oneAPI

インテル® oneAPI ツールキットを使用した

GPU アクセラレーター向けのチューニング

iSUS 編集長　すがわら　きよふみ

iSUS

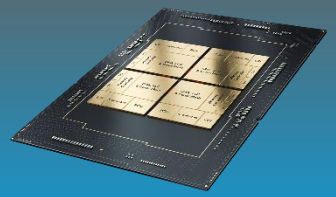
内容

- オフロードカーネルの最適化概要
- GPU 実行モデル概要
- GPU オフロード向けの最適化
- デバイス固有の最適化例
  - インテル® データセンター GPU マックス·シリーズ向けの最適化
  - NVIDIA* と AMD* GPU 向けの最適化
- 環境変数の活用
- NVIDIA* および AMD* GPU プラグインの制限事項と注意事項
- まとめ

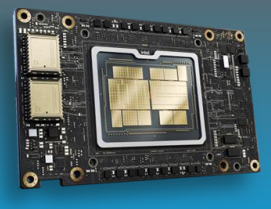
高い処理能力が求められる現代のアプリケーション

今日のパフォーマンス要件を満たすためには多様なアクセラレーターが必要
開発者の 48% が 2 種類以上のプロセッサーやコアを使用する
ヘテロジニアス・システムをターゲットにしている[1]

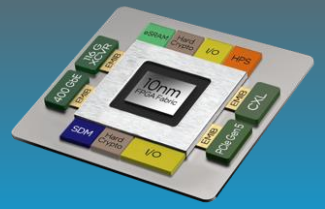
CPU



GPU



FPGA



その他のアクセラレーター

開発者の課題: 複数のアーキテクチャー、ベンダー、プログラミング・モデル

oneAPI

オープンな標準ベースのマルチアーキテクチャー・プログラミング

1. 2022年6月に Evans Data が実施した「Global Development Survey Report 22.1」

オフロードカーネルの最適化概要

オフロードカーネルの最適化

oneAPI は、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が、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

アクセラレーター全体における一般的な最適化

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

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

高レベルの最適化

- 並列ワークの量を増やします
- カーネルのコードサイズを最小化します
- カーネルのロードバランスを取ります
- 高コストの関数は避けてください

ループ関連の最適化

- 適切に構造化/構成された、単純な終了条件のループを使用します
- 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
- 可能な限り深いスコープで変数を宣言します
- 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
- pragma unroll でループをアンロールします

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**
- 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**
- 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**
- **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**
- **カーネルに [[intel::kernel_args_restrict]] 属性を適用することを検討します**

SYCL* 固有の最適化

- 可能であれば、**work-group サイズを指定**します
- 可能であれば、**-Xsfp-relaxed オプションを使用**してください
- 可能であれば、**-Xsfpc オプションの使用**を検討してください
- **-Xsno-accessor-aliasing** オプションの利用を検討してください

Codeplay の推奨

- 共有メモリーの使用には注意してください (malloc_shared)
- work-group サイズを明確にしましょう
- **インデックスの入れ替えをしましょう**
- インライン展開をしましょう
- ループをアンロールしましょう

GPU 実行モデル概要

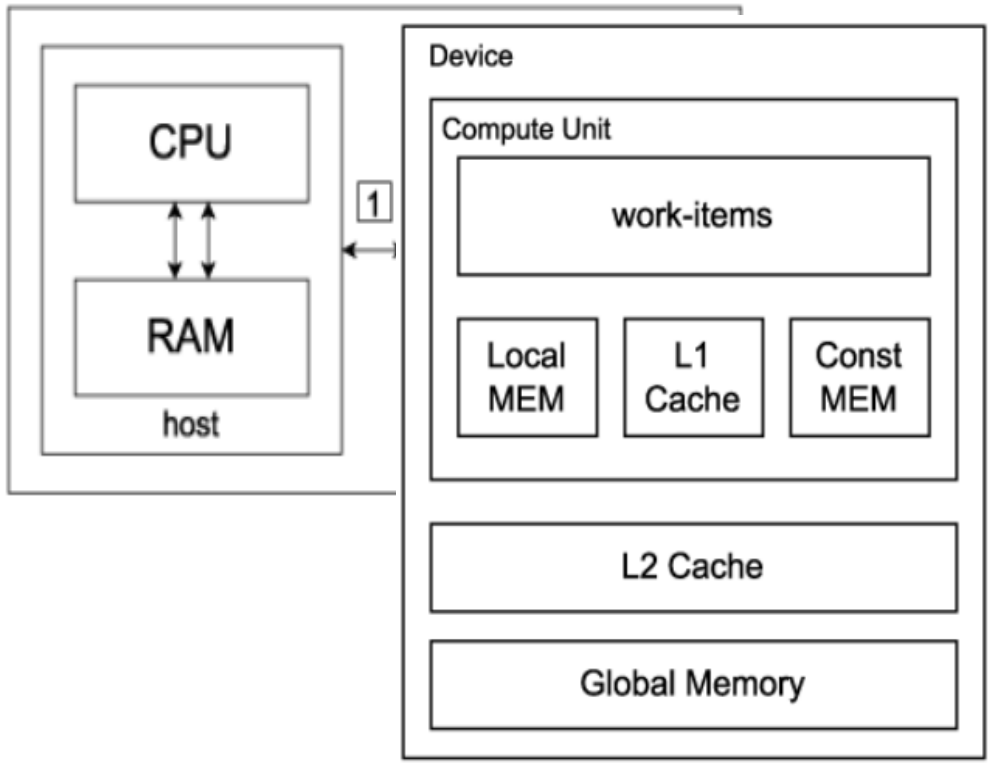
用語と構成

インテル® Iris® Xe GPU アーキテクチャー (第 12.7 世代以降) では、第 9 世代から第 12 世代インテル® Core™ アーキテ クチャーで使用された GPU 用語が変更されています。以下は、新旧の用語の対照表です

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

インテル® Iris® Xe GPU ファミリーのハードウェア構成の詳細については、こちらを参照してください

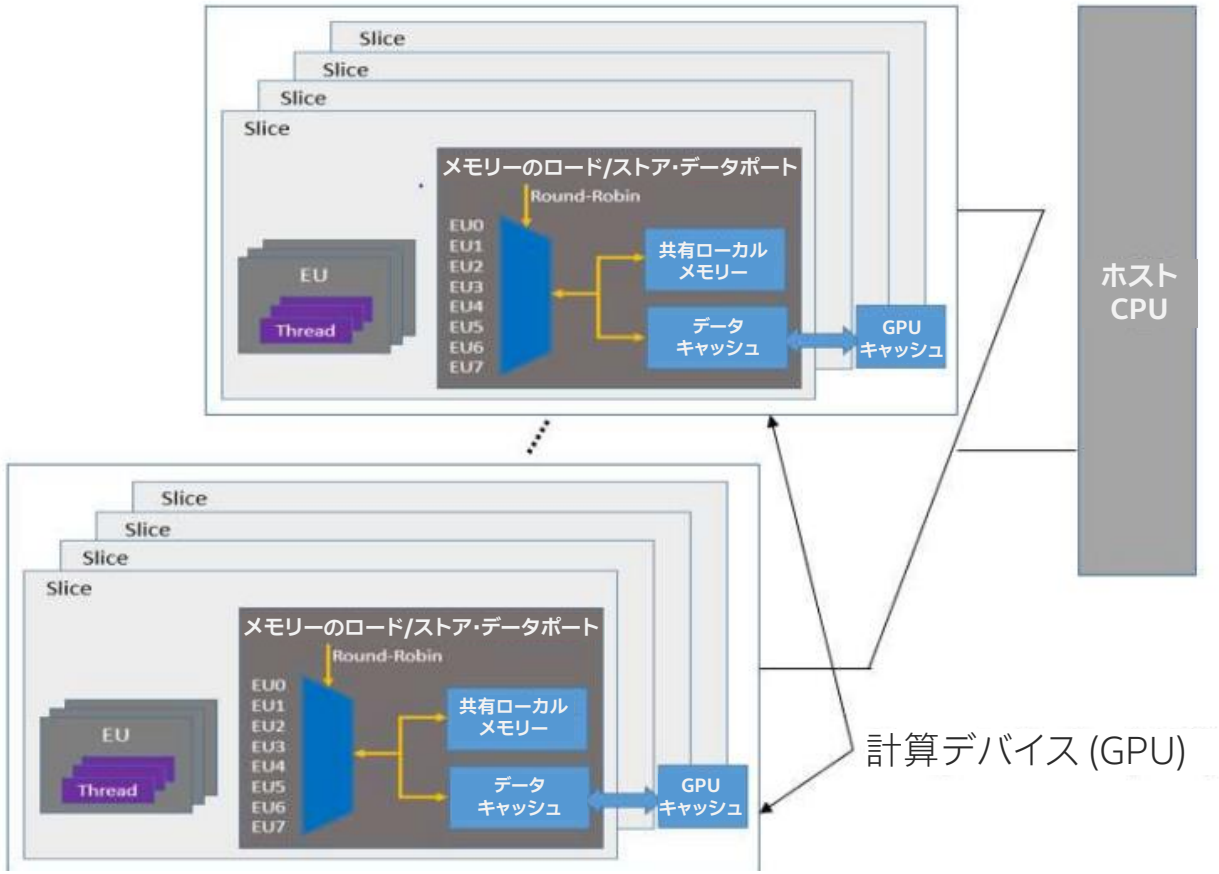
いくつかの用語 …



NVIDIA*/CUDA*	SYCL*
ストリーミング・マルチプロセッサー (SM)	計算ユニット (CU)
ワープ	sub-group
スレッド	work-item
スレッドブロック	work-group
協調スレッドアレイ (CTA)	work-group
共有メモリー	ローカルメモリー
グローバルメモリー	デバイスメモリー
ローカルメモリー	プライベート・メモリー

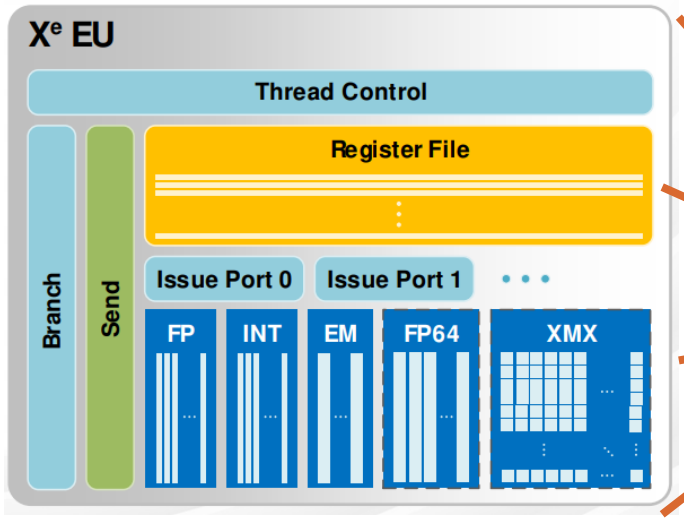
- デバイスメモリー (グローバルメモリー): すべての work-group のすべての work-item 間で共有されます
- ローカルメモリー (共有メモリー): 同一 work-group のすべての work-item 間で共有されます
- プライベート・メモリー (ローカルメモリー): 各 work-item でプライベートです

GPU 実行モデル概要

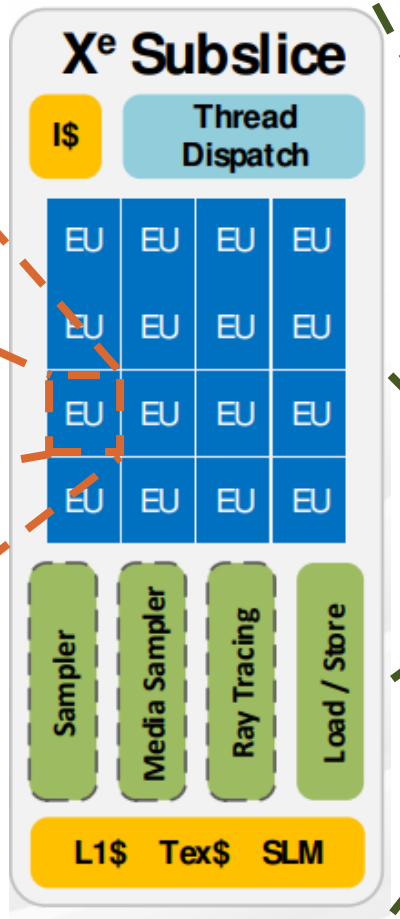


- 汎用 GPU (GPGPU) の計算モデルは、1 つ以上の計算デバイスに接続されたホストで構成されます
- 各計算デバイスは、実行ユニット (EU) または X^e ベクトルエンジン (XVE) と呼ばれる多数の GPU 計算エンジン (CE) で構成されています
- 計算デバイスは図に示すように、キャッシュ、共有ローカルメモリー (SLM)、高帯域幅メモリー (HBM) などを含む 場合があります
- アプリケーションは、ホスト上のソフトウェア (ホスト・フレームグラフごと) と、ホストから送出されるカーネルの組み合わせとして構築され、事前に定義された分岐ポイントで EU (XVE) 上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

GPU アーキテクチャー

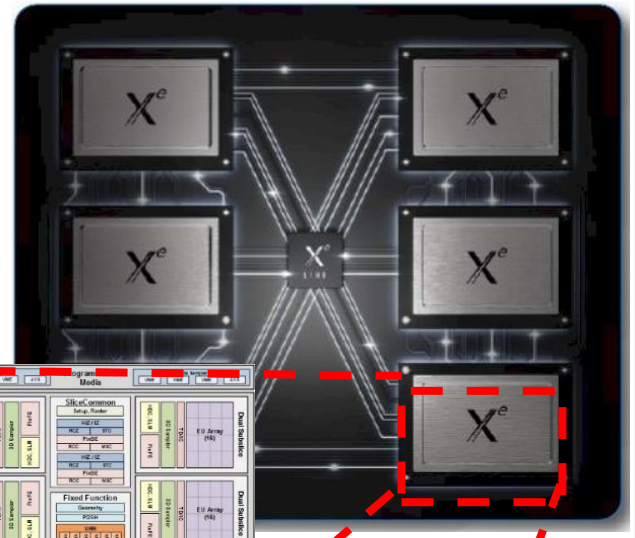


Xe ベクトルエンジン (XVE)



Xe-core (XC)

計算スライス (SLC)



スタック (STK)

SYCL* 実行モデル

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

そして、work-group のスレッドは、sub-group と呼ばれる同じサイズのベクトルのグループに分割されます

- **work-item (スレッド)**: カーネルの並列実行のコレクションの 1 つを表します
- **sub-group (ワープ)**: レングスが 8、16、32 の SIMD ベクトル、またはインテル® UHD グラフィックスを搭載する CPU のネイティブベクトル長の倍数として同時に処理される、短い範囲で連続した work-item を表します
- **work-group (スレッドブロック)**: スレッド階層内の 1、2、または 3 次元のスレッドのセットです。SYCL* における work-item 間の同期は、同一 work-group 内で work-item のバリアがある場合にのみ可能です

nd_range とスレッドの階層

nd_range は、スレッド階層を 1、2、または 3 次元の work-group のグリッドに分割します

これは、各 work-group の ローカルレンジであるグローバルレンジとして表現されます

GPU オフロード向けの最適化

GPU オフロード向けの最適化

- GPU の占有率を最大化
- インデックスの入れ替え
- インライン展開
- ループアンロール
- エイリアス解析
- 浮動小数点計算における精度とパフォーマンス
- メモリーアクセスの最適化

CPU の占有率を最大化 (1)
Xe-core を占有するように work-group をマップ

Xe-LP GPU の場合

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

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

64 x 64 x 64 = 262,144

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

CPU の占有率を最大化 (2)
Xe-core を占有するように work-group をマップ

- カーネルに十分な数のレジスターがある限り、sub-group (SG) のサイズを増やすことができます
- 各 VE には 8 つの 128 ビット SIMD レジスターがあります

最大スレッド数	最小 SG サイズ	最大 SG サイズ	最大 WG サイズ	条件 Xe-LP GPU の場合
64	8	32	512	スレッド数 x SG サイズ <=512

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

デバイスごとのパラメーター確認

Intel(r) oneAPI Tools

```
C> dev
default_selector: Selcted Device: 13th Gen Intel(R) Core(TM) i7
-> Device Vendor: Intel(R) Corporation
-> Number of CU: 24
-> Max work-group size: 8192
-> Max sub-group size: 2048
-> SubGroup size: 4 8 16 32 64
-> Max local mem size: 32 KB
-> Max global mem size: 34 GB
-> Max global mem cache size: 2 MB

C> set ONEAPI_DEVICE_SELECTOR=

C> dev
default_selector: Selcted Device: Intel(R) UHD Graphics 770
-> Device Vendor: Intel(R) Corporation
-> Number of CU: 32
-> Max work-group size: 512
-> Max sub-group size: 64
-> SubGroup size: 8 16 32
-> Max local mem size: 65 KB
-> Max global mem size: 13 GB
-> Max global mem cache size: 1 MB

C> set ONEAPI_DEVICE_SELECTOR="level_zero:1"

C> dev
default_selector: Selcted Device: Intel(R) Arc(TM) A380 Graphics
-> Device Vendor: Intel(R) Corporation
-> Number of CU: 128
-> Max work-group size: 1024
-> Max sub-group size: 128
-> SubGroup size: 8 16 32
-> Max local mem size: 65 KB
-> Max global mem size: 5 GB
-> Max global mem cache size: 4 MB
```

Intel(r) oneAPI Tools

```
C> set SYCL_DEVICE_FILTER=cpu

C> dev
default_selector: Selcted Device: Intel(R) Xeon(R) Platinum 8276 CPU @ 2.20GHz
-> Device Vendor: Intel(R) Corporation
-> Number of CU: 112
-> Max work-group size: 8192
-> Max sub-group size: 2048
-> SubGroup size: 4 8 16 32 64
-> Max local mem size: 32 KB
-> Max global mem size: 137 GB
-> Max global mem cache size: 262 KB

C> set SYCL_DEVICE_FILTER=gpu

C> dev
default_selector: Selcted Device: Intel(R) Arc(TM)
-> Device Vendor: Intel(R) Corporation
-> Number of CU: 512
-> Max work-group size: 1024
-> Max sub-group size: 128
-> SubGroup size: 8 16 32
-> Max local mem size: 65 KB
-> Max global mem size: 13 GB
-> Max global mem cache size: 16777 KB
```

```
kiyo@Xeon-E5:~/デスクトップ/SYCL/wg$ export ONEAPI_DEVICE_SELECTOR="cuda:*"
kiyo@Xeon-E5:~/デスクトップ/SYCL/wg$ dev.exe
default_selector: Selcted Device: NVIDIA GeForce RTX 3060
-> Device Vendor: NVIDIA Corporation
-> Number of CU: 28
-> Max work-group size: 1024
-> Max sub-group size: 32
-> SubGroup size: 32
-> Max local mem size: 49152 B
-> Max global mem size: 8359051264 B
-> Max global mem cache size: 1572864 B
```

kiyo@Xeon-E5: ~/デスクトップ/SYCL/wg

```
 $ export ONEAPI_DEVICE_SELECTOR="opencl:1"
 $ dev.exe
default_selector: Selcted Device: Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz
-> Device Vendor: Intel(R) Corporation
-> Number of CU: 44
-> Max work-group size: 8192
-> Max sub-group size: 2048
-> SubGroup size: 4 8 16 32 64
-> Max local mem size: 32768 B
-> Max global mem size: 67402444800 B
-> Max global mem cache size: 262144 B
 $ export ONEAPI_DEVICE_SELECTOR="cuda:*"
 $ dev.exe
default_selector: Selcted Device: NVIDIA GeForce GTX 1060 6GB
-> Device Vendor: NVIDIA Corporation
-> Number of CU: 10
-> Max work-group size: 1024
-> Max sub-group size: 32
-> SubGroup size: 32
-> Max local mem size: 49152 B
-> Max global mem size: 6365839360 B
-> Max global mem cache size: 1572864 B
 $ vi dev.cpp
 $
```

コードサンプル

```
…
std::cout << "-> Number of CU: " <<
dev.get_info<info::device::max_compute_units>() << "¥n";

std::cout << "-> Max work-group size: " <<
dev.get_info<sycl::info::device::max_work_group_size>() << "¥n";

std::cout << "-> Max sub-group size: " <<
dev.get_info<sycl::info::device::max_num_sub_groups>() << "¥n";

std::cout << "-> SubGroup size: " ;
for (const auto &s : dev.get_info<sycl::info::device::sub_group_sizes>()){
   std::cout << s << " ";
}
std::cout << std::endl;
…
```

CPU の占有率を最大化 (3)
Xe-core を占有するように work-group をマップ

- SYCL* work-group は通常、Xe-core にディスパッチされ、work-group 内のすべての work-item は、work-group 内のスレッドバリアとメモリーフェンスによる同期のため、Xe-core の同じ SLM を共有します
  - VE ALU、SLM、およびスレッド・コンテキストが十分であれば、複数の work-group を同じ Xe-core にディスパッチできます
- 利用可能なすべての Xe-core を完全に利用することで、高いパフォーマンスを得ることができます
  - カーネルの GPU 占有率に影響するパラメーターは、work-group サイズと SIMD sub-group サイズであり、これは work-group 内のスレッド数を決定します

簡単なカーネルの例 (同期あり)

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

- このカーネルのローカルレンジは、range(1, R, 128) として与えられます
- sub-group サイズ が 8 であると仮定すると、変数 R の値が VE の占有率にどのように影響するか確認してみます
- R=1 の場合、ロー カルグループのレンジは (1, 1, 128) で、work-group のサイズは 128 です

sub-group サイズ変更の影響

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

- R>4 の場合、work-group のサイズはシステムでサポートされる最大 work-group サイズ 512 を超えるため、カーネルの起動に失敗します
- R=4 の場合、Xe-core は 57% しか占有されておらず (4/7)、3 つの未使用スレッド・コンテキストは別の work-group に対応するには十分でなく、利用可能な VE の 43% が浪費されています
- ドライバーは、未使用の Xe-core に部分的な work-group をディスパッチできる可能性があることに注意してください

明示的に制御する例

work-group サイズ (wg_size) 512 と可変数の work-group 数 (num_groups)、および SIMD 幅 32 (intel::reqd_sub_group_size(32)) を明示的に指示する例

```
…
size_t num_groups = groups;
size_t wg_size = 512;
…
h.parallel_for(
        sycl::nd_range<1>(num_groups * wg_size, wg_size),
        [=](sycl::nd_item<1> index) [[intel::reqd_sub_group_size(32)]] {
          size_t grp_id = index.get_group()[0];
          size_t loc_id = index.get_local_id();
          size_t start = grp_id * mysize;
          ...
 });
```

実際に占有率が変化したことをインテル® VTune™ プロファイラーの「EU スレッド占有率」で確認できます

GPU オフロード向けの最適化

- GPU の占有率を最大化
- **インデックスの入れ替え**
- インライン展開
- ループアンロール
- エイリアス解析
- 浮動小数点計算における精度とパフォーマンス
- メモリーアクセスの最適化

インデックスの入れ替え

- SYCL* 2020 仕様では、「**整数から多次元 id やレンジを構成する場合、多次元空間の線形化において右端の要素が最も速く変化するように要素を記述します**」と定義されています
- インテル® DPC++ コンパイラーでは、右端の次元が GPU (SYCL*、CUDA*、HIP) の x 次元にマップされ、右から 2 つ目の次元が GPU の y 次元にマップされます

```
cgh.parallel_for(sycl::nd_range{sycl::range(WG_X), sycl::range(WI_X)}, ...)

cgh.parallel_for(sycl::nd_range<2>{sycl::range<2>(WG_Y, WG_X),
sycl::range<2>(WI_Y, WI_X)}, ...)

cgh.parallel_for(sycl::nd_range<3>{sycl::range<3>(WG_Z, WG_Y, WG_X),
sycl::range<3>(WI_Z, WI_Y, WI_X)}, ...)
```

2 または 3 次元のカーネルの parallel_for 実行での留意事項

```
cgh.parallel_for(sycl::nd_range{sycl::range(WG_X), sycl::range(WI_X)}, ...)

cgh.parallel_for(sycl::nd_range<2>{sycl::range<2>(WG_Y, WG_X),
sycl::range<2>(WI_Y, WI_X)}, ...)

cgh.parallel_for(sycl::nd_range<3>{sycl::range<3>(WG_Z, WG_Y, WG_X),
sycl::range<3>(WI_Z, WI_Y, WI_X)}, ...)
```

- ローカルまたはグローバルメモリー内の (1-d) 配列は、線形アクセスされます
- 結合されていないグローバル・メモリー・アクセスまたはローカルメモリー内のバンク競合によるパフォーマンスの問題を回避するため、これを考慮する必要があります
- 線形化の詳細については、SYCL* 仕様の 3.11 節の多次元オブジェクトと線形化を参照してください

線形アクセスの問題

- 次のエラー (または同等のエラー) が発生します:

  Number of work-groups exceed limit for dimension 1 : 379957 > 65535

- これは、CUDA* など一部のプラットフォームでは、x 次元が y および z 次元よりも多くの work-group をサポートするためです:

  Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)

ComputeCpp v1.1.6: Changes to Work-item Mapping Optimization - Codeplay Software Ltd (英語)

GPU オフロード向けの最適化

- GPU の占有率を最大化
- インデックスの入れ替え
- **インライン展開**
- **ループアンロール**
- **エイリアス解析**
- 浮動小数点計算における精度とパフォーマンス
- メモリーアクセスの最適化

インライン展開

- カーネルラムダがカーネル実装を含む大きな関数を単純に呼び出すような SYCL* アプリケーションでは、DPC++ がインライン展開に対し保守的になり、パフォーマンスが低下することもあります

```
__attribute__((always_inline)) void function(...) {
  ...
}

...

q.submit([&](sycl::handler &cgh) {
    cgh.parallel_for(..., [=](...) {
      function(...);
    });
}
```

ループのアンロール

- コンパイラーは一部のループアンロールを自動的に行いますが、unrolling プラグマを使用して、デバイスコード内の計算集約型ループのアンロールを手動でコンパイラーに指示することが有益な場合もあります

```
#pragma unroll <unroll factor>

for( ... ) {

  ...

}
```

エイリアス解析

- エイリアス解析では、2 つのメモリー参照が互いにエイリアスでないことが証明でき、最適化が有効になることがあります
- デフォルトでは、コンパイラーはエイリアス解析によって証明されない限り、メモリー参照はエイリアスであると想定する必要があります
- デバイスコード内のメモリー参照がエイリアスではないことをコンパイラーに明示的に通知します
  - oneapi 拡張の no_alias プロパティーをアクセサーに
  - __restrict__ 修飾子をポインターに追加
  - intel::kernel_args_restrict 属性をカーネルで宣言

no_alias プロパティーをアクセサーに追加

```
q.submit([&](sycl::handler &cgh) {

  sycl::accessor acc{...,
  sycl::ext::oneapi::accessor_property_list
  {sycl::ext::oneapi::no_alias}};

  ...

});
```

__restrict__ 修飾子をポインターに追加

- __restrict__ は C++ では非標準であり、SYCL* 実装全体で一貫性がない可能性があることに注意してください

```
void function(int __restrict__ *ptr) {
    ...
}

...
int *ptr = sycl::malloc_device<int>(..., q);
...
q.submit([&](sycl::handler &cgh) {
      cgh.parallel_for(..., [=](...) {
        function(ptr);
      });
});
```

DPC++ では、restrict 修飾されたデバイス関数 (SYCL* カーネルから呼び出される関数) パラメーターのみが考慮されます

intel::kernel_args_restrict 属性を使用する

- これは、各 USM ポインター間、またはそのモデルがカーネル内で使用される場合はバッファーアクセサー間のすべてのエイリアス依存関係を無視するようにコンパイラーに指示します

```
q.submit([&](handler& cgh) {
    accessor in_accessor(in_buf, cgh, read_only);
    accessor out_accessor(out_buf, cgh, write_only);
      cgh.single_task<NoAliases>([=]() [[intel::kernel_args_restrict]] {
        for (int i = 0; i < N; i++)
            out_accessor[i] = in_accessor[i];
      });
});
```

GPU オフロード向けの最適化

- GPU の占有率を最大化
- インデックスの入れ替え
- インライン展開
- ループアンロール
- エイリアス解析
- **浮動小数点計算における精度とパフォーマンス**
- メモリーアクセスの最適化

浮動小数点計算における精度とパフォーマンスのトレードオフ

- 浮動小数点を使用するアプリケーションの開発者は、通常、次の 2 つのことを意識します
  - **精度:** 正確な計算結果に「近い」結果を生成します。
  - **パフォーマンス:** 可能な限り高速に実行できるアプリケーションを作成します

通常、これら 2 つの目的は相反します。しかし、適切なプログラミング手法と適切なコンパイラー・オプションを使用することで、そのトレードオフを制御できます

SYCL* 仕様では、数学関数のサブセットのネイティブバージョンが提供されています

カーネルで使用する数学関数

- SYCL* では、呼び出す数学関数を選択することで、ソースレベルで浮動小数点セマンティクスを制御できます

  - **std::log**: C++ 標準ライブラリーの log 関数を参照します。実装の選択は、コンパイルオプション (-fp-model と -cl-fast-relaxed-math) の規定によって決定されます。例えば、ネイティブ数学命令を使用する実装を選択するには、-cl-fast-relaxed-math オプションを指定してコンパイルします。

  - **sycl::log**: SYCL* が提供する sycl 名前空間の log 関数を参照します。-cl-fast-relaxed-math オプションが指定されていなくても、この関数はネイティブ命令を使用する場合があります

  - **sycl::native::log**: SYCL* が提供する sycl 名前空間のネイティブ log 関数を参照します。この関数はネイティブ命令を使用するため、-cl-fast-relaxed-math オプションは必要ありません。SYCL* (およびインテル® GPU) は、単精度 (float、real) のネイティブ命令のみをサポートします

SYCL* におけるパフォーマンスと精度

浮動小数点のセマンティクス	ホストとデバイスのコンパイルに適用	ホストのコンパイルにのみ適用	デバイスのコンパイルにのみ適用
precise	-fp-model=precise	-fp-model=precise と -Xsycl-target-frontend "-fp-mode=fast" を指定	-Xsycl-target-frontend "-fp-model=precise"
fast-math	-fp-model=fast-math (デフォルト)	-fp-model=fast と -Xsycl-target-frontend "-fp-model=precise" を指定	-fp-model=precise と -Xsycl-target-frontend "-fp-model=fast" を指定
relaxed-math (ネイティブ命令)	デバイスにのみ適用	デバイスにのみ適用	-Xsycl-target-backend "-options -cl-fast-relaxed-math"

- 矛盾するオプションを指定しない。結果は予測できません
- ホストとデバイスで最も一般的なオプションは、-fp-model=fast です
- デバイス上で最良のパフォーマンスを得るには relaxed-math を使用します
- 最も高い精度が必要な場合 -fp-model=precise を使用します

GPU オフロード向けの最適化

- GPU の占有率を最大化
- インデックスの入れ替え
- インライン展開
- ループアンロール
- エイリアス解析
- 浮動小数点計算における精度とパフォーマンス
- **メモリーアクセスの最適化**

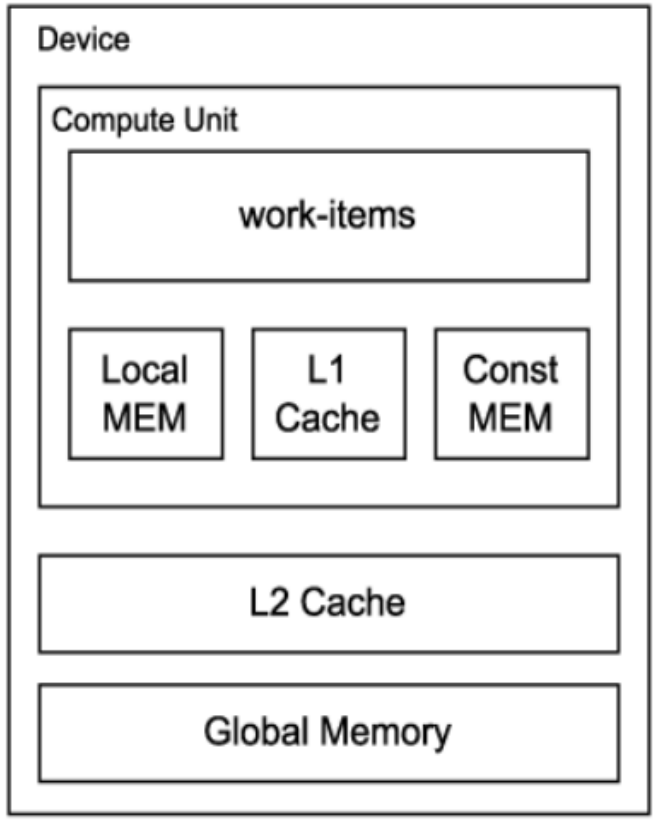
メモリーアクセスと階層

デバイスのメモリーにはいくつかの種類と階層があります:

デバイスメモリー、プライベート・メモリー、ローカルメモリー

デバイスにはキャッシュや高帯域幅メモリーがあります:

SLM、HBM、L1/L2 キャッシュ



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

メモリーアクセスの最適化

- ホストとアクセラレーター間のメモリー移動の最適化
- ホストとデバイス間でデータ移動の往復を避ける
- ループ内でのバッファー宣言を避ける
- バッファー・アクセサー・モードを正しく設定する
- ホストとデバイス間の帯域幅を理解する
- デバイスでの計算とホスト/デバイス間のオーバーラップ・データ転送

use_host_ptr によるメモリーアクセスの最適化の例

```
int VectorAdd0(sycl::queue &q, AlignedVector<int> &a, AlignedVector<int> &b,
               AlignedVector<int> &sum, int iter) {
  sycl::range num_items{a.size()};

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

  for (int i = 0; i < iter; i++) {
    sycl::buffer a_buf(a, props);
    sycl::buffer b_buf(b, props);
    sycl::buffer sum_buf(sum.data(), num_items, props);
    {
      sycl::host_accessor a_host_acc(a_buf);
      std::cout << "add0: buff memory address =" << a_host_acc.get_pointer()
                << "\n";
      std::cout << "add0: address of vector a = " << a.data() << "\n";
    }
    q.submit([&](auto &h) {
      // 入力アクセサー
      sycl::accessor a_acc(a_buf, h, sycl::read_only);
      sycl::accessor b_acc(b_buf, h, sycl::read_only);
      // 出力アクセサー
      sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
      sycl::stream out(1024 * 1024, 1 * 128, h);

      h.parallel_for(num_items, [=](auto i) {
        if (i[0] == 0)
          out << "add0:  dev addr = " << a_acc.get_pointer() << "\n";
        sum_acc[i] = a_acc[i] + b_acc[i];
      });
    });
  }
  q.wait();
  return (0);
}
```

引数が const として宣言されていないことに注意してください。const にするとバッファーが作成されると、新しいメモリーが確保されコピーが行われます

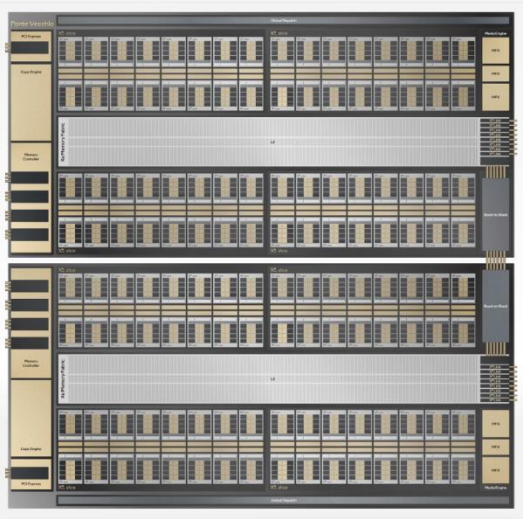
バッファーに use_host_ptr プロパティーを設定することで、可能であればホストメモリーをコピーすることなく、バッファーを直接参照できます

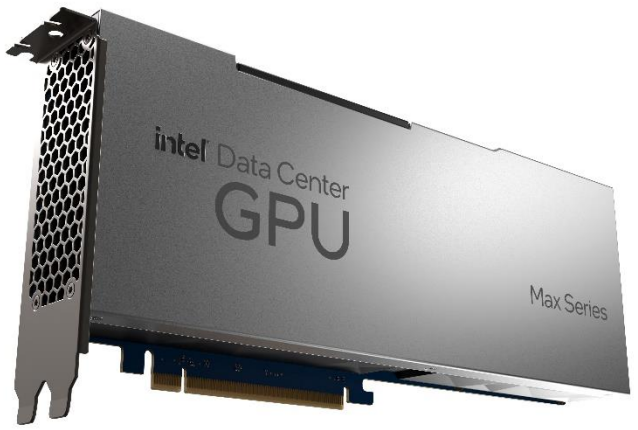
統合 GPU で実行すると、ホスト、バッファー、アクセラレーターで同じアドレスが使用されます。dGPU ではバッファーとデバイスのアドレスは異なります

デバイス固有の最適化例

インテル® データセンター GPU マックス·シリーズ固有の最適化

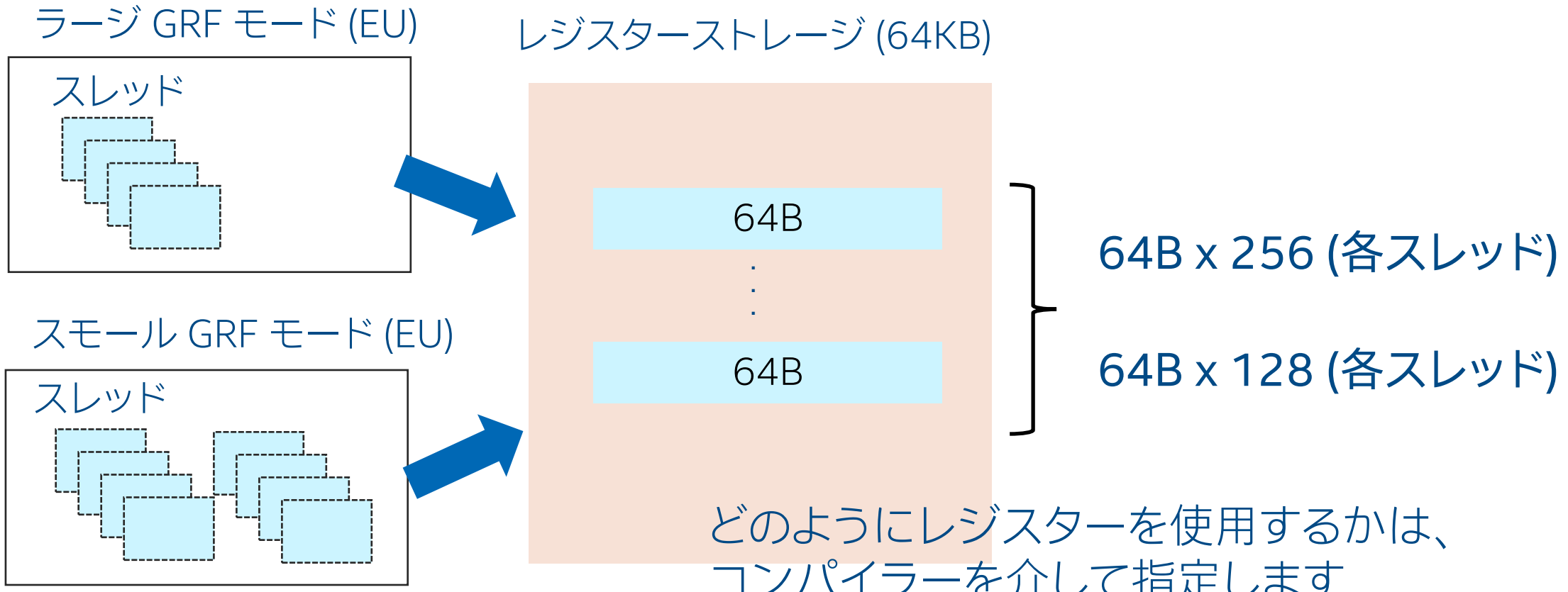
- 2 つの汎用レジスターモード
- マルチスタック
- 暗黙のスケーリング·モードと
  明示的なスケーリング·モード
- インテル® MPI ライブラリーとの
  併用のヒント





2 つの汎用レジスターモード

インテル® データセンター GPU マックス・シリーズでは、2 つの汎用レジスター (GRF) モード、スモール GRF モードとラージ GRF モードを利用できます



GRF モードの指定 (コンパイルオプション)

GRF モードの選択に利用できるバックエンド・オプションをインテル® グラフィックス・コンパイラー (IGC) に渡します

- **-ze-opt-large-register-file:**
  - IGC はすべてにカーネルに対しラージ GRF モードを選択します
- **-ze-opt-intel-enable-auto-large-GRF-mode:**
  - IGC はヒューリスティックに従ってカーネルごとにスモール/ラージ GRF モードを選択できるようになります
- **デフォルト:**
  - IGC はすべてのカーネルに対しスモール GRF モードを選択します

```
-Xsycl-target-backend "-options -ze-opt-large-register-file"
```

GRF モードの指定 (ソースコード)

GRF モードの選択に利用できるバックエンド・オプションをインテル® グラフィックス・コンパイラー (IGC) に渡します

1. ラージ GRF モードを使用するには、次のヘッダーファイルをインクルードします

```
#include <sycl/ext/intel/experimental/kernel_properties.hpp>
```

2. 次に、次の API をカーネルの関数呼び出しツリー内に追加します

```
set_kernel_properties(kernel_properties::use_large_grf);
```

```
cgh.parallel_for<class FillBuffer>(
 NumOfWorkItems, [=](sycl::id<1> WIid) {
   set_kernel_properties(kernel_properties::use_large_grf);
   // インデックスでバッファーを埋めます
   Accessor[WIid] = (sycl::cl_int)WIid.get(0);
});
```

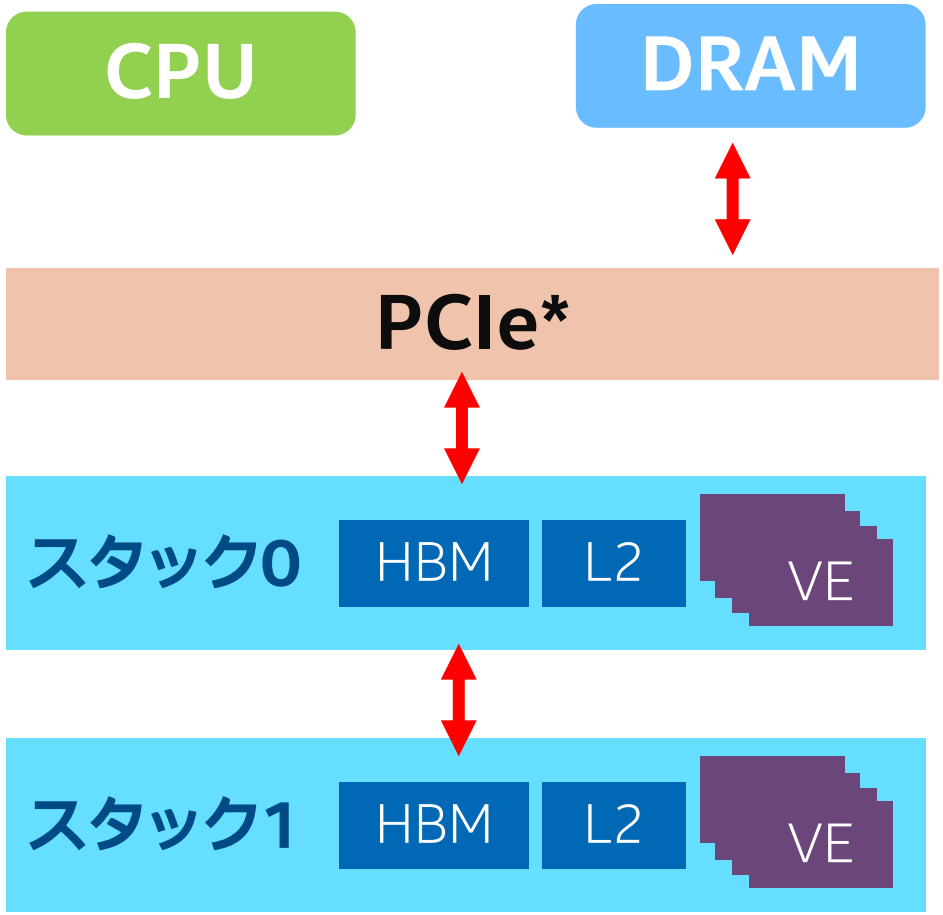
GRF 最適化の指標

- アンロールを無効にすると、スモール GRF モードでパフォーマンスが向上する傾向があります。これは、ラージ GRF モードのメリットを得るほど、レジスター・プレッシャーが高くないことを意味します

- アンロールを有効にすると、ラージ GRF モードでパフォーマンスが向上する傾向があります。レジスター・プレッシャーが高く、ラージ GRF モードがプレッシャーの軽減に有効であることを意味します

- スモール GRF モードを使用すると、チーム数が増加し、ワークロードが大きくなる (プレッシャーが高まる) 場合、パフォーマンスが向上する傾向があります

インテル® データセンター GPU マックス・シリーズのマルチスタック



それぞれのスタックには専用のリソースがあります

ベクトルエンジン (VE) :

計算ユニットはスタックに属します

高帯域幅メモリー (HBM) :

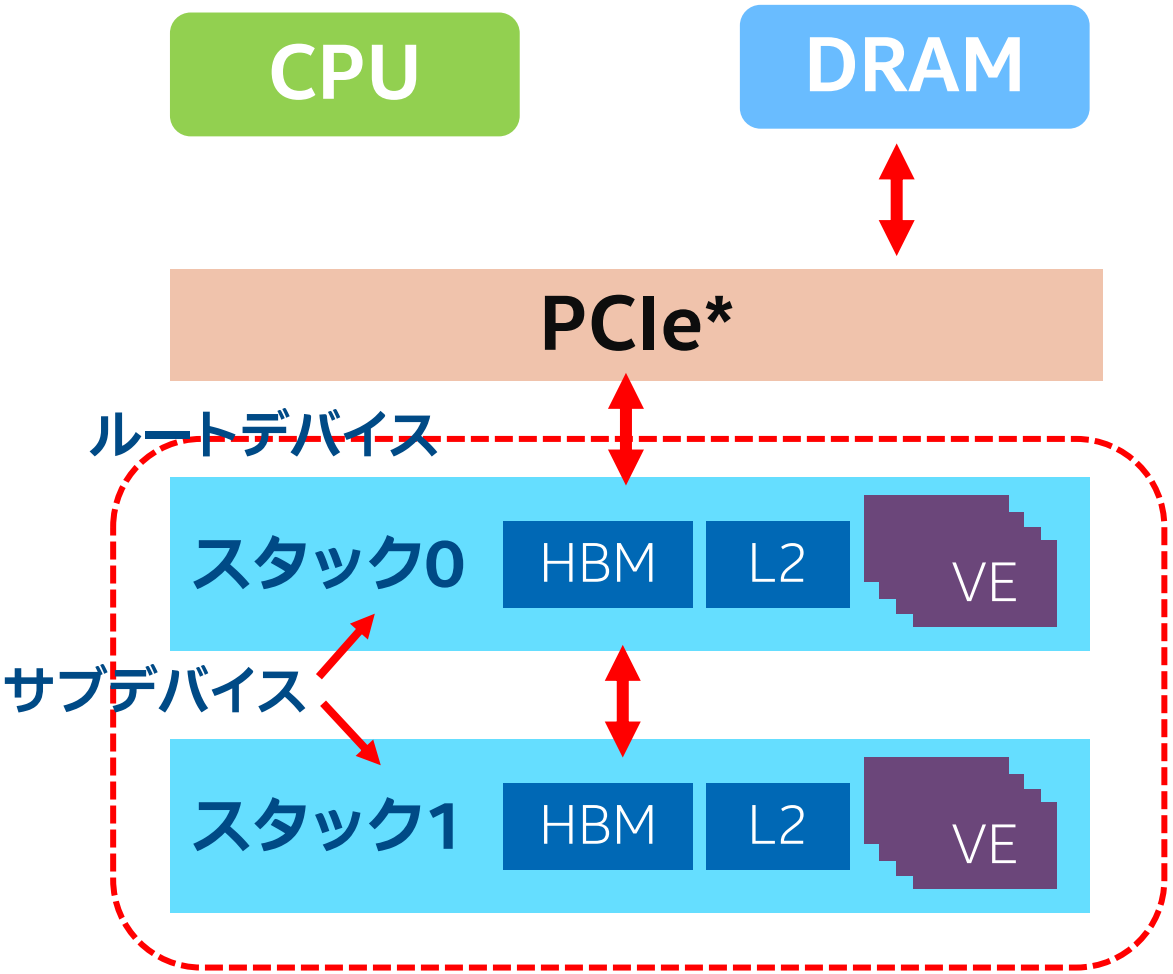
スタックに直接接続される HBM

レベル 2 キャッシュ (L2) :

レベル 2 キャッシュはスタックに属します

暗黙のスケーリング・モードと明示的なスケーリング・モード

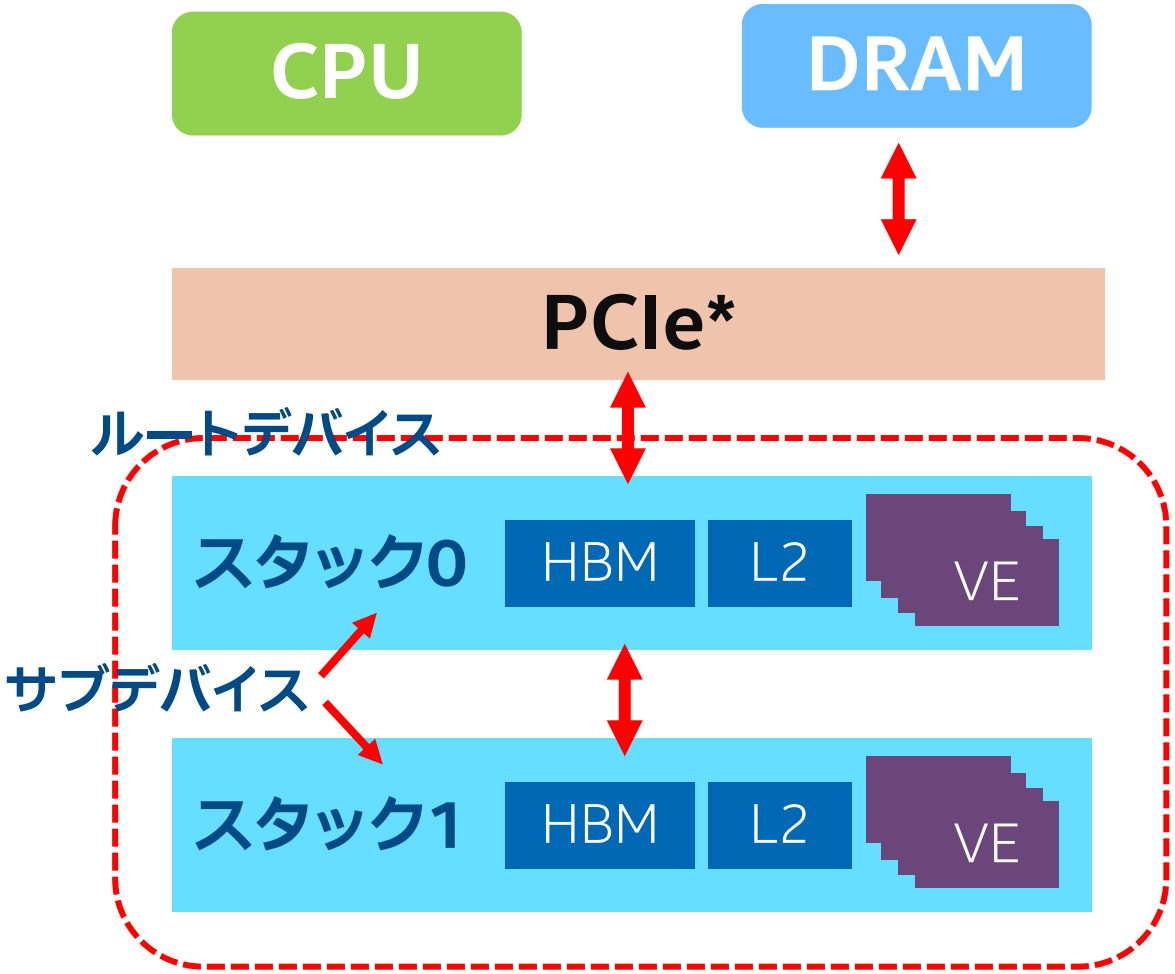
暗黙のスケーリングと明示的なスケーリング



- ルートデバイスは、スタックとも呼ばれる複数のサブデバイスで構成されます
- スタックは、明示的なスタック間の通信を必要としない、ルートデバイスをモノリシック・デバイスとして扱うことを可能にする共有メモリー空間を形成します
- 暗黙のスケーリングでは、ルートデバイスのドライバーは、アプリケーション・コードがカーネルを起動する際にすべてのスタックにワークを分散する役割を持ちます

環境変数 EnableImplicitScaling=1 に設定 (デフォルト)

暗黙のスケーリングと明示的なスケーリング



- 明示的なスケーリングでは、プログラマーが work-group の分散とメモリー配置を直接制御します
- sycl-ls コマンドを使用してルートデバイスを検出します
- 環境変数 **CreateMultipleRootDevices=N** と **NEOReadDebugKeys=1** を設定してサブデバイスを確認します

```
try {
vector<device> SubDevices = RootDevice.create_sub_devices<
cl::sycl::info::partition_property::partition_by_affinity_domain>(
cl::sycl::info::partition_affinity_domain::numa);
}
```

ルートデバイスの確認と設定

```
$ export ONEAPI_DEVICE_SELECTOR=level_zero:*
$ sycl-ls

[level_zero:0] GPU : Intel(R) Level-Zero 1.1 [1.1.20776]
```

```
$ export CreateMultipleRootDevices=2
$ export NEOReadDebugKeys=1
$ export ONEAPI_DEVICE_SELECTOR=level_zero:*
$ sycl-ls

[level_zero:0] GPU : Intel(R) Level-Zero 1.1 [1.1.20776]
[level_zero:1] GPU : Intel(R) Level-Zero 1.1 [1.1.20776]
```

明示的なスケーリング・モードの注意点

マルチスタック GPU のパフォーマンス・チューニングは、並列化の粒度が細かくなるため、面倒な作業が発生します。しかし、基本は CPU のパフォーマンス・チューニングと同様です

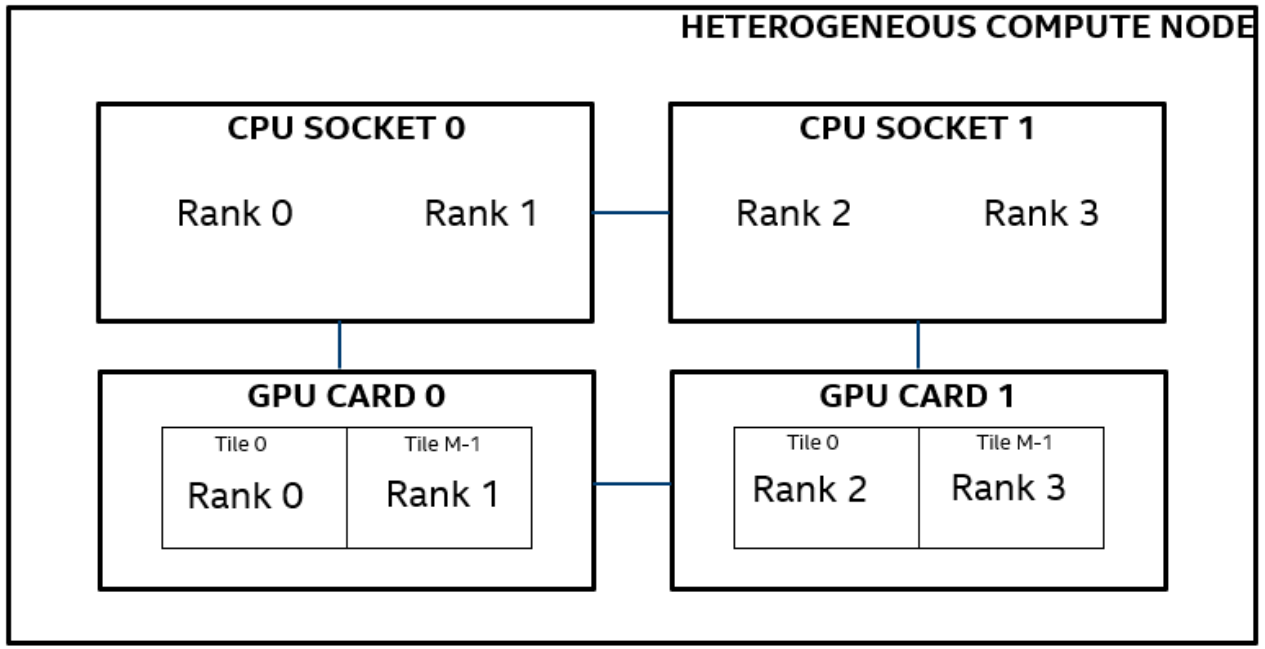
パフォーマンス・スケーリングを理解するには、次の点に注意してください:

- VE 利用効率: カーネルが異なるスタックの実行リソースをどのように利用するか
- データ配置: 割り当てが異なるスタックの HBM にどのように分散されるか
- スレッドデータのアフィニティー: データがどこに配置され、システム内でどのようにアクセスされるか

このプレゼンテーションでは、詳細については触れません。詳しくは、『oneAPI GPU 最適化ガイド 2023.1』をご覧ください

インテル® MPI ライブラリーとの併用のヒント

最新のヘテロジニアス (CPU + GPU) クラスターには、さまざまな GPU 固有のスケーリングの可能性があります



1. スタック・スケーリング: 同一 GPU カード内のスタック間でのスケーリング (GPU 内、スタック間)
2. スケールアップ: 同じノードに接続された複数の GPU カード間でのスケーリング (ノード内、GPU 間)
3. スケールアウト: 異なるノード間に接続された GPU カード間のスケーリング (GPU 間、ノード間)

これらのシナリオはすべてインテル® MPI ライブラリーによって透過的に処理されるため、アプリケーション開発者はソースコードを変更する必要はありません

NVIDIA* と AMD* GPU 向けの最適化

- NVIDIA* GPU 向けにビルドする場合、GPU のアーキテクチャーは省略できます。この場合、デフォルトのアーキテクチャーは sm_50 になります
- AMD* GPU 向けのデフォルトはありません。アーキテクチャーの指定は必須です
- 新しい世代の GPU を使用する場合、適切なアーキテクチャーを指定することでコンパイラーがより高性能な新しい機能を使用できる可能性があります

```
- -Xsycl-target-backend=amdgcn-amd-amdhsa --offload-arch=gfx1030
- -Xsycl-target-backend=nvptx64-nvidia-cuda --offload-arch=sm_80
```

環境変数の活用

環境変数の活用

- デバイス選択
  - SYCL_DEVICE_FILTER、ONEAPI_DEVICE_SELECTOR …
- JIT コンパイルの制御
  - SYCL_CACHE_*、SYCL_ENABLED_FUSION_CACHING …
- CUDA* と HIP プラグインの制御
  - SYCL_PI_CUDA_MAX_LOCAL_MEM_SIZE、SYCL_PI_HIP_MAX_LOCAL_MEM_SIZE

https://www.isus.jp/products/oneapi/llvm-sycl-environment-variables/

デバイス選択

- 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

  例: `set ONEAPI_DEVICE_SELECTOR="cuda:*"`
  `export ONEAPI_DEVICE_SELECTOR="opencl:1"`

```
$ sycl-ls
[opencl:acc:0] Intel(R) FPGA Emulation Platform for OpenCL(TM), Intel(R) FPGA Emulation Device 1.2 [2023.15.3.0.20_160000]
[opencl:cpu:1] Intel(R) OpenCL, Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz 3.0 [2023.15.3.0.20_160000]
[opencl:gpu:2] Intel(R) OpenCL HD Graphics, Intel(R) Arc(TM) A380 Graphics 3.0 [23.05.25593.18]
[ext_oneapi_level_zero:gpu:0] Intel(R) Level-Zero, Intel(R) Arc(TM) A380 Graphics 1.3 [1.3.25593]
[ext_oneapi_cuda:gpu:0] NVIDIA CUDA BACKEND, NVIDIA GeForce GTX 1060 6GB 0.0 [CUDA 12.1]
```

SYCL_DEVICE_FILTER は非推奨となりました。2023.1 では警告が表示されますが、まだ利用できます (今後廃止される可能性があります)

JIT コンパイルのキャッシュ制御

- SYCL* ランタイムが JIT コンパイルしたデバイスのバイナリーを永続的に維持する、キャッシュ排出のサイズや日数、バイナリーのサイズなどを SYCL_CACHE_* 環境変数で制御できます

例:
export SYCL_CACHE_PERSISTENT=1 // JIT コンパイルしたバイナリーを保持
export SYCL_CACHE_THRESHOLD=[整数] // 指定された日数保持

CUDA* と HIP プラグインの制御

SYCL_PI_CUDA_MAX_LOCAL_MEM_SIZE
SYCL_PI_HIP_MAX_LOCAL_MEM_SIZE

- ローカルメモリーの割り当て最大サイズをバイト単位で指定します
- 値がデバイスの能力を上回ると、sycl::runtime_error がスローされます
- 完全なエラーメッセージを取得するには、SYCL_RT_WARNING_LEVEL=2 に設定します
- SYCL_PI_CUDA/HIP_MAX_LOCAL_MEM_SIZE のデフォルト値は、ハードウェアごとに異なります

制限事項と注意事項

- Codeplay* NVIDIA*/AMD* GPU プラグインを使用する場合、インテル® oneAPI ツールキット 2023.0 のリリースでは、clang++ コンパイラーを使用する必要がありました
- インテル® oneAPI ツールキット 2023.1 では icpx を利用できるようになりましたが、いくつかの制限事項があります。この場合 clang++ を使用すると制限は緩和されます

icpx でサポートされない機能

- CXX stdlib のサポート
- SYCL* グループ・アルゴリズム:

  broadcast、joint_exclusive_scan、joint_inclusive_scan、

  exclusive_scan_over_group、inclusive_scan_over_group
- デバイスコードの分割
- SYCL* リダクション

まとめ

- すでにマルチスレッド化の経験があれば、ヘテロジニアス・コンピューティングの導入は容易です
- 利用可能なデバイスがすでに皆さんのシステムには搭載されているかもしれません
- 開発ツールはすでに用意されています

参考資料

- oneAPI プログラミング・ガイド
- oneAPI for NVIDIA* GPU 2023.1.0 ガイド
- ベータ版 oneAPI for AMD* GPU 2023.1.0 ガイド
- oneAPI DPC++ 導入ガイド
- oneAPI GPU 最適化ガイド
- DPC++ ランタイム環境変数
- SYCL* 2020 仕様 Rev6 日本語版

オンデマンド・ウェビナーの紹介

- インテル® oneAPI ツールキットのアップデート ~ バージョン 2023 の注目機能と他社製 GPU 向けプラグインの紹介 ~
- OpenMP* を使用した GPU オフロード方法
- GPU 向けのインテル® VTune™ プロファイラーの機能と GPU 最適化
- インテル® DPC++ 互換性ツールの紹介

https://www.isus.jp/products/oneapi/webinar/