インテル® Xe アーキテクチャー向け インテル® ISPC ユーザーガイド

このドキュメントは ispc GitHub に公開されている『Intel® ISPC for Xe』の日本語参考訳です。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。

インテル® インプリシット SPMD プログラム・コンパイラー (インテル® ISPC) は、最新のインテル® GPU をサポートするため積極的に開発されています。GPU 向けのコンパイルは、ユーザーから見ると非常に簡単ですが、GPU で実行されるコードを管理するのは複雑になることがあります。開発者は、低レベル API である oneAPI レベルゼロ (英語) を介して、利用可能な GPU デバイスや CPU と GPU 間のメモリー転送、コードの実行、および同期を管理できます。もう 1 つの可能性は、インテル® ISPC パッケージに含まれるインテル® ISPC ランタイム (インテル® ISPCRT) (英語) を使用して複雑な実行を管理し、CPU と GPU でタスクを実行する統一された抽象化を作成することです。

目次

インテル® ISPC を使用する

インテル® ISPC からのインテル® Xe ターゲットの出力は、デフォルトでは SPIR-V* ファイルであり、Gen9 またはインテル® Xe ターゲットのどちらかが選択されていると使用されます。

```
ispc foo.ispc --target=gen9-x8 -o foo.spv
```

SPIR-V ファイルは、GPU 向けにコンパイルして実行するためランタイムによって処理されます。

コンパイル時に --emit-zebin フラグを指定して、L0 バイナリーを作成することもできます。現在、SPIR-V 形式の方が安定していますが、L0 バイナリーを試す価値はあります。

環境

インテル® ISPC for Xe は、かなり前から Linux* でサポートされていましたが (推奨される検証済みの Linux* ディストリビューションは Ubuntu* 20.04 です)、v1.16.0 から Windows* でもサポートされています。

第 9 世代インテル® プロセッサー・グラフィックス (Gen9) 以降を搭載するシステムが必要です。

GPU 上でインテル® ISPC プログラムを実行するには、インテル® グラフィックス・コンピュート・ランタイム (英語) とレベルゼロローダー (英語) をインストールする必要があります。

CPU 向けにインテル® ISPC ランタイムを使用するには、システムに OpenMP* ランタイムがインストールされている必要があります。OpenMP* ランタイムのインストール手順については、Linux* ディストリビューションのドキュメントを参照してください。

基本的なコマンドライン・オプション

4 つの新しい GPU サポートが追加されました: gen9-x8、gen9-x16、xelp-x8、および xelp-x16。

-o フラグが指定されると、インテル® ISPC は SPIR-V* 出力ファイルを生成します。オプションとして、--emit-spirv フラグを指定できます。

```
ispc --target=gen9-x8 --emit-spirv foo.ispc -o foo.spv
```

L0 バイナリーを作成するには、--emit-zebin フラグを指定します。L0 バイナリーを使用する場合、ベクトル・バックエンドに追加のオプションを渡すことがあります。これは、--vc-options フラグを使用して指定できます。

また、xe32 と xe64 の新しいアーキテクチャー・オプションが導入されました。xe64 はデフォルトであり、64 ビット・ホストに対応し、64 ビット・ポインターを使用します。xe32 は 32 ビット・ホストに対応し、32 ビット・ポインターを使用します。

LLVM ビットコードを生成するには、--emit-llvm を指定します。LLVM ビットコードをテキスト形式で生成するには、--emit-llvm-text を指定します。

最適化はデフォルトで有効になっていますが、無効にするには -O0 オプションを指定します。

テキスト形式でアセンブリー・ファイルを生成する --emit-asm はサポートされていません。SPIR-V* ファイルからアセンブリー・ファイルを生成するには、「SPIR-V* からアセンブリー・ファイルを生成する方法」を参照してください。

デフォルトは 64 ビット・アドレス指定です。--addressing=32 または --arch=xe32 フラグを使用して 32 ビット・アドレス指定に変更できますが、ポインターサイズはホストコードとデバイスコードで一致させる必要があるため、32 ビット・アドレス指定は 32 ビット・ホスト・システムのプログラムでのみ機能します。

インテル® ISPC ランタイム (インテル® ISPCRT)

インテル® ISPC ランタイム (インテル® ISPCRT) は、CPU ターゲットと GPU ターゲットの実行モデルを統合します。これは、oneAPI レベルゼロ (英語) の上位にある高レベルの抽象化です。このランタイムなしでも CPU でインテル® ISPC を使用でき、代替として GPU には oneAPI レベルゼロを使用します。インテル® ISPCRT を利用したフィードバックお寄せください。インテル® ISPCRT は、ヘッダーファイル (ispcrt.h と ispcrt.hpp を参照) でドキュメント化され、リンク可能なライブラリーとして配布される C および C++ API を提供します。ispc/examples/xpu ディレクトリーにあるサンプルでは、この API を使用した CPU と GPU 向けの SPMD プログラムを実行する方法を紹介しています。sgemm のサンプルで、oneAPI レベルゼロ・ランタイムの使い方を確認できます。同じプロセスから oneAPI レベルゼロを使用して、oneAPI DPC++ コンパイラー向けに記述されたインテル® ISPC カーネルと DPCPP カーネルを実行し、それらの間でデータを共有することもできます。この可能性を詳しく見るには、Simple-DPCPP と Pipeline-DPCPP のサンプルを試してください。ただし、この機能はまだ実験的なものであることにご注意ください。

インテル® ISPCRT オブジェクト

インテル® ISPC ランタイムは、次の抽象化によりコードの実行を管理します。

- **デバイス** - SPMD プログラムを実行可能で、操作可能なメモリーがある CPU または GPU を示します。ユーザーは、特定タイプのデバイス (CPU または GPU) を選択するか、デバイスの選択をランタイムに任せることができます。
- **メモリービュー** - 異なるデバイスからアクセスするデータを示します。例えば、GPU で実行されるコードの入力データは、最初に CPU によってメモリー内に準備され、GPU メモリーに転送されて計算に使用されます。メモリービューは、oneAPI レベルゼロによって提供される統合共有メモリー (USM) のメカニズムによって割り当てられたメモリーを表わすこともできます。USM に割り当てられたデータへのポインターは、ホストとデバイスの両方で利用できます。CPU と GPU 間のデータ移動を明示的に行う必要はありません。これは、oneAPI レベルゼロのランタイムによって自動的に処理されます。
- **タスクキュー** - 各デバイスにはコマンドを実行するタスク (コマンド) キューがあります。実行は非同期であり、前のコマンドの完了を待たずに後続のコマンドを実行できます。実行を同期するために使用できる同期プリミティブが用意されています。
- **バリア** - キューに投入されたすべてのタスクが実行を完了したことを確認するため、タスクキューに挿入できる同期プリミティブです。メモリーコピーとカーネル実行の間にバリアを挿入する必要はありません。カーネル実行の前にコピーされるメモリーは、カーネルが起動される前に完了することが保証されます。これは、バリアよりもきめ細かいメカニズムを使用してインテル® ISPC ランタイムによって実装されるため効率的です。
- **モジュール** - 同時にコンパイルされ、いくつかの共通コードを共有することができるカーネルの集合を表わします。その意味では、インテル® ISPC が生成する SPIR-V* ファイルは、インテル® ISPCRT のモジュールであると考えることができます。ユーザーは、ISPCRTModuleOptions を使用してモジュールをコンパイルする際の追加オプ

ションを指定できます。現在、ISPCRTModuleOptions 構造体では、SPIR-V* のコンパイルに使用される VC バックエンドのスタックサイズを設定できます。サポートされるオプションのセットは、必要に応じて拡張されます。

- **カーネル** - カーネルは、モジュールへのエントリーポイントとなる関数であり、カーネル実行コマンドをタスクキューに投入することで呼び出します。カーネルには、1 つのパラメーター (実際のカーネル・パラメーター構造体へのポインター) があります。
- **フューチャー** - 特定の時点で、このオブジェクトに関連付けられたカーネルの実行が完了し、オブジェクトが有効であることを確実にします。カーネル呼び出しがタスクキューに投入されると、フューチャーが返されます。タスクキューがデバイスで実行されると、フューチャー・オブジェクトが有効になり、カーネルの実行に関連する情報を取得できます。
- **配列** - メモリー・ビュー・オブジェクトをラップし、デバイスまたは USM (統合共有メモリー) のメモリーを簡単に割り当てられるようにします。インテル® ISPCRT では、USM のデータ割り当てを簡単に行うアロケーターのサンプルや、同じ用途で使用できる SharedVector クラスも提供しています。詳細は、XPU のサンプルのドキュメントを参照してください。

すべてのインテル® ISPCRT オブジェクトは参照カウントをサポートしており、細かなメモリー管理を行う必要はありません。オブジェクトは、使用されなくなると解放されます。

実行モデル

インテル® ISPC タスク (英語) の概念は、GPU 上でカーネルの実行をサポートするために拡張されました。タスクキューに投入される各カーネル実行コマンドは、GPU 上で起動するタスク (スレッド) 数がパラメーター化されています。各タスクは、CPU と同様に問題のどの部分を処理するか決定する必要がります。タスク内は、SPMD 方式で実行されます (ここでも通常のインテル® ISPC 実行モデルがコピーされます)。そのため、使用されるすべてのビルトイン変数 (taskIndex、taskCount、programIndex、programCount など) は GPU でも利用できます。

設定

インテル® ISPCRT の動作は、次の環境変数で設定できます。

- **ISPCRT_USE_ZEBIN** - この変数を定義すると、実験的な L0 ネイティブバイナリー形式の使用が強制されます。SPIR-V* ファイルとは異なり、zebin ファイルは異なる GPU タイプ間でポータブルではありません。
- **ISPCRT_IGC_OPTIONS** - インテル® ISPCRT は、インテル® グラフィックス・コンパイラー (IGC) を使用して、GPU で実行されるバイナリーコードを生成します。インテル® ISPCRT では、環境変数 ISPCRT_IGC_OPTIONS にオプションを設定して IGC に渡すことができます。この変数には、+ または = 記号をプリフィクスとして記述できます。+ はインテル® ISPCRT のデフォルトの IGC オプションに環境変数の内容を追加することを意味し、= はインテル® ISPCRT のデフォルト・オプションを環境変数の内容で置き換えることを意味します。
- **ISPCRT_GPU_DEVICE** - 複数の GPU デバイスがシステムに存在する場合、この環境変数を設定してインテル® ISPCRT で使用する GPU デバイスを選択できます。レベルゼロランタイムで列挙されたデバイス番号を設定する必要があります。例えば、2 つの GPU デバイスが存在する場合、この変数には 0 または 1 を設定できます。
- **ISPCRT_MAX_KERNEL_LAUNCHES** - タスクキューにエンキューされるカーネルの起動数の最大値を設定します。制限に達すると、sync() メソッドを呼び出してキューを実行する必要があります。現在、上限値は

100000 に設定されていますが、この環境変数で少ない値に設定できます (テスト用途など)。100000 以上の値は設定できません。それ以上の値を設定すると、インテル® ISPCRT は制限値をデフォルトに戻し、警告メッセージを出力します。

また、ISPCRTModuleOption 構造体を使用して、GPU モジュールにオプションを渡すこともできます。現在、VC バックエンドのスタックサイズを決定するには、stackSize という設定のみがサポートされます。デフォルト値は 8192 です。

インテル® ISPC プログラムのコンパイルと実行

インテル® ISPC がインストールされているディレクトリーの examples/xpu/simple に、インテル® ISPC ランタイムを使用した CPU と GPU ターゲット向けの簡単な C++ プログラムでインテル® ISPC を使用するサンプルが用意されています。ファイ simple.ispc を開いてください (以下に同じコードを示します)。

```
struct Parameters {
    float *vin;
    float *vout;
    int    count;
};

task void simple_ispc(void *uniform _p) {
    Parameters *uniform p = (Parameters * uniform) _p;

    foreach (index = 0 ... p->count) {
        // このプログラム・インスタンスの適切な入力値をロードします
        float v = p->vin[index];

        // 任意の小規模な計算を行うマスが、処理される値に
        // 依存する計算を行います
        if (v < 3.)
            v = v * v;
        else
            v = sqrt(v);

        // 結果を出力配列に書き込みます
        p->vout[index] = v;
    }
}

#include "ispcrt.isph"
DEFINE_CPU_ENTRY_POINT(simple_ispc)
```

examples/simple にある CPU バージョンのサンプルと比較すると、いくつかの違いが分かります。このプログラムで最初に分かるのは、関数定義に export キーワードではなく task キーワードが使用されていることです。これは、この関数がカーネルであるため、ホストから呼び出されることを意味します。

次に注目すべき点は、DEFINE_CPU_ENTRY_POINT です。これは、どの関数が CPU のエントリーポイントであるかをインテル® ISPCRT に伝えます。DEFINE_CPU_ENTRY_POINT の定義を調べると、単純な launch 呼び出しであることが分かります。

```
launch[dim0, dim1, dim2] fcn_name(parameters);
```

これは、ホストコードからシームレスに CPU および GPU ターゲットのスレッド空間を設定するために使用されます。CPU でインテル® ISPCRT を使用しない場合、インテル® ISPC プログラムで DEFINE_CPU_ENTRY_POINT を使用する必要はありません。それ以外は、インテル® ISPCRT から呼び出す関数ごとに DEFINE_CPU_ENTRY_POINT が必要になります。

最後に注意すべきことは、カーネルに実パラメーターを渡す代わりに void* uniform を使用して、後で構造体パラメーターにキャストすることに注目してください。これにより、ホスト側の CPU と GPU でシームレスにカーネルのパラメーターを設定できます。

では、simple.cpp を詳しく見てみましょう。入力パラメーターに応じて、CPU または GPU でインテル® ISPC カーネルを実行します。デバイスタイプは、ISPCRTDeviceType で管理され、ISPCRT_DEVICE_TYPE_CPU、ISPCRT_DEVICE_TYPE_GPU または ISPCRT_DEVICE_TYPE_AUTO (GPU が利用できない場合 CPU にフォールバック) に設定できます。

インテル® ISPCRT ヘッダーをインクルードすることから始めます。

```
#include "ispcrt.hpp"
```

インテル® ISPCRT デバイスを作成します。

```
ispcrt::Device device(device_type)
```

インテル® ISPC カーネルのパラメーターを設定します。

```
// 入力配列を設定します
ispcrt::Array<float> vin_dev(device, vin);

// 出力配列を設定します
ispcrt::Array<float> vout_dev(device, vout);

// Parameters 構造体を設定します
Parameters p;

p.vin = vin_dev.devicePtr();
p.vout = vout_dev.devicePtr();
p.count = SIZE;

auto p_dev = ispcrt::Array<Parameters>(device, p);
```

配列や構造体のような参照タイプは、インテル® ISPC カーネルに適切に渡すため ispcrt::Array にラップされることに注意してください。

実行するモジュールとカーネルを設定します。

```
ispcrt::Module module(device, "xe_simple");
ispcrt::Kernel kernel(device, module, "simple_ispc");
```

モジュール名は、インテル® ISPC から出力される拡張子なしの名前に対応させる必要があります。つまり、この例では、simple.cpp は GPU 用に xe_simple.spv に、CPU 用に libxe_simple.so にコンパイルされるため、モジュール名には xe_simple を使用します。カーネル名は、インテル® ISPC カーネルが必要とするタスク関数名です。

プログラムの残りの部分では、ispcrt::TaskQueue を作成して必要なステップを記述しています。

```
ispcrt::TaskQueue queue(device);

// ISPC カーネルの入力である ispcrt::Array オブジェクトは、
// ホストからデバイスに明示的にコピーします
queue.copyToDevice(p_dev);
queue.copyToDevice(vin_dev);

// 1 スレッドでデバイス上のカーネルを起動します
queue.launch(kernel, p_dev, 1);

// ISPC カーネルの出力である ispcrt::Array オブジェクトは、
// デバイスからホストへ明示的にコピーします。
queue.copyToHost(vout_dev);

// キューの同期を行います
queue.sync();
```

サンプルをビルドして実行するには、examples/xpu フォルダーに移動して build フォルダーを作成します。ビルドフォルダーから、cmake -DISPC_EXECUTABLE=<path_to_ispc_binary> -Dispcrt_DIR=<path_to_ispcrt_cmake> ../ を実行します。または、インテル® ISPC のパスを PATH に追加して、cmake ../ を実行します。Windows* では、-DLEVEL_ZERO_ROOT=<path_lo_level_zero> をシステムの oneAPI レベルゼロへの PATH と一緒に渡す必要があります。make または Visual Studio* ソリューションを使用してサンプルをビルドします。simple フォルダーに移動して生成されたファイルを確認します。

- xe_simple.spv には SPIR-V* 表現が含まれます。このファイルは、インテル® ISPCRT によってインテル® グラフィックス・コンピュート・ランタイムに渡され、GPU で実行されます。
- libxe_simple.so (Linux*)/xe_simple.dll (Windows*) には、さまざまなターゲット向けにインテル® ISPC カーネルから生成されたオブジェクト・ファイルが含まれています (local_ispc サブフォルダーにあります)。このライブラリーは、ホスト・アプリケーションである host_simple からロードされ、CPU での実行に使用されます。
- simple_ispc_<target>.h ファイルには、C 呼び出しが可能な関数宣言が含まれています。実際には使用されず、参照用に作成されています。

- host_simple はメインの実行可能ファイルです。実行ファイル simple を実行すると、次の出力が表示されます。

```
実行： Auto
0: simple(0.000000) = 0.000000
1: simple(1.000000) = 1.000000
2: simple(2.000000) = 4.000000
3: simple(3.000000) = 1.732051
4: simple(4.000000) = 2.000000
...
```

アプリケーションのすべてのコンパイル/リンクコマンドを設定するには、インテル® ISPC 配布パッケージに含まれる CMake モジュールの add_ispc_kernel 関数を使用することを強く推奨します。

インテル® ISPC ビルドシステムから抽出した簡単なサンプルをビルドする完全な CMakeFile.txt は、次のようになります。

```
cmake_minimum_required(VERSION 3.14)
project(simple)
find_package(ispcrt REQUIRED)
add_executable(host_simple simple.cpp)
add_ispc_kernel(xe_simple simple.ispc "")
target_link_libraries(host_simple PRIVATE ispcrt::ispcrt)
```

次のコマンドで設定とビルドを行います。

```
cmake ../ && make
```

別のコンパイルコマンドを実行して、同じ結果を得ることもできます。Linux* 向けのコマンドを以下に示します。

- GPU 向けに ISPC カーネルをコンパイルします。

```
ispc -I /home/ispc_package/include/ispcrt -DISPC_GPU --target=gen9-x8 --woff
-o /home/ispc_package/examples/xpu/simple/xe_simple.spv
/home/ispc_package/examples/xpu/simple/simple.ispc
```

- CPU 向けにインテル® ISPC カーネルをコンパイルします。

```
ispc -I /home/ispc_package/include/ispcrt --arch=x86-64
--target=sse4-i32x4,avx1-i32x8,avx2-i32x8,avx512knl-x16,avx512skx-x16
--woff --pic --opt=disable-assertions
-h /home/ispc_package/examples/xpu/simple/simple_ispc.h
-o /home/ispc_package/examples/xpu/simple/simple.dev.o
/home/ispc_package/examples/xpu/simple/simple.ispc
```

- オブジェクト・ファイルからライブラリーを作成します。

```
/usr/bin/c++ -fPIC -shared -Wl,-soname,libxe_simple.so -o libxe_simple.so
simple.dev*.o
```

- ホストコードをコンパイルおよびリンクします。

```
/usr/bin/c++ -DISPCRT -isystem /home/ispc_package/include/ispcrt -fPIE
-o /home/ispc_package/examples/xpu/simple/host_simple
/home/ispc_package/examples/xpu/simple/simple.cpp -lispcrt -
L/home/ispc_package/lib
-Wl,-rpath,/home/ispc_package/lib
```

デフォルトでは SPIR-V* 形式が使用されます。次のコマンドで L0 バイナリー形式にできます。

```
cd examples/xpu/build
cmake -DISPC_XE_FORMAT=zebin ../ && make
export ISPCRT_USE_ZEBIN=y
cd simple && ./host_simple --gpu
```

言語の制約と既知の問題

次に、インテル® ISPC for Xe の既知の制限事項を示します。

- インテル® Xe プラットフォームでは関数ポインターのサポートに制限があります。プログラムが正しく実行されない可能性があるため、関数ポインターの利用は避けてください。また、関数ポインターを介して呼び出される関数内の print はサポートされません。
- スタック呼び出しのサポートに制限があります。関数は可能な限りインライン展開することを推奨します。
- 浮動小数点演算は、CPU と GPU 間でビット・レベルの再現性が保証されていません。これは、特に数学ライブラリー関数に該当します。アルゴリズムを設計する際は、このことに注意してください。
- 非定数パラメーターを持つ alloca は、まだサポートされていません。
- グローバル変数は「カーネルローカル」です。CPU とは異なり、GPU ではグローバル変数値は複数回カーネルを起動すると、その間保持されません。
- VC バックエンドは依存関係が未解決のモジュールをサポートしないため、SPIR-V* モジュールをインテル® ISPC ランタイムに渡す前に、すべての依存関係を解決する必要があります。

GPU 向けの実装を予定していない機能がいくつかあります。

- カーネルの実行はホストコードで管理されるため、インテル® IPSC プログラムでは launch および sync キーワードはサポートされません。
- インテル® Xe ターゲットのインテル® ISPC プログラムでは、new と delete キーワードはサポートされません。メモリー管理はすべてホスト側で行われることが想定されています。

- インテル® Xe ターゲットでは、export 関数は void を返す必要があります。

パフォーマンス

インテル® ISPC for Xe のパフォーマンスは、このリリースで大幅に改善されましたが、まだ改善の余地があり、時期リリースに向けて改善作業に取り組んでいます。第 9 世代インテル® HD グラフィックスを搭載したインテル® Core™ i9-9900K プロセッサー @ 3.60GHz (最大計算ユニット 24) で実行したマンデルブロの結果を以下に示します。

- @CPU の実行時間: [9.285] ミリ秒
- @GPU の実行時間: [10.886] ミリ秒
- @シリアル実行の時間: [569] ミリ秒

実際のワークロードでは、インテル® ISPC はハードウェアの利用率が高いプログラムを作成する方法を提供しますが、結果として得られるパフォーマンスは適切なデータセットのパーティション化やメモリー管理など、多くの要因に依存します。

GPU プログラミングのパフォーマンス・ガイド

パフォーマンスを向上する GPU プログラミングにはいくつかの規則があります。

レジスター・プレッシャーを軽減

最初に、ローカル変数の数を減らします。すべての変数は GPU のレジスターに保持されますが、変数の数がレジスターの数を超えると、コストの高いレジスタースピルが発生します。

例えば、第 9 世代インテル® グラフィックス (Gen9) のレジスターファイルのサイズは、128x8x32 ビットです。各 32 ビットの可変値は、SIMD-8 では 8x32 ビット、SIMD-16 では 16x32 ビットになります。

ローカル変数の数を減らすには、次の規則に従ってください。

- 可能な限り、varying の代わりに uniform 変数を使用します。この方法は、CPU と GPU の両方に適用できますが、GPU では不可欠です。

```
// 良い例
for(uniform int j=0; j<3; j++) {
    do_something();
}
// 悪い例
for(int j=0; j<3; j++) {
    do_something();
}
```

- 多数のローカル変数を使用するコードの入れ子は避けてください。カーネルを異なる変数スコープを持つステージに分割するのが効果的です。

- 関数から複雑な構造体を返さないようにします。構造体をコピーするのではなく、参照やポインターの利用を検討してください。今後のリリースでは、このような最適化を自動的に行う予定です。

```
// これの代わりに:
struct ExampleStructure
{
  //...
}

ExampleStructure createExampleStructure()
{
  ExampleStructure retVal;
  //... 初期化
  return retVal;
}

int test()
{
  ExampleStructure s;
  s = createExampleStructure();
}
// ポインターの利用を検討
struct ExampleStructure
{
  //...
}

void initExampleStructure(ExampleStructure* init)
{
  //... 初期化
}

int test()
{
  ExampleStructure s;
  initExampleStructure( &s );
}
```

- 再帰を避けてください。
- 利用可能なレジスター数に収まらない場合は、SIMD-8 を使用してください。実行時に次の警告メッセージが表示された場合、コードを SIMD-8 ターゲット (--target=gen9-x8) 用にコンパイルすることを検討してください。

```
Spill memory used = 32 bytes for kernel kernel_name___vyi
```

コード分岐

次の規則は、コードの分岐に関連します。

- 分岐の代わりに選択を使用します。

```
if (x > 0)
  a = x;
else
  a = 7;
// 分岐なしで実装できます
a = (x > 0)? x : 7;
```

- 選択を使用する場合、できるだけ単純にします。

```
// 最適化されていないバ–ション
varying int K;
uniform bool Constant;
...
return bConstant == true ? inParam[0] : InParam[K];
// 最適化されたバ–ション
return InParam[bConstant == true ? 0 : K];
```

- 分岐幅はできるだけ小さくします。一般的な操作は分岐の外に移動する必要があります。大きなコード分岐が必要な場合、アルゴリズムを変更して 1 つのタスクで処理されたデータをグループ化し、分岐内の同じパスを実行することを検討してください。

```
// どちらの分岐も 'array' へのメモリーアクセスを行います。異なるレーン間の分割分岐では、
// 2 つのメモリーアクセス命令が実行されます
if (x > 0)
  a = array[x];
else
  a = array[0];
// 代わりに、共通部分を分岐の外に移動します
int i;
if (x > 0)
  i = x;
else
  i = 0;
a = array[i];
```

- ループでの同様の状況。

```
// 良い例
uniform int j;
foreach (i = 0 ... WIDTH) {
  p->output[i + WIDTH * taskIndex] = 0;
  int temp = p->output[i + WIDTH * taskIndex];
  for (j = 0; j < DEPTH; j++) {
    temp += N;
    temp += M;
  }
  p->output[i + WIDTH * taskIndex] = temp;
}
// 悪い例
foreach (i = 0 ...WIDTH) {
  p->output[i + WIDTH * taskIndex] = 0;
  for (int j = 0; j < DEPTH; j++) {
    p->output[i + WIDTH * taskIndex] += N;
    p->output[i + WIDTH * taskIndex] += M;
  }
}
```

メモリー操作

GPU のメモリー操作には、コストが発生することに注意してください。GPU のカーネルコードは動的メモリー割り当てをサポートしないため、ホストで事前に割り当てた固定サイズのバッファーを使用します。

ギャザー/スキャッター有効など、いくつかの GPU メモリー最適化手法があります。ただし、現在の実装では一部のケースしかカバーしておらず、次のリリースで改善される予定です。

パフォーマンス解析ツール

インテル® GPU でプログラムのパフォーマンスを解析するには、次のツールを利用することを推奨します。

- GTPin (英語) 動的バイナリー・インストルメント・コマンドライン・フレームワークを使用して、インテル® Xe プラットフォームの実行ユニットで動作するコードをプロファイルできます。
- GPU 向けプロファイル・ツール・インターフェイス (英語) は、インテル® ISPC ランタイムのベースであるレベルゼロ呼び出しのパフォーマンスを解析できる ze_tracer を含む、トレースおよび計測ツールです。
- インテル® VTune プロファイラーは、さまざまなハードウェア・ターゲット (CPU、GPU、FPGA) と OS プラットフォーム (Linux*、Windows* など) 向けのパフォーマンス解析ツールです。

これらのツールのほとんどは、インテル® ISPC カーネルの SIMD 幅を 1 としてレポートすることに注意してください。ただし、実際にはインテル® ISPC カーネルは「任意」の SIMD 幅を持つ可能性があります。VC バックエンドは、一部の命令をインテル® ISPC の --target オプションを使用して広い SIMD 幅に最適化できます。

相互運用性

インテル® ISPC は、実験的に Explicit SIMD SYCL* Extension (ESIMD) (英語) との相互運用をサポートしています。

インテル® ISPC カーネルから ESIMD 関数を呼び出したり、その逆を行うこともできます。この機能を試すには、interop.cmake を CMakeLists.txt にインクルードして、add_ispc_kernel と link_ispc_esimd 関数を使用します。simple-esimd のサンプルが参考になります。

よくある質問 (FAQ)

SPIR-V* からアセンブリー・ファイルを生成する

intel-ocloc パッケージに含まれる ocloc ツールを使用します。

```
// 最初にバイナリーを作成
ocloc compile -file file.spv -spirv_input -options "-vc-codegen" -device <name>
// 次にアセンブルを行う
ocloc disasm -file file_Gen9core.bin -device <name> -dump <FOLDER_TO_DUMP>
```

<FOLDER_TO_DUMP> で指定する各カーネルの .asm ファイルを取得できます。

GPU をデバッグする

アプリケーションをデバッグするには、「Linux* ホストでの oneAPI 向け GDB* 導入ガイド」(英語) の説明に従って oneAPI デバッガーを使用してください。現時点で、デバッガーのサポートはかなり制限されていますが、カーネルコードにブレークポイントを設定したり、ステップ実行を行ったり、変数を表示したりできます。

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。