションの GDB を使用して GPU 実行をデバッグ

インテル® ディストリビューションの GDB については、『インテル® ディストリビューションの GDB 導入ガイド (Linux* 版)』(英語) または『インテル® ディストリビューションの GDB 導入ガイド (Windows* 版)』(英語) に詳しく記載されています。有用なコマンドについては、インテル® ディストリビューションの GDB の「リファレンス・シート」(英語) で簡単に説明されています。ただし、GDB を使用して GPU アプリケーションをデバッグする方法は、ホストでの手順とは若干異なるため (一部のコマンドの使い方が異なり、見慣れない出力が表示されることがあります)、それらの違いの一部をここで紹介します。

「インテル® ディストリビューションの GDB を使用したデバッグのチュートリアル (Linux* 版)」(英語) では、SYCL* プログラムのデバッグセッションを開始し、カーネル内にブレークポイントを設定し、プログラムを実行して GPU にオフロードしローカル値を出力し、スレッドの SIMD レーン 5 を切り替えて変数を再度出力するサンプルのデバッグセッションを紹介しています。

通常の GDB と同様に、`command <CMD>` には GDB の `help <CMD>` コマンドを使用して、`<CMD>` の情報テキストを読み取ります。次に例を示します。

```
(gdb) help info threads
Display currently known threads.Usage: info threads [OPTION]...[ID]...
If ID is given, it is a space-separated list of IDs of threads to display.Otherwise, all
threads are displayed.


Options:
-gid
Show global thread IDs.
```

6.3.3.3.1. GDB でインフェリアー、スレッド、および SIMD レーンの参照

アプリケーションのスレッドは、デバッガーによって一覧表示できます。表示される情報には、スレッド ID とスレッドが停止している位置が含まれます。GPU スレッドの場合、デバッガーはアクティブな SIMD レーンも表示します。

上記の例では、GDB の `info threads` コマンドでスレッドを表示していますが、見慣れない形式で情報が示されることがあります。

```
Id                  Target Id                   Frame
1.1                 Thread <id omitted> <frame omitted>
1.2                 Thread <id omitted> <frame omitted>

* 2.1:1             Thread 1073741824   <frame> at array-transform.cpp:61
2.1:[3 5 7]         Thread 1073741824   <frame> at array-transform.cpp:61
2.2:[1 3 5 7]       Thread 1073741888   <frame> at array-transform.cpp:61
2.3:[1 3 5 7]       Thread 1073742080   <frame> at array-transform.cpp:61
```

GDB は次の形式でスレッドを表示します: <インフェリアー番号>.<スレッド番号>:<SIMD レーン/s>

例えば、スレッド ID `2.3:[1 3 5 7]` は、インフェリアー 2 で実行されるスレッド 3 の SIMD レーン 1、3、5 および 7 を意味します。

GDB 用語の「inferior (インフェリアー)」は、デバッグされるプロセスを指します。GPU にオフロードするプログラムのデバッグセッションには、通常 2 つのインフェリアーがあります。プログラムのホストを示す「ネイティブ・インフェリアー」(上記のインフェリアー 1) と、GPU デバイスを示す「リモート・インフェリアー」(上記のインフェリアー 2) です。インテル® ディストリビューションの GDB は自動的に GPU インフェリアーを生成するため、特に操作は必要ありません。

式の値を出力すると、式は現在のスレッドの現在の SIMD レーンのコンテキストで評価されます。`thread 3:4`、`thread :6`、または `thread 7` などの `thread` コマンドを使用して、スレッドと SIMD レーンを切り替えることができます。最初のコマンドは、スレッド 3 と SIMD レーン 4 に切り替えます。2 番目のコマンドは、現在のスレッドで SIMD レーン 6 に切り替えます。3 番目のコマンドは、スレッド 7 に切り替えます。選択されるデフォルトレーンは、以前に選択したレーン (アクティブであれば)、またはスレッド内で最初にアクティブになったレーンのどちらかになります。

`thread apply` コマンドは、同様に広域または集中的である可能性があります (これにより、変数を調査するコマンドからの出力を制限しやすくなります)。SIMD レーンのデバッグの詳細と例については、「インテル® ディストリビューションの GDB を使用したデバッグのチュートリアル (Linux* 版)」(英語) を参照してください。

GDB のスレッドとインフェリアーに関する詳細は以下をご覧ください。

- https://sourceware.org/gdb/current/onlinedocs/gdb/Threads.html (英語)
- https://sourceware.org/gdb/current/onlinedocs/gdb/Inferiors-Connections-and-Programs.html#Inferiors-Connections-and-Programs (英語)

6.3.3.3.2. スケジューラーの制御

デフォルトでは、スレッドがブレークポイントに到達すると、デバッガーはブレーク・ポイント・ヒット・イベントをユーザーに通知する前にすべてのスレッドを停止します。これは GDB のすべて停止モードです。非停止モードでは、ほかのスレッドが実行される間、スレッドの停止イベントが表示されます。

すべて停止モードでは、スレッドが再開されると (例: `continue` コマンドで通常のように再開する、または `next` コマンドでステップ実行する場合)、ほかのすべてのスレッドも再開されます。スレッド化されたアプリケーションで複数のブ

レークポイントが設定されていると、ブレークポイントに到達した次のスレッドが続くスレッドではない可能性があるため、混乱を招く可能性があります。

`set scheduler-locking` コマンドを使用することで、現在のスレッドが再開されたときにほかのスレッドが再開されないように制御することができます。これは、現在のスレッドのみが命令を実行しているときに、ほかのスレッドの介入を避けるのに有効です。`help set scheduler-locking` と入力すると利用可能なオプションが表示されます。詳細は、https://sourceware.org/gdb/current/onlinedocs/gdb/Thread-Stops.html (英語) をご覧ください。SIMD レーンは個別に再開できないことに注意してください。これは、ベースとなるスレッドとともに再開されます。

非停止モードのデフォルトでは、現在のスレッドのみが再開されます。すべてのスレッドを再開するには、`continue` コマンドで `-a` フラグを指定します。

6.3.3.3.3. 1 つ以上のスレッド/レーンの情報をダンプ (Thread Apply)

プログラム状態を調査するコマンドは、通常、現在のスレッドの現在の SIMD レーンのコンテキストに適用されます。複数のコンテキストの値を調査することが必要なこともあります。そのような場合、`thread apply` コマンドを使用します。例えば、以下はスレッド 2,5 の SIMD レーン 3-5 に対して `print element` コマンドを実行します。

```
(gdb) thread apply 2.5:3-5 print element
```

同様に、以下は、現在のスレッドの SIMD レーン 3、5、および 6 のコンテキストに対し print element コマンドを実行します。

```
(gdb) thread apply :3 :5 :6 print element
```

6.3.3.3.4. ブレークポイント停止後の GPU コードのステップ実行

GPU にオフロードされたカーネル内で停止するには、カーネル内のソース行にブレークポイントを設定するだけです。GPU スレッドがそのソース行に到達すると、デバッガーは実行を停止してブレークポイントの到達を示します。ソース行単位でスレッドをステップ実行するには、`step` または `next` コマンドを使用します。`step` コマンドは関数にステップインし、`next` コマンドは関数呼び出しをステップオーバーします。ステップ実行する前に、ほかのスレッドの介入を避けるため `set scheduler-locking step` を設定することを推奨します。

6.3.3.3.5. インテル® ディストリビューションの GDB で使用する DPC++ 実行形式をビルド

ホスト・アプリケーションのデバッグと同様に、GPU でデバッグ可能なバイナリーを作成するには、いくつかの追加フラグを指定する必要があります。詳細については、『インテル® ディストリビューションの GDB 導入ガイド (Linux* 版)』(英語) をご覧ください。

JIT コンパイルを行う際にスムーズなデバッグを可能にするには、`-g` フラグを指定してコンパイラーのデバッグ情報生成を有効にし、アプリケーションのホストと JIT コンパイルカーネルの両方で `-O0` フラグを指定して最適化を無効にします。カーネルのフラグはリンク時に適用されます。次に例を示します。

- プログラムのコンパイル: `icpx -fsycl -g -O0 -c myprogram.cpp`
- プログラムのリンク: `icpx -fsycl -g -O0 myprogram.o`

Cmake を使用してプログラムをビルドする場合、CMAKE_BUILD_TYPE の Debug タイプを使用し、CMAKE_CXX_FLAGS_DEBUG 変数に -O0 を追加します。次に例を示します。

```
$ set (CMAKE_CXX_FLAGS_DEBUG "${CMAKE_CXX_FLAGS_DEBUG} -O0")
```

デバッグ向けにビルドされたアプリケーションは、通常の「リリース」ビルドで最適化されたアプリケーションよりも起動に時間がかかる場合があります。そのため、デバッガーで起動すると、プログラムの実行速度が遅くなったように感じられることがあります。これにより問題が生じる場合、大規模なアプリケーションの開発者は、プログラムの実行時ではなくビルド時に JIT オフロードコードを事前 (AOT) コンパイルすることを推奨します (このとき、-g -O0 を使用するとビルドに時間がかかる場合があります)。詳細については、「コンパイルの手順」をご覧ください。

GPU 向けの事前コンパイルを行う場合、ターゲットデバイスに対応したデバイスタイプを指定する必要があります。

次のコマンドを使用して、現在のマシンで利用可能な GPU デバイスオプションを確認できます:。

```
$ ocloc compile --help
```

さらに、カーネルのデバッグモードを有効にします。次の AOT コンパイルの例は、KBL デバイスをターゲットにしています。

```
$ dpcpp -g -O0 -fsycl-targets=spir64_gen-unknown-unknown-sycldevice \
-Xs "-device kbl -internal_options -cl-kernel-debug-enable -options -cl-opt-disable"
myprogram.cpp
```

6.3.3.3.6. インテル® ディストリビューションの GDB で使用する DPC++ 実行形式をビルド

プログラムのコンパイルとリンクに -g -O0 フラグを使用します。次に例を示します。

```
$ icpx -fiopenmp -O0 -fopenmp-targets=spir64 -c -g myprogram.cpp
$ icpx -fiopenmp -O0 -fopenmp-targets=spir64 -g myprogram.o
```

次の環境変数を設定して最適化を無効にし、カーネルのデバッグ情報を有効にします。

```
$ export LIBOMPTARGET_OPENCL_COMPILATION_OPTIONS="-g -cl-opt-disable"
$ export LIBOMPTARGET_LEVEL0_COMPILATION_OPTIONS="-g -cl-opt-disable"
```

6.3.3.4. GPU 実行をデバッグ

オフロードプログラムの一般的な問題は、実行に失敗し、追加情報をほとんど持たない OpenCL* エラーが生成されることです。OpenCL* アプリケーションのインターセプト・レイヤーと onetrace、ze_tracer、および cl_tracer を使用して、このエラーに関する詳細情報を取得できます。これは、開発者が問題の原因を特定するのに役立ちます。

6.3.3.4.1. OpenCL* アプリケーションのインターセプト・レイヤー

このライブラリーを使用する場合、Buildlogging、ErrorLogging、および USMChecking=1 オプションを使用してエラーの原因を特定できます。

1. 次のテキストを含む clintercept.conf ファイルをホーム・ディレクトリーに作成します。

```
SimpleDumpProgramSource=1
CallLogging=1
LogToFile=1
//KernelNameHashTracking=1
BuildLogging=1
ErrorLogging=1
USMChecking=1
//ContextCallbackLogging=1
// Profiling knobs KernelInfoLogging=1 DevicePerformanceTiming=1
DevicePerformanceTimeLWSTracking=1
DevicePerformanceTimeGWSTracking=1
```

2. 次のように cliloader を使用してアプリケーションを実行します。

```
<OCL_Intercept_Install_Dir>/bin/cliloader/cliloader -d ./<app_name> <app_args>
```

3. ~CLIntercept_Dump/<app_name> ディレクトリーで次の結果を確認します。

   - clintercept_report.txt: プロファイルの結果
   - clintercept_log.txt: OpenCL* の問題をデバッグする際に使用される OpenCL* 呼び出しログ

次のテキストは、CL_INVALID_ARG_VALUE (-50) ランタイムエラーが発生したプログラムで生成されたログファイルの例の一部です。

```
...
<<<< clSetKernelArgMemPointerINTEL -> CL_SUCCESS >>>>
clGetKernelInfo( _ZTSZZ10outer_coreiP5mesh_i16dpct_type_1c0e3516dpct_type_60257cS2_S2_S2_S2_S
2_S2 _S2_S2_fS2_S2_S2_S2_iENKUlRN2cl4sycl7handlerEE197->45clES6_EUlNS4_7nd_itemILi3EEEE225-
>13 ): param_name = CL_KERNEL_CONTEXT (1193)
<<<< clGetKernelInfo -> CL_SUCCESS >>>>
clSetKernelArgMemPointerINTEL( _ZTSZZ10outer_coreiP5mesh_i16dpct_type_1c0e3516dpct_type_60257
cS2_ S2_S2_S2_S2_S2_S2_S2_fS2_S2_S2_S2_iENKUlRN2cl4sycl7handlerEE197-
>45clES6_EUlNS4_7nd_itemILi3EEEE225->13 ): kernel = 0xa2d51a0, index = 3, value = 0x41995e0
mem pointer 0x41995e0 is an UNKNOWN pointer and no device support shared system pointers!
ERROR! clSetKernelArgMemPointerINTEL returned CL_INVALID_ARG_VALUE (-50)
<<<< clSetKernelArgMemPointerINTEL -> CL_INVALID_ARG_VALUE
```

この例は、次の値がエラーのデバッグに役立ちます。

- ZTSZZ10outer_coreiP5mesh
- index = 3, value = 0x41995e0

このデータによりどのカーネルに問題があるか、またどの引数に問題があるかが分かり、その理由を特定できます。

6.3.3.4.2. onetrace、ze_tracer、および cl_tracer

OpenCL* アプリケーションのインターセプト・レイヤーと同様に、onetrace、ze_tracer および cl_tracer ツールはレベルゼロのランタイムエラーの原因を検出するのに役立ちます。

onetrace または ze_tracer ツールを使用してレベルゼロにおける問題の根本的な原因を特定します (cl_tracer は、OpenCL* の問題の同様の原因を特定するのに使用されます)。

1. 呼び出しログモードでアプリケーションを実行します。ツールの出力をファイルにリダイレクトすることを推奨します。

```
$ ./onetrace -c ./<app_name> <app_args> [2> log.txt]
```

ze_tracer のコマンドも同様です。`onetrace` を `ze_tracer` に置き換えるだけです。

1. 呼び出しトレースを確認してエラーを特定します (`log.txt`)。次に例を示します。

```
>>>> [102032049] zeKernelCreate: hModule = 0x55a68c762690 desc = 0x7fff865b5570 {29 0 0 GEMM}
phKernel = 0x7fff865b5438 (hKernel = 0)
<<<< [102060428] zeKernelCreate [28379 ns] hKernel = 0x55a68c790280 -> ZE_RESULT_SUCCESS (0)
...
>>>> [102249951] zeKernelSetGroupSize: hKernel = 0x55a68c790280 groupSizeX = 256 groupSizeY =
1 groupSizeZ = 1
<<<< [102264632] zeKernelSetGroupSize [14681 ns] -> ZE_RESULT_SUCCESS (0)
>>>> [102278558] zeKernelSetArgumentValue: hKernel = 0x55a68c790280 argIndex = 0 argSize = 8
pArgValue = 0x7fff865b5440
<<<< [102294960] zeKernelSetArgumentValue [16402 ns] -> ZE_RESULT_SUCCESS (0)
>>>> [102308273] zeKernelSetArgumentValue: hKernel = 0x55a68c790280 argIndex = 1 argSize = 8
pArgValue = 0x7fff865b5458
<<<< [102321981] zeKernelSetArgumentValue [13708 ns] -> ZE_RESULT_ERROR_INVALID_ARGUMENT
(2013265924)
>>>> [104428764] zeKernelSetArgumentValue: hKernel = 0x55af5f3ca600 argIndex = 2 argSize = 8
pArgValue = 0x7ffe289c7e60
<<<< [104442529] zeKernelSetArgumentValue [13765 ns] -> ZE_RESULT_SUCCESS (0)
>>>> [104455176] zeKernelSetArgumentValue: hKernel = 0x55af5f3ca600 argIndex = 3 argSize = 4
pArgValue = 0x7ffe289c7e2c
<<<< [104468472] zeKernelSetArgumentValue [13296 ns] -> ZE_RESULT_SUCCESS (0) ...
```

この例のログには次のデータが示されています。

- 問題の原因となるレベルゼロ API 呼び出し (`zeKernelSetArgumentValue`)
- 問題の原因 (`ZE_RESULT_ERROR_INVALID_ARGUMENT`)
- 引数インデックス (`argIndex = 1`)
- 不正な値の場所 (`pArgValue = 0x7fff865b5458`)
- カーネルハンドル (`hKernel = 0x55a68c790280`)、この問題が検出されたカーネル名を示します (`GEMM`)

「ファイルへのリダイレクト」オプションを省略して、すべての出力 (アプリケーションの出力 + ツールの出力) を 1 つのストリームにダンプすることで、より多くの情報を取得できます。単一のストリームにダンプを行うことで、アプリケー

ションの出力に関連するエラーの原因を特定するのに役立つことがあります (例えば、アプリケーションの初期化と計算の最初のフェーズでエラーが発生しているなどが分かります)。

```
Level Zero Matrix Multiplication (matrix size: 1024 x 1024, repeats 4 times) Target device:
Intel® Graphics [0x3ea5]
...
>>>> [104131109] zeKernelCreate: hModule = 0x55af5f39ca10 desc = 0x7ffe289c7f80 {29 0 0 GEMM}
phKernel = 0x7ffe289c7e48 (hKernel = 0)
<<<< [104158819] zeKernelCreate [27710 ns] hKernel = 0x55af5f3ca600 -> ZE_RESULT_SUCCESS
(0) ...
>>>> [104345820] zeKernelSetGroupSize: hKernel = 0x55af5f3ca600 groupSizeX = 256 groupSizeY =
1 groupSizeZ = 1
<<<< [104360082] zeKernelSetGroupSize [14262 ns] -> ZE_RESULT_SUCCESS (0)
>>>> [104373679] zeKernelSetArgumentValue: hKernel = 0x55af5f3ca600 argIndex = 0 argSize = 8
pArgValue = 0x7ffe289c7e50
<<<< [104389443] zeKernelSetArgumentValue [15764 ns] -> ZE_RESULT_SUCCESS (0)
>>>> [104402448] zeKernelSetArgumentValue: hKernel = 0x55af5f3ca600 argIndex = 1 argSize = 8
pArgValue = 0x7ffe289c7e68
<<<< [104415871] zeKernelSetArgumentValue [13423 ns] -> ZE_RESULT_ERROR_INVALID_ARGUMENT
(2013265924)
>>>> [104428764] zeKernelSetArgumentValue: hKernel = 0x55af5f3ca600 argIndex = 2 argSize = 8
pArgValue = 0x7ffe289c7e60
<<<< [104442529] zeKernelSetArgumentValue [13765 ns] -> ZE_RESULT_SUCCESS (0)
>>>> [104455176] zeKernelSetArgumentValue: hKernel = 0x55af5f3ca600 argIndex = 3 argSize = 4
pArgValue = 0x7ffe289c7e2c
<<<< [104468472] zeKernelSetArgumentValue [13296 ns] -> ZE_RESULT_SUCCESS (0) ...
Matrix multiplication time: 0.0427564 sec Results are INCORRECT with accuracy: 1 ...
Matrix multiplication time: 0.0430995 sec Results are INCORRECT with accuracy: 1 ...
Total execution time: 0.381558 sec
```

6.3.3.5. 正当性

オフロードコードは、接続された計算デバイスで大量の情報を効率良く処理するカーネル、または一部の入力パラメーターから大量の情報を生成する際に利用されます。それらのカーネルがクラッシュすることなく実行されると、多くの場合、正しい結果が得られていないことはプログラム実行のかなり後で判明します。

そのような場合、どのカーネルが誤った結果を生成しているか特定するのは困難です。誤った結果を生成するカーネルを特定する方法として、プログラムを 2 回実行することが考えられます。最初はホストベースの実装を実行し、2 回目はオフロード実装を実行してすべてのカーネル (多くは個々のファイル) の入力と出力を取得します。次に、結果を比較して、どのカーネルが予期しない結果を生成しているか確認します (特定のイプシロンで、オフロード・ハードウェアの操作順序やネイティブの精度の違いにより、結果の最後の 1 桁もしくは 2 桁がホストコードと異なる可能性があります)。

誤った結果を生成するカーネルが特定されたら、インテル® ディストリビューションの GDB を使用して原因を調査します。基本情報と詳細なドキュメントへのリンクについては、『インテル® ディストリビューションの GDB を使用したデバッグのチュートリアル (Linux* 版)』(英語) を参照してください。

SYCL* と OpenMP* はどちらもオフロードされたカーネル内で標準のプリントメカニズム (SYCL* と C++ OpenMP* では `printf`、Fortran OpenMP* オフロードでは `print *` など) を利用できます。これを使用して、実行中の動作を確認できます。出力元のスレッドと SIMD レーンをプリントし、プリントされる情報がプリント時に一貫性を持つように、同期メカニズムを追加することを検討してください。ストリームクラスを使用して DPC++ で同様のことを行う例は、『oneAPI GPU 最適化ガイド』記載されています。OpenMP* オフロード向けの SYCL* の説明を同様のアプローチを適用できます。

OpenMP* ディレクティブを使用してアプリケーションに並列処理を追加する方法の詳細については、「インテルツールによる OpenMP* アプリケーションのオフロードと最適化」(英語) を参照してください。

ヒント:

SYCL* カーネルでは `printf` は冗長的になる可能性があります。簡単にするため次のマクロを追加します。

```
#ifdef __SYCL_DEVICE_ONLY
  #define CL_CONSTANT __attribute__((opencl_constant))
#else
  #define CL_CONSTANT
#endif
#define PRINTF(format, ...){ \
    static const CL_CONSTANT char _format[] = format; \
    sycl::ONEAPI::experimental::printf(_format, ## __VA_ARGS__); }
```

使用例: `PRINTF("My integer variable:%d\n", (int) x);`

6.3.3.6. 障害

SYCL* または OpenMP* オフロード言語の誤った用法が原因で JIT コンパイルが失敗すると、プログラムはエラーで終了します。

SYCL* では事前コンパイルされていることを判定できない場合、OpenCL* バックエンドを選択して 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と、構文エラーを持つカーネルを特定できることがあります。

ロジックエラーは、実行中にクラッシュが発生したり、エラーメッセージが表示されることがあります。これには以下が含まれます:

- 誤ったコンテキストに属するバッファーをカーネルに渡す場合
- `this` ポインターをクラス要素ではなくカーネルに渡す場合
- デバイスバッファーではなくホストバッファーを渡す場合
- カーネルで使用されなくても、初期化されていないポインターを渡す場合

インテル® ディストリビューションの GDB (またはネイティブ GDB) を使用して注意深く調査することで、生成されたすべてのコンテキストのアドレスを記録してオフロードカーネルに渡されるアドレスが正しいコンテキストに属するか確認できます。同様に、変数のアドレスがそれを含むクラスでなく、変数自身のアドレスと一致するか確認できます。

OpenCL* 割り当て用のインターセプト・レイヤーまたは、onetrace/cl_tracer を使用して、適切なバックエンドを選択する方がバッファーとアドレスをトレースするよりも簡単なことがあります。OpenCL* バックエンドを使用する場合、`CallLogging`、`BuildLogging`、`ErrorLogging` および `USMChecking` を設定してプログラムを実行すると、コード内のどのエラーが OpenCL* エラーを生成したか明らかにする出力が生成されます。

onetrace や ze_tracer の呼び出しログやデバイス・タイムラインを参照すると、レベルゼロのバックエンドからのエラーの原因を理解するのに役立つ追加のエラー情報が得られます。これは、上記の論理エラーを検出するのに役立ちます。

レベルゼロ・バックエンドを使用してデバイスにオフロードする際にコードでエラーが発生する場合、OpenCL* バックエンドを試してみてください。プログラムが正常に機能する場合、レベルゼロのバックエンドにエラーをレポートしてください。デバイス向けの OpenCL* バックエンドでもエラーが再現する場合、OpenCL* CPU バックエンドを試します。OpenMP* オフロードでは、`OMP_TARGET_OFFLOAD` を CPU に設定することで指定できます。SYCL* では、`ONEAPI_DEVICE_SELECTOR=opencl:cpu` を設定します。CPU 上でのデバッグはかなり容易になり、データのコピーとプログラムのデバイスへの変換によって生じる複雑性も排除できます。

問題が発生する可能性があるロジックの例として、次の SYCL* コードで `parallel_for` を実装する際に使用されるラムダ関数でキャプチャーされる対象を考えます。

```
class MyClass {
private:
  int *data;
  int factor;
  :
void run() {
  :
  auto data2 = data;
  auto factor2 = factor;
  {
    dpct::get_default_queue_wait().submit([&](cl::sycl::handler &cgh)
    {
       auto dpct_global_range = grid * block;
       auto dpct_local_range = block;
       cgh.parallel_for<dpct_kernel_name<class kernel_855a44>>(
           cl::sycl::nd_range<1>(
             cl::sycl::range<1> dpct_global_range.get(0)),
             cl::sycl::range<1>( dpct_local_range.get(0))),
             [=](cl::sycl::nd_item<3> item_ct1)
       {
           kernel(data, b, factor, LEN, item_ct1);    // This blows up
       });
     });
   }
} // run
} // MyClass
```

上記のコードでは、`[=]` がラムダ内で使用される変数を値でコピーするため、プログラムはクラッシュします。この例では、`factor` は実際には `this->factor` で、`data` は `this->data` であるため、`this` は `data` および `factor` を使用するために取得される変数です。OpenCL* またはレベルゼロでは、`kernel(data, b, factor, LEN, item_ct1)` 呼び出しで不正な引数エラーが原因でクラッシュします。

この問題を解決するには、ローカル変数 `auto data2` と `auto factor2` を使用します。`auto factor2 = factor` は `int factor2 = this->factor` になるため、ラムダ内で `[=]` を指定して `factor2` を使用すると、`int` が取得されます。内部セクションを `kernel(data2, b, factor2, LEN, item_ct1);` に変更します。

注: この問題は、CUDA* カーネルを移行する際によく発生します。同じ CUDA* カーネルの起動シグネチャーを保持し、コマンドグループとラムダをカーネル内に配置することで問題を解決することもできます。

OpenCL* 割り当てのインターセプト・レイヤーや onetrace または ze_tracer を使用すると、カーネルが 2 つの同一アドレスで呼び出されることが分かり、拡張エラー情報を見ると、重要なデータ構造をオフロードデバイスにコピーしようとしていることを確認できます。

統合共有メモリー (USM) を使用して `MyClass` を USM に割り当てる場合、上記のコードが動作することに注意してください。ただし、USM に `data` のみが割り当てられている場合、前述の理由からプログラムはクラッシュします。

この例では、カーネル呼び出しですべてを変更する必要がないように、ローカルスコープ内で同じ名前の変数を再宣言できることも留意してください。

インテル® Inspector は、このような障害の診断に役立ちます。次の環境変数を設定して、CPU デバイスでオフロードコードのメモリー解析を実行すると、インテル® Inspector は上記の問題の多くを通知します。

- OpenMP*
  - `export OMP_TARGET_OFFLOAD=CPU`
  - `export OMP_TARGET_OFFLOAD=MANDATORY`
  - `export LIBOMPTARGET_PLUGIN=OPENCL`
- SYCL*
  - `export ONEAPI_DEVICE_SELECTOR=opencl:cpu`
  - または、CPU セレクターを使用してキューを初期化し、OpenCL* CPU デバイスの仕様を強制します。
    ```
    cl::sycl::queue Queue(cl::sycl::cpu_selector{});
    ```
- 両方
  - `export CL_CONFIG_USE_VTUNE=True`
  - `export CL_CONFIG_USE_VECTORIZER=false`

注: コンパイル中に最適化を有効にするとクラッシュする可能性があります。最適化を無効にしてクラッシュが解決される場合、デバッグ向けに `-g -[最適化レベル]` を指定します。詳細については、『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) を参照してください。

6.3.3.7. SYCL* 例外ハンドラーを使用する

『Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL』(英語) の書籍では次のことが説明されています。

C++ の例外機能は、プログラム内でエラーが検出された位置と、エラーが処理される位置は明確に分離するように設計されており、この概念は SYCL* の同期エラーと非同期エラーの両方にも適合します。

この書籍で推奨されるメソッドを使用すると、C++ 例外処理は、エラー発生時にプログラムが何の通知もなく終了するのではなく、なんらかの通知を行って終了するのに役立ちます。

注: この節の紺色で示したテキストは、C++ と SYCL* を使用する異種システムのプログラミングを説明する『Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*』の第 5 章「Error Handling」からの抜粋です。簡素化のため一部のテキストを省略しています。詳細は書籍を参照してください。

エラー処理の無視

C++ と SYCL* では、エラーを明示的に処理しなくても問題が発生したことを通知できるように設計されています。未処理の同期または非同期のエラーのデフォルトの結果は、オペレーティング・システムが通知するプログラムの異常終了になります。次の 2 つの例は、それぞれ同期エラーと非同期エラーを処理しない場合に発生する動作を模倣します。

以下のコード例は、ハンドルされていない C++ 例外の結果を示しています。これは、ハンドルされていない SYCL* 同期エラーなどが原因である可能性があります。このコードでは、特定のオペレーティング・システムがどのように振舞うかテストできます。

C++ の未処理例外

```
#include <iostream>

class something_went_wrong {};

int main() {
std::cout << "Hello\n";

throw(something_went_wrong{});
}

Linux* での実行例:

$ Hello
terminate called after throwing an instance of 'something_went_wrong'

Aborted (core dumped)
```

以下のコード例は、呼び出された `std::terminate` からの出力例を示します。これは、アプリケーションで未処理の SYCL* 非同期エラーが発生した結果です。このコードでは、特定のオペレーティング・システムがどのように振舞うかテストできます。

プログラムでエラーを処理する必要がありますが、キャッチされないエラーをキャッチしてからプログラムが終了するため、プログラムが何も通知することなく失敗する心配がありません。

`std::terminate` は SYCL* 非同期例外が処理されないときに呼び出される

```
#include <iostream>

int main() {
std::cout << "Hello\n";

std::terminate();
}

Linux* での実行例:

$ Hello
terminate called without an active exception Aborted (core dumped)
```

このドキュメントでは、同期エラーを C++ の例外で処理できる理由を詳しく説明していますが、アプリケーションで制御する位置で非同期エラーを処理するには、SYCL* 例外が呼び出される状況に注意して、SYCL* 例外を利用する必要があります。

SYCL* で定義される同期エラーは、`std::exception` タイプからの派生クラスであり、以下に示すように `try-catch` 構造によって SYCL* エラーをキャッチできます。

`sycl::exception` をキャッチするパターン

```
try{
// SYCL* ワークを実行
} catch (sycl::exception &e) {
// 例外を出力または処理するため何かを実行
std::cout << "Caught sync SYCL exception: " << e.what() << "\n"; return 1;
}
```

C++ のエラー処理メカニズムに加えて、SYCL* ではランタイムによってスローされる `sycl::exception` 例外タイプを追加します。それ以外はすべて標準 C++ の例外処理であるため、開発者には馴染みがあります。さらに詳しい例を以下に示します。ここでは、追加の例外クラスが処理され、`main()` からリターンすることでプログラムが終了します。C++ のエラー処理メカニズムに加えて、SYCL* ではランタイムによってスローされる `sycl::exception` 例外タイプを追加します。それ以外はすべて標準 C++ の例外処理であるため、開発者には馴染みがあります。さらに詳しい例を以下に示します。ここでは、追加の例外クラスが処理され、`main()` からリターンすることでプログラムが終了します。

コードブロックから例外をキャッチするパターン

```
try{
buffer<int> B{ range{16} };

// ERROR: 親バッファーよりも大きな sub-buffer を定義
// バッファー・コンストラクターから例外がスローされます
buffer<int> B2(B, id{8}, range{16});

} catch (sycl::exception &e) {
// 例外を出力または処理するため何かを実行
std::cout << "Caught sync SYCL exception: " << e.what() << "\n";
return 1;
} catch (std::exception &e) {
std::cout << "Caught std exception: " << e.what() << "\n";
return 2;
} catch (...){
std::cout << "Caught unknown exception\n";
return 3;
}

return 0;

出力例:
Caught sync SYCL exception: Requested sub-buffer size exceeds the
size of the parent buffer -30 (CL_INVALID_VALUE)
```

非同期エラー処理

非同期エラーは SYCL* ランタイム (またはベースとなるバックエンド) によって検出され、エラーはホストプログラムのコマンドの実行とは無関係に発生します。エラーは SYCL* ランタイムの内部リストに格納され、プログラマーが制御できる特定の位置でのみ処理を行うためリリースされます。非同期エラーの処理をカバーするのに次の 2 つのことを理解してください。

1. 処理すべき未処理の非同期エラーがある場合に呼び出される**非同期ハンドラー**
2. **いつ**非同期ハンドラーが起動されるか

非同期ハンドラーは、アプリケーションが定義する関数であり、SYCL* コンテキストやキューに登録されます。次のセクションで定義された時点で処理可能な未処理の非同期例外がある場合、SYCL* ランタイムによって非同期ハンドラーが呼び出されて例外リストが渡されます。非同期ハンドラーは `astd::function` としてコンテキストまたはキューのコンストラクターに渡され、通常の関数、ラムダ、ファンクターなどで定義できます。ハンドラーは、以下に示すように `sycl::exception_list` 引数を受け入れる必要があります。

ラムダとして定義された非同期ハンドラーの実装例

```
// 単純な非同期ハンドラー関数
auto handle_async_error = [](exception_list elist) {
for (auto &e : elist) {
try{ std::rethrow_exception(e); }
catch ( sycl::exception& e ) {
std::cout << "ASYNC EXCEPTION!!\n";
std::cout << e.what() << "\n";
}
}
};
```

上記の例では、`std::rethrow_exception` の後に特定例外タイプの `catch` が続き、例外タイプ (この場合 `sycl::exception` のみ) のフィルター処理を提供しています。C++ で異なるフィルター処理を行うことも、タイプにかかわりなくすべての例外を処理することもできます。ハンドラーは、構築時にキューまたはコンテキストに関連付けられます (低レベルの詳細については第 6 章 (英語) で詳しく説明されています)。例えば、上記のリストで定義されたハンドラーをキューに登録するには、`queue my_queue{ gpu_selector{}, handle_async_error }` のように記述し、ハンドラーをコンテキストに登録するには、`context my_context{ handle_async_error }` のように記述します。

ほとんどのアプリケーションでは、コンテキストを明示的に作成したり管理する必要はありません (バックグラウンドで自動的に作成されます)。そのため、非同期ハンドラーを使用する場合、そのハンドラーを特定のデバイスのキューに関連付ける必要があります (明示的なコンテキストではありません)。

注: 非同期ハンドラーを定義する場合、何らかの理由でコンテキストを明示的に管理しない限り、キューで定義する必要があります。

キューやその親コンテキストに対して非同期ハンドラーが定義されていないと、そのキュー (またはコンテキスト) で処理が必要な非同期エラーが発生した場合、デフォルトの非同期ハンドラーが呼び出されます。デフォルトのハンドラーは次のリストと同じように動作します。

デフォルトの非同期ハンドラーの例

```
// 単純な非同期ハンドラー関数
auto handle_async_error = [](exception_list elist) {
for (auto &e : elist) {
try{ std::rethrow_exception(e); }
catch ( sycl::exception& e ) {
// 非同期例外に関連する情報を出力
}
}

// 異常終了して、未処理の例外があることを
// ユーザーに通知します
std::terminate();
};
```

デフォルトのハンドラーは、例外リスト内のエラー情報をユーザーに通知し、アプリケーションを異常終了させます。この場合、オペレーティング・システムも異常終了したことをレポートする必要があります。

非同期ハンドラーにどのようなエラーを知らせるかはプログラマーに依存します。アプリケーションが処理を正常に続行できるよう、エラーのログにはアプリケーションの終了、そしてエラー状態の回復までさまざまな情報があります。

一般には、`sycl::exception::what()` を呼び出して利用可能なエラーの詳細を通知し、アプリケーションを終了させます。非同期ハンドラーが内部で何を処理するかを決定するのはプログラマー次第ですが、しばしば見られる間違いとして、エラーメッセージ (プログラムのほかのメッセージで見逃される可能性があります) を出力してから、ハンドラー関数を終了することです。プログラムの状態を回復して、実行を継続しても安全であると確信できるようエラー管理が成されていない限り、非同期ハンドラー関数内でアプリケーションを終了することを検討してください。

これにより、エラーが検出されたにもかかわらずアプリケーションの不正実行を続行するプログラムから誤った結果が表示される可能性が減少します。ほとんどのプログラムでは、非同期例外が発生したら異常終了することが適切です。

6.3.3.7.1. 例: サイズゼロのオブジェクトでの SYCL* のスロー

次のサンプルコードは、サイズがゼロのオブジェクトが渡されたときに SYCL* ハンドラーがどのようにエラーを生成するかを示しています。

```
1.  #include <cstdio>
2.  #include <CL/sycl.hpp>
3.
4.  template <bool non_empty>
5.  static void fill(sycl::buffer<int> buf, sycl::queue & q) {
6.    q.submit([&](sycl::handler & h) {
7.      auto acc = sycl::accessor { buf, h, sycl::read_write };
8.      h.single_task([=]() {
9.        if constexpr(non_empty) {
10.         acc[0] = 1;
11.       }
12.     }
13.     );
14.   }
15.   );
16.   q.wait();
17.
18. }
19.
20. int main(int argc, char *argv[]) {
21.   sycl::queue q;
22.   sycl::buffer<int, 1> buf_zero ( 0 );
23.
24.   fprintf(stderr, "buf_zero.count() = %zu\n", buf_zero.get_count());
25.   fill<false>(buf_zero, q);
26.   fprintf(stdout, "PASS\n");
27.
28.   return 0;
29. }
```

アプリケーションが実行中にサイズがゼロのオブジェクトに遭遇すると、プログラムはアボートし、エラーメッセージが出力されます。

```
$ dpcpp zero.cpp
$./a.out
buf_zero.count() = 0
submit...
terminate called after throwing an instance of 'cl::sycl::invalid_object_error'
  what(): SYCL buffer size is zero. To create a device accessor, SYCL buffer size must be
greater than zero. -30 (CL_INVALID_VALUE)
Aborted (core dumped)
```

プログラマーは、デバッガーで例外をキャッチし、エラーを招いたソース行のバックトレースを調査することで、プログラミングの問題を特定できます。

6.3.3.7.2. 例: 不正な NULL ポインターでの SYCL* スロー

以下のコードについて考えてみます。

```
deviceQueue.memset(mdlReal, 0, mdlXYZ \* sizeof(XFLOAT));
deviceQueue.memcpy(mdlImag, 0, mdlXYZ \* sizeof(XFLOAT)); // コーディング・エラー
```

コンパイラーは、`deviceQueue.memcpy` で指定された不正な (NULL ポインター) 値のフラグをセットしません。このエラーは、実行されるまでキャッチされません。

```
terminate called after throwing an instance of 'cl::sycl::runtime_error'

what(): NULL pointer argument in memory copy operation.-30 (CL_INVALID_VALUE)

Aborted (core dumped)
```

次のコード例は、NULL ポインターのエラーを示すプログラムに実装された、特定キューでの実行時の例外出力が検出された際に、ユーザーが例外出力の形式を制御する方法を示しています。

```
1.  #include "stdlib.h"
2.  #include <stdio.h>
3.  #include <cmath>
4.  #include <signal.h>
5.  #include <fstream>
6.  #include <iostream>
7.  #include <vector>
8.  #include <CL/sycl.hpp>
9.
10. #define XFLOAT float
11. #define mdlXYZ 1000
12. #define MEM_ALIGN 64
13.
14. int main(int argc, char *argv[]) {
15.   XFLOAT *mdlReal, *mdlImag;
16.
17.   cl::sycl::property_list propList =
18.       cl::sycl::property_list{cl::sycl::property::queue::enable_profiling()};
19.   cl::sycl::queue deviceQueue(cl::sycl::gpu_selector { },
20.       [&](cl::sycl::exception_list eL) {
```

```
21.     bool error = false;
22.     for (auto e : eL) {
23.       try {
24.         std::rethrow_exception(e);
25.       } catch (const cl::sycl::exception& e) {
26.         auto clError = e.get_cl_code();
27.         bool hascontext = e.has_context();
28.         std::cout << e.what() << "CL ERROR CODE : " << clError << std::endl;
29.         error = true;
30.         if (hascontext) {
31.           std::cout << "We got a context with this exception" << std::endl;
32.         }
33.       }
34.     }
35.     if (error) {
36.       throw std::runtime_error("SYCL errors detected");
37.     }
38.   }, propList);
39.
40.   mdlReal   = sycl::malloc_device<XFLOAT>(mdlXYZ, deviceQueue);
41.   mdlImag   = sycl::malloc_device<XFLOAT>(mdlXYZ, deviceQueue);
42.
43.   deviceQueue.memset(mdlReal, 0, mdlXYZ * sizeof(XFLOAT));
44.   deviceQueue.memcpy(mdlImag, 0, mdlXYZ * sizeof(XFLOAT)); // コーディング・エラー
45.
46.   deviceQueue.wait();
47.
48.   exit(0);
49. }
```

6.3.3.8. リソース

SYCL* API の誤った使用による SYCL* 例外をデバッグするガイド付きのアプローチについては、「ガイド付き行列乗算の例外のサンプル」(英語) を参照してください。

リソースをオフロードする拡張機能を備えた OpenMP* または SYCL* API を使用するアプリケーションのトラブルシューティングは、「高度な並列アプリケーションのトラブルシューティング」(英語) のチュートリアルを参照してください。

6.3.4 オフロードのパフォーマンスを最適化

オフロード・パフォーマンスの最適化は、3 つの作業に要約できます:

1. デバイス上のカーネルの実行時間を最大化しつつ、デバイス間とのデータ転送回数とサイズを最小化します。
2. 可能であれば、デバイス上の計算とデバイスとのデータ転送をオーバーラップさせます。
3. デバイス上のカーネルのパフォーマンスを最大化します。

OpenMP* オフロードと SYCL*, の両方でデータ転送を明示的に制御することができますが、これを自動的に行うこともできます。また、ホストとオフロードデバイスはほとんど非同期に動作するため、データ転送を制御しようとしても転送が期待どおりに行われず、予想よりも時間を要することがあります。デバイスとホストの両方でアクセスされるデータが

統合共有メモリー (USM) に格納されている場合、パフォーマンスに影響する透過的な別のレイヤーでデータ転送が行われる可能性があります。

リソース:

- oneAPI GPU 最適化ガイド
- インテル® oneAPI ツールキット向け FPGA 最適化ガイド (英語)

6.3.4.1. バッファー転送時間と実行時間

オフロードデバイス間とのデータ転送には比較的コストがかかり、ユーザー空間でのメモリー割り当て、システムコール、およびハードウェア・コントローラーとのインターフェイスが必要になります。統合共有メモリー (USM) は、ホストまたはオフロードデバイスのいずれかのメモリーの更新を同期するバックグラウンド・プロセスによるコストがかかります。さらに、オフロードデバイス上のカーネルは、実行に必要なすべての入力および出力バッファーがセットアップされて利用可能になるまで待機する必要があります。

1 回のデータ転送でオフロードデバイスとやり取りする情報量にかかわりなく、このオーバーヘッドのコストはほぼ同じです。そのため、1 回に 1 つずつではなく、10 回分の転送をまとめて行うほうがはるかに高効率です。いずれにしても、すべてのデータ転送にはコストが生じるため、転送の総数を最小限にすることが重要です。例えば、複数のカーネルまたは同じカーネルの複数の呼び出しで必要とする定数がある場合、それらをカーネルを呼び出すたびに送信するのではなく、一度だけオフロードデバイスに転送して再利用するようにします。最後に、単一の大量のデータ転送には、単一の少量のデータ転送よりも時間がかかります。

送信されるバッファーの数とサイズは式の一部にすぎません。データがオフロードデバイスに到達したら、カーネルが実行される時間を検討します。オフロードデバイスへのデータ転送よりも実行時間が短い場合、同一操作をホストで実行する時間が、カーネルの実行とデータ転送の合計時間よりも長くない限り、オフロードのメリットはありません。

最後に、あるカーネルの実行と次のカーネルの実行の間に、オフロードデバイスがアイドル状態になっている時間を調査します。長い待機時間は、データ転送やホスト上のアルゴリズムの特性が原因である可能性があります。前者は、データ転送とカーネル実行のオーバーラップを試す価値があります。

ホストでのコード実行、オフロードデバイスでのコード実行、およびデータ転送の関係は複雑です。これらの順番と時間は、単純なコードであっても直感的に理解できるものではありません。すべてのアクティビティーを視覚的に理解し、その情報を参考にしてオフロードコードを最適化するには、次のようなツールが必要になります。

6.3.4.2. 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、ホスト上の詳しいパフォーマンス情報を提供するだけでなく、接続された GPU のパフォーマンスに関する詳細情報も提供できます。GPU 向けの設定に関する情報は、『インテル® VTune™ プロファイラー・ユーザーガイド』をご覧ください。

インテル® VTune™ プロファイラーの GPU オフロードビューには、それぞれのカーネル間とのデータ転送に費やされた時間など、GPU 上のホットスポットに関するサマリーが表示されます。GPU 計算/メディア・ホットスポット・ビューでは、**動的命令カウント**を使用して GPU カーネルのパフォーマンスのマイクロ解析など、GPU カーネルで何が起こっているか詳しく調査することができます。これらのプロファイル・モードでは、データ転送と計算が時間経過でどのように変化するか観察したり、カーネルを効率良く実行するのに十分なワークがあるか判断したり、カーネルが GPU メモリー階層をどのように利用するか調べたりできます。

これらの解析タイプの詳細については、『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。インテル® VTune™ プロファイラーを使用した GPU の最適化に関する詳細は、「インテル® VTune™ プロファイラーでインテル® GPU 向けのアプリケーションの最適化」(英語) をご覧ください。

カーネルの実行時間を計測するには、インテル® VTune™ プロファイラーも使用できます。次のコマンドは、軽量プロファイルの結果を示します:

- 収集
  - レベルゼロ・バックエンド

```
$ vtune -collect-with runss -knob enable-gpu-level-zero=true -finalization-mode=none
-app-working-dir <app_working_dir> <app>
```

  - OpenCL* バックエンド

```
$ vtune -collect-with runss -knob collect-programming-api=true -finalization-mode=none -r
<result_dir_name> -app-working-dir <app_working_dir> <app>
```

- レポート

```
$ vtune --report hotspots --group-by=source-computing-task --sort-desc="Total Time" -r
<result_dir_name>
```

6.3.4.3. インテル® Advisor

インテル® Advisor は、計算を GPU にオフロードするパフォーマンスを向上させのに役立つ 2 つの機能を提供します。

- オフロードのモデル化は、ホストの OpenMP* プログラムを調査して、GPU へのオフロードに適したコード領域を示します。また、多種多様なターゲット向けに GPU をモデル化できるため、ターゲットに適したオフロードコード領域を特定できます。オフロード・アドバイザーは、オフロードのパフォーマンスを制限する可能性がある要因に関する詳細情報を提供します。
- GPU ルーフライン解析は、GPU で実行されるアプリケーションを監視し、各カーネルが GPU のメモリー・サブシステムと計算ユニットをどの程度効率良く使用しているか視覚的に示します。これにより、カーネルが GPU にどれくらい最適化されているか知ることができます。

すでにオフロードを実装するアプリケーションでこのモードを使用するには、CPU 上の OpenCL* デバイスを解析ターゲットにするよう環境を設定する必要があります。手順は、『インテル® Advisor ユーザーガイド』(英語) をご覧ください。

オフロードのモデル化では、GPU を使用するようにアプリケーションを変更する必要はありません。ホストコードで完全に機能します。

リソース:

- インテル® Advisor クックブック: GPU オフロード
- オフロードのモデル化入門 (英語)
- GPU ルーフライン入門 (英語)

6.3.4.4. オフロード API 呼び出しのタイムライン

インテル® VTune™ プロファイラーを使用して、データが GPU にコピーされるタイミング、およびカーネルが実行されるタイミングを調査したくない (またはできない) 場合、onetrace、ze_tracer、cl_tracer と OpenCL* アプリケーションのインターセプト・レイヤーでもこの情報を監視する方法が用意されています (ただし視覚的なランタイム情報が必要な場合、出力を視覚化するスクリプトを用意する必要があります)。詳細については、「oneAPI デバッグツール」、「オフロード処理のトレース」、および「オフロード処理のデバッグ」をご覧ください。

6.4 パフォーマンス・チューニング・サイクル

パフォーマンス・チューニング・サイクルの目標は、対話型の応答時間やバッチジョブの経過時間など、問題解決までの時間を短縮することです。ヘテロジニアス・プラットフォームの場合、ホストと独立して実行されるデバイスで利用可能な計算サイクルがあります。これらのリソースを利用してパフォーマンスを向上させます。

パフォーマンス・チューニング・サイクルには、次のステップがあります。

1. ベースラインの確定
2. オフロードするカーネルの特定
3. カーネルのオフロード
4. 最適化
5. 目標を達成するまで 1 ~ 4 を繰り返す

6.4.1 ベースラインの確定

経過時間、計算カーネル時間、1 秒あたりの浮動小数点演算などのメトリックを含むベースラインを確定します。これは、パフォーマンス向上の測定だけでなく、結果の正当性を検証する手段としても利用できます。

ベースラインを確定する簡単な方法は、C++ の chrono ライブラリー・ルーチンを使用して、ワークロードの実行前後でタイマー呼び出しを行い時間計測を行うことです。

6.4.2 オフロードするカーネルの特定

ヘテロジニアス・プラットフォームのデバイスで利用可能な計算サイクルを最大限に活用するには、計算集約型で並列実行可能なタスクを特定することが重要です。CPU のみで実行されるアプリケーションを調査すると、GPU で実行するのに適したタスクが見つかることがあります。これは、インテル® Advisor (英語) のオフロードのモデル化機能で判別できます。

インテル® Advisor は、アクセラレーターで実行できる可能性があるワークロードのパフォーマンス特性を推測できます。ワークロードのプロファイル情報から、パフォーマンスを見積もり、高速化、ボトルネックの特性を評価して、さらにオフロードデータ転送を推測し、推奨事項を示します。

一般に、計算主体で、大規模なデータセットを持ち、限られたデータ転送を行うカーネルは、デバイスへのオフロードに適しています。

オフロードのモデル化を使用した手順については、『導入ガイド: GPU にオフロードにより大きな効果が得られる候補の特定』(英語) を参照してください。GPU プラットフォームでのアプリケーションのモデル化のパフォーマンスに関するその他のリソースについては、「インテル® Advisor ユーザー向けオフロードのモデル化のリソース」(英語) を参照してください。

6.4.3 カーネルをオフロード

オフロードに適したカーネルを特定したら、SYCL* または OpenMP* を使用してカーネルをデバイスにオフロードします。詳細は前の節をご覧ください。

6.4.4 SYCL* アプリケーションの最適化

oneAPI は、CPU、GPU、FPGA など、複数のアクセラレーターで実行できるコードを生成します。ただし、コードはすべてのアクセラレーターで最適ではない可能性があります。パフォーマンスの目標を達成するには 3 段階の最適化を行うことを推奨します。

1. アクセラレーター全体に適用される一般的な最適化を行います。
2. 優先するアクセラレーターに対して積極的に最適化を行います。
3. ステップ 1 と 2 を組み合わせてホストコードを最適化します。

最適化とは、ボトルネック (ほかのコードセクションに比べて実行時間が長いコード領域) を排除する作業です。これらのセクションは、デバイスまたはホストで実行できます。最適化では、インテル® VTune™ プロファイラーなどのプロファイル・ツールを使用して、コード内のボトルネックを特定します。

ここでは、最初のステップであるアクセラレーター全体に適用される一般的な最適化を検討します。デバイス固有の最適化、デバイス固有のベスト・プラクティス (ステップ 2)、およびホストとデバイス間の最適化 (ステップ 3) については、『インテル® oneAPI ツールキット向け FPGA 最適化ガイド』(英語) などのデバイス固有の最適化ガイドで詳しく説明されています。この節では、アクセラレーターにオフロードするカーネルがすでに決定されていること、および単独のアク

セラレーターでワークが完了することを想定しています。このガイドは、ホストとアクセラレーター間、またはホストと複数の/異なるアクセラレーター間のワークの分割については考慮していません。

アクセラレーター全体に適用される一般的な最適化は次の 4 つに分類されます。

1. 高レベルの最適化
2. ループ関連の最適化
3. メモリー関連の最適化
4. SYCL* 固有の最適化

次の節では上記の最適化のみをカバーします。これらの最適化を実際にコードに反映する方法の詳細は、オンラインや一般に入手できるコード最適化関連の資料で見付けることができます。ここでは、SYCL* 固有の最適化に関する詳細を示します。

6.4.4.1. 高レベルの最適化

- 並列ワークの量を増やします。処理要素を十分に活用するには、処理要素よりも多くのワークが必要です。
- カーネルのコードサイズを最小化します。これにより、アクセラレーターの命令キャッシュにカーネルが保持されます。
- カーネルのロードバランスを取ります。実行時間の長いカーネルはボトルネックとなり、ほかのカーネルのスループットに影響する可能性があるため、カーネル間の実行時間は大きく異ならないようにします。
- 高コストの関数は避けてください。実行時間が長い関数は、ボトルネックとなる可能性があるため呼び出さないでください。

6.4.4.2. ループ関連の最適化

- 適切に構造化および構成された、単純な終了条件のループを使用します。単純な終了条件のループとは、出口が 1 つであり、整数上限値との比較で単一の終了条件を持つループです。
- 線形インデックスと定数上限値を持つループを優先します。例えば、配列への整数インデックスを使用し、コンパイル時に上限値が判明しているループなどです。
- 可能な限り深いスコープで変数を宣言します。これにより、メモリーまたはスタックの使用量を軽減できます。
- ループ伝搬されるデータの依存関係を最小化または緩和します。ループ伝搬の依存関係により、並列化が制限されることがあります。可能な限り依存関係を排除します。排除できない場合は、依存関係の距離を最大化するか、依存関係をローカルメモリー内に留めます。
- `pragma unroll` でループをアンロールします。

6.4.4.3. メモリー関連の最適化

- 可能な限り、メモリー使用よりも計算を優先します。メモリーのレイテンシーと帯域幅のほうが計算よりもボトルネックになる可能性があります。
- 可能であれば、グローバルメモリーよりもローカルおよびプライベート・メモリーを使用します。
- ポインターのエイリアシングを避けます。
- メモリーアクセスを結合します。メモリーアクセスをグループ化して個々のメモリー要求の数を減らし、キャッシュラインの利用率を高めます。
- 可能であれば、頻繁に実行されるコード領域の変数と配列をプライベート・メモリーに保持します。同時メモリーアクセスに対するループアンロールの影響に注意してください。
- 別のカーネルが読み取るグローバルメモリーへの書き込みを避けます。代わりにパイプを使用します。
- カーネルに `[[intel::kernel_args_restrict]]` 属性を適用することを検討してください。この属性により、コンパイラーはカーネルのアクセサー引数間の依存関係を無視できます。アクセサー引数の依存関係を無視すると、コンパイラーはさらに積極的な最適化を行い、カーネルのパフォーマンスが向上する可能性があります。

6.4.4.4. SYCL* 固有の最適化

- 可能であれば、work-group サイズを指定します。属性 `[[cl::reqd_work_group_size(X, Y, Z)]]` (`X`、`Y`、`Z` は Nd-range の整数次元) を使用して、work-group のサイズを設定できます。コンパイラーはこの情報を利用して積極的な最適化を行うことができます。
- 可能であれば、`-Xsfp-relaxed` オプションを使用してください。このオプションは、算術浮動小数点演算の順序付けを緩和します。
- 可能であれば、`-Xsfpc` オプションの使用を検討してください。このオプションは、可能な場合は常に中間の浮動小数点丸め操作と変換を排除して、精度を維持するため追加ビットをキャリーします。
- `-Xsno-accessor-aliasing` オプションの利用を検討してください。このオプションは、SYCL* カーネルのアクセサー引数間の依存関係を無視します。

6.4.5 再コンパイル、実行、プロファイル、そして繰り返し

コードを最適化したら、パフォーマンスを測定することが重要です。以下を確認します。

- メトリックは改善されましたか?
- パフォーマンスの目標は達成できましたか?
- 利用可能な計算サイクルが残されていますか?

結果の正当性を確認します。数値結果を比較すると、コンパイラーの最適化やコード変更により異なる場合があります。差異は許容範囲内ですか？そうでなければ、最適化ステップに戻ります。

6.5 oneAPI ライブラリーの互換性

oneAPI アプリケーションには、インテルのツール:のリリースバージョンとの互換性のため、動的ライブラリーが実行時に含まれる場合があります。インテル® oneAPI ツールキットとコンポーネント製品は、互換性を維持するためセマンティック・バージョニング (英語) を使用します。

次のポリシーが、インテル® oneAPI ツールキットで提供される API および ABI に適用されます。

注: oneAPI アプリケーションは、64 ビットのターゲットデバイスでサポートされます。

- 新しいインテル® oneAPI デバイスドライバー、インテル® oneAPI 動的ライブラリー、およびインテル® oneAPI コンパイラーは、インテル® oneAPI ツールを使用してビルドされた展開済みのアプリケーションを破損することはありません。現在の API は、通知とメジャーバージョンの重複なしで削除および変更されることはありません。
- oneAPI アプリケーションの開発者は、ヘッダーファイルとライブラリーのリリースバージョンが同じであることを確認する必要があります。例えば、アプリケーションで、インテル® oneAPI マス・カーネル・ライブラリー (oneMKL) 2021.2 のヘッダーファイルをインテル® oneAPI マス・カーネル・ライブラリー (oneMKL) 2021.1 で使用してはなりません。
- インテル® コンパイラーで提供される新しい動的ライブラリーは、古いバージョンのコンパイラーで作成されたアプリケーションでも動作します (これは一般に下位互換と呼ばれます)。しかし、その逆は当てはまりません。oneAPI 動的ライブラリーの新しいバージョンには、以前のバージョンでは提供されていないルーチンが含まれる場合があります。
- oneAPI 対応のインテル® コンパイラーで提供される古い動的ライブラリーは、新しいバージョンのインテル® oneAPI コンパイラーでは機能しません。

oneAPI アプリケーションの開発者は、oneAPI アプリケーションが互換性のある oneAPI ライブラリーとともに展開されていることを確認するため、完全なアプリケーションのテストを実施する必要があります。

6.6 SYCL* 拡張

SYCL* 拡張機能は、クロスアーキテクチャー・システムを促進する Khronos Group の SYCL* のようなオープンな標準化団体が、迅速に実験、革新、そして開発する好循環を確立できるようにます。インテル® oneAPI DPC++ コンパイラーで動作する拡張機能については、GitHub* の SYCL* 拡張 (英語) を参照してください。

7 用語集

アクセラレーター (Accelerator)

操作のサブセットを迅速に実行する計算リソースを含む専用コンポーネント。CPU、GPU、または FPGA など。

「デバイス」も参照。

アクセサー (Accessor)

アクセスする場所 (ホスト、デバイス) とモード (read、write) 情報を通信します。

アプリケーション・スコープ (Application Scope)

ホスト上で実行されるコード。

バッファー (Buffers)

計算を行うためデバイスに送られる項目の数や型を通信するメモリー・オブジェクト。

コマンド・グループ・スコープ (Command Group Scope)

ホストとデバイス間のインターフェイスとして動作するコード。

コマンドキュー (Command Queue)

コマンドグループを同時に発行します。

計算ユニット (Compute Unit)

処理要素間で使用する共有要素を含み、デバイス上のほかの計算ユニットにあるメモリーよりも高速にアクセスするため、処理要素を「コア」にグループ化したもの。

デバイス (Device)

操作のサブセットを迅速に実行する計算リソースを含むアクセラレーターまたは専用コンポーネント。CPU はデバイスとして利用できますが、その場合、CPU はアクセラレーターとして使用されます。CPU、GPU、または FPGA など。

「アクセラレーター」も参照。

デバイスコード (Device Code)

ホストではなくデバイスで実行されるコード。デバイスコードは、ラムダ式、ファンクター、またはカーネルクラスを介して指定されます。

DPC++

SYCL* サポートを LLVM C++ コンパイラーに追加したオープンソース・プロジェクト。

ファットバイナリー (Fat Binary)

複数デバイス向けのデバイスコードを含むアプリケーション・バイナリー。バイナリーには、汎用コード (SPIR-V* 形式) とターゲット固有の実行可能コードの両方が含まれます。

ファット・ライブラリー (Fat Library)

複数デバイス向けのオブジェクト・コードを含むアーカイブまたはライブラリー。ファット・ライブラリーには、汎用オブジェクト (SPIR-V* 形式) とターゲット固有のオブジェクト・コードの両方が含まれます。

ファット・オブジェクト (Fat Object)

複数デバイス向けのオブジェクト・コードを含むファイル。ファット・オブジェクトには、汎用オブジェクト (SPIR-V* 形式) とターゲット固有のオブジェクト・コードの両方が含まれます。

ホスト (Host)

プログラムの主要部分、具体的にはアプリケーション・スコープとコマンド・グループ・スコープを実行する CPU ベースのシステム (コンピューター)。

ホストコード (Host Code)

ホスト・コンパイラーによってコンパイルされ、デバイスではなくホストで実行されるコード。

イメージ (Images)

組込み関数を介してアクセスされるフォーマット済みのあいまいなメモリー・オブジェクト。通常、RGB などの形式で保存されピクセルで構成される画像に関係します。

カーネルスコープ (Kernel Scope)

デバイス上で実行されるコード。

ND-range

1 次元、2 次元、または 3 次元の N 次元レンジ、カーネル・インスタンスのグループ、またはワーク項目の略。

処理要素 (Processing Element)

計算ユニットを構成する計算向けの個別のエンジン。

単一ソース (Single Source)

ホストとアクセラレーターで実行できる同じファイルのコード。

SPIR-V*

グラフィカル・シェーダー・ステージと計算カーネルを表現するバイナリー中間言語。

SYCL*

同一ソースファイルに含まれるアプリケーションのホストコードとカーネルコードと標準の ISO C++ を使用して、異種プロセッサーのコードを記述できるようにするクロスプラットフォームの抽象化レイヤーの標準化。

ワークグループ (work-group)

計算ユニットで実行するワーク項目の集合。

ワーク項目 (work-item)

oneAPI プログラミング・モデルにおける計算の基本単位。処理要素で実行されるカーネルに関連付けられます。

8 著作権と商標について

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。

絶対的なセキュリティーを提供できるコンピューター・システムはありません。

実際の結果は異なる場合があります。

* 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。

製品および性能に関する情報

性能は、使用状況、構成、その他の要因によって異なります。 詳細については、http://www.intel.com/PerformanceIndex/ (英語) を参照してください。 注意事項の改訂 #20201201