この記事は、インテル® デベロッパー・ゾーンに掲載されている「OpenCL* Design and Programming Guide for the Intel® Xeon Phi™ Coprocessor」の日本語参考訳です。

インテル® Xeon Phi™ コプロセッサー向け OpenCL* アプリケーションの設計とプログラミング・ガイド

概要

この記事は、インテル® Xeon Phi™ コプロセッサー向けのハイパフォーマンス OpenCL* アプリケーションを開発するための設計とコーディング・ガイドです。インテル® Xeon Phi™ コプロセッサーのアーキテクチャーおよびマイクロアーキテクチャーを紹介した後、OpenCL* 構造を効率良く利用してインテル® Xeon Phi™ コプロセッサーのハードウェアを活用する方法を説明します。パフォーマンス・アプリケーションではハードウェアの並列性を利用することが不可欠であるため、ここではインテル® Xeon Phi™ コプロセッサー上で OpenCL* アプリケーションの並列性を高める方法を説明します。この記事をお読みになることで、OpenCL* を使用してインテル® Xeon Phi™ コプロセッサー向けに最適なアプリケーションを設計しプログラミングする準備ができます。

この記事が重要である理由

OpenCL* は移植性のあるプログラミング・モデルですが、移植後にパフォーマンスが維持されることは保証されません。従来の GPU とインテル® Xeon Phi™ コプロセッサーのハードウェア設計は異なります。その違いは、アプリケーションの異なる最適化が、それぞれ影響することです。例えば、従来の GPU は、開発者が明示的にプログラミングする必要がある、高速な共有ローカルメモリーに依存しています。インテル® Xeon Phi™ コプロセッサーには、メモリーアクセスが自動的に高速化される、従来の CPU キャッシュに似た完全にコヒーレントなキャッシュ階層を実装しています。また、従来の GPU は小さなたくさんのスレッドのハードウェア・スケジューリングに基づいていますが、インテル® Xeon Phi™ コプロセッサーは中規模サイズのスレッドをスケジューリングするデバイス OS に依存します。このため、通常、アプリケーションを実行するハードウェアに合わせてパフォーマンスをチューニングすると良いでしょう。

異なるデバイスには異なる OpenCL* の最適化が必要か?

必ずしもそうとは言えません。インテル® Xeon Phi™ コプロセッサーで 50% 高速に実行するため、小規模な #ifdef をコードに追加しますか? さらにそのため 1,000 行のファイルをコピーしますか? わずか 10% のスピードアップためそのような作業を行いますか? あるいは、最適化を無条件に追加してインテル® Xeon Phi™ コプロセッサーのパフォーマンスが 50% 向上する代わりにほかのデバイスのパフォーマンスが 10% 低下してもかまいませんか? すべて開発者が決めることです。場合によっては、クロスデバイスのパフォーマンスと OpenCL* アプリケーションの保守性をトレードオフする必要があります。この記事は、開発者がインテル® Xeon Phi™ コプロ

セッサーの潜在的なパフォーマンスを調査した後、測定値に基づいた決定を下せるように構成されています。この記事では、開発者が自身で回答を見つける上で役立つツールも紹介します。

インテル® Xeon Phi™ コプロセッサーの高レベル・ハードウェアの概要

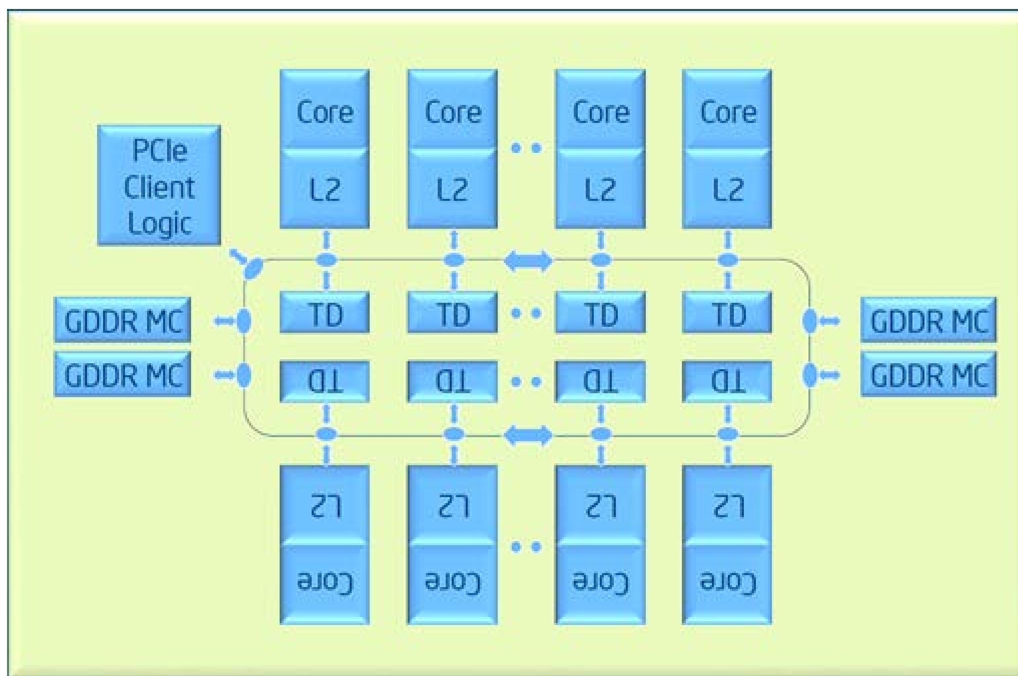

図 1. インテル® Xeon Phi™ コプロセッサーのマイクロアーキテクチャー

インテル® Xeon Phi™ コプロセッサーには多くのコアが備わっています。各コアには SIMD ベクトル命令を実行できる 512 ビット幅のベクトル演算ユニットがあります。L1 キャッシュは各コアに含まれています (32KB データ + 32KB 命令)。L2 キャッシュは各コアに結合しています (512KB データ + 命令、L1 D キャッシュは包括的)。高速インターコネクトによって L2 キャッシュとメモリー・サブシステム間のデータを転送します。各コアは最大 4 つのハードウェア・スレッドを同時に実行できます。この同時マルチスレッディングは、命令とメモリーのレイテンシーの隠蔽に役立ちます。OpenCL* は、これらの詳細のほとんどを開発者から見えないようにします。

インテル® Xeon Phi™ コプロセッサーのハードウェアについての詳細:

http://www.isus.jp/article/idz/mic-developer/

インテル® Xeon Phi™ コプロセッサーのパフォーマンスの考察

マルチスレッド並列処理

インテル® Xeon Phi™ コプロセッサーのハードウェアには多くのコアが搭載されます (製品ごとにコア数は異なりますが、この記事では 60 個のコアを仮定しています)。各コアは最大 4 つのハードウェア・スレッドを実行できます。ほとんどの場合、パフォーマンスを最大限にするには 240 のスレッドでタスクを実行します。ハードウェア・スレッドの正確な数は clGetDeviceInfor(NUM_COMPUTE_UNITS); で調べることができます。

ベクトル化

インテル® Xeon Phi™ コプロセッサーのベクトルサイズは 512 ビット 幅です。このベクトルで、8 つの倍精度浮動小数点数または 16 個の単精度浮動小数点数を処理します。インテル® Xeon Phi™ コプロセッサーの各コアは、1 サイクルごとに 1 つのベクトル演算命令を発行できます。

PCI Express* (PCIe) バス・インターフェイス

インテル® Xeon Phi™ コプロセッサーは PCIe バスを介してホスト・プラットフォームに接続されます。PCIe バス上のデータ転送はレイテンシーが最も高く、帯域幅が最も低くなります。ほかの PCIe デバイスと同様、このトラフィックを最小限に抑えなければいけません。

メモリー・サブシステム

インテル® Xeon Phi™ コプロセッサーには、3 レベルのメモリー階層 (GDDR、L2 キャッシュ、L1 キャッシュ) があります。次の表に、重要なキャッシュ情報を示します。

	L1 (データ + 命令)	**共有 L2**
合計サイズ	32KB + 32KB	512KB
ミス・レイテンシー	15-30 サイクル	500-1000 サイクル

L1 キャッシュのアクセス・レイテンシーは 1 サイクルです。

インテル® Xeon Phi™ コプロセッサーはインオーダー・マイクロアーキテクチャーであるため、メモリーアクセスのレイテンシーはソフトウェアのパフォーマンスに大きく影響します。幸い、開発者はこれらのレイテンシーを減らすことができます。プリフェッチャーはメモリー・レイテンシーの隠蔽に役立つ機能の 1 つです。プリフェッチャーについては、後で詳しく説明します。

データ・アクセス・パターン

連続したメモリーにアクセスすることが、インテル® Xeon Phi™ コプロセッサーのメモリーを最も高速にアクセスする方法です。キャッシュの利用効率を高め、TLB (トランスレーション・ルックアサイド・バッファー) ミスの数を減らし、ハードウェア・プリフェッチャーが動作するようにします。

OpenCL* 構造をインテル® Xeon Phi™ コプロセッサーへマッピング

OpenCL* 構造がインテル® Xeon Phi™ コプロセッサーにどのように実装されるか理解することで、コプロセッサーのハードウェアを最大限に活用するアプリケーションを設計できます。また、コプロセッサーのパフォーマンスの落とし穴を避けることができるでしょう。

概念的には、初期化の際に、OpenCL* ドライバーは 240 のソフトウェア・スレッド (60 コア構成の場合) を生成して、ハードウェア・スレッドに関連付けます。次に、clEnqueueNDRange() 呼び出しにより、ドライバーは現在の DRange のワークグループを 240 のスレッドにスケジューリングします。ワークグループはスレッドにスケジューリングされる最も小さなタスクです。そのため、240 未満のワークグループで clEnqueueNDRange() を呼び出すと、コプロセッサーが十分に活用されません。

OpenCL* コンパイラーは、ワークグループを実行する最適化されたルーチンを作成します。次の擬似コードで示すように、このルーチンは最大 3 つの入れ子のループで構成されます。

```
__Kernel ABC(…)
For (int i = 0; i < get_local_size(2); i++)
	For (int j = 0; j < get_local_size(1); j++)
		For (int k = 0; k < get_local_size(0); k++)
			Kernel_Body;
```

最内ループは NDRange の次元 0 に適用されることに注意してください。これは、高いパフォーマンスが要求されるコードのアクセスパターンに直接影響します。また、暗黙のベクトル化の効率にも影響を及ぼします。

OpenCL* コンパイラーは次元 0 のループに基づいてワークグループのルーチンを暗黙にベクトル化します。つまり、次元 0 のループはベクトルサイズによってアンロールされます。ベクトル化されたワークグループのコードは次のようになります。

```
__Kernel ABC(…)
For (int i = 0; i < get_local_size(2); i++)
	For (int j = 0; j < get_local_size(1); j++)
		For (int k = 0; k < get_local_size(0); k += VECTOR_SIZE)
			Vector_Kernel_Body;
```

インテル® Xeon Phi™ コプロセッサーのベクトルサイズは、(カーネルで使われるデータ型にかかわらず) 16 です。ただし、将来は、より多くの命令レベルの並列化 (ILP) が可能になるようにベクトルサイズを増やす予定です。

アルゴリズムの並列性

OpenCL* 仕様は並列性と並行性を表現するさまざまな方法を提供しますが、それらの一部はインテル® Xeon Phi™ コプロセッサーには適切にマップされません。この記事では、OpenCL* の並列性を利用してアプリケーションを設計できるように、OpenCL* 構造をコプロセッサーにマップする方法を説明します。

マルチスレッド

240 のハードウェア・スレッドを効率良く利用するには、NDRange あたりのワークグループを 1000 以上にします。NDRange あたりのワークグループが 180 から 240 であれば最低限のスレッドは利用されますが、実行時のロードバランスが悪くなり、大きなオーバーヘッドが発生します。

推奨: インテル® Xeon Phi™ コプロセッサーのハードウェア・スレッドを適切に利用するには、NDRange あたりのワークグループを少なくとも 1000 にしてください。NDRange あたりのワークグループが 100 未満のアプリケーションではスレッドが十分に利用されません。

1 つのワークグループの実行を継続することもスレッド化の効率に影響を及ぼします。オーバーヘッドが大きくなる軽量のワークグループも推奨しません。

ベクトル化

インテル® Xeon Phi™ コプロセッサー上の OpenCL* には暗黙のベクトル化モジュールが含まれます。OpenCL* コンパイラーは、次元 0 のワークアイテムを扱うワークグループ・ループを暗黙的に自動ベクトル化します (上記の例を参照)。カーネルで利用されるデータ型にかかわらず、現在のベクトル幅は 16 です。将来の実装では、32 要素のベクトルをサポートする予定です。OpenCL* のワークアイテムが独立していることが保証されれば、OpenCL* ベクトライザーはベクトル化が適用可能かどうかを調べる必要がなくなります。ただし、ベクトル化されたカーネルは、次元 0 のローカルサイズが 16 以上の場合にのみ使用されます。そうでない場合、OpenCL* ランタイムは各ワークアイテムをスカラーカーネルで実行します。次元 0 のワークグループのサイズが 16 で割り切れない場合、ワークグループの最後をスカラーコードで実行する必要があります。これは大きな (例えば、次元 0 のアイテムが 1024 の) ワークグループでは問題ありませんが、次元 0 のサイズが 31 のワークグループでは問題になります。

推奨 1: OpenCL* コンパイラーが暗黙のベクトル化を準備するためにスカラー処理を行えるよう、手動でカーネルをベクトル化しないでください。

推奨 2: 32 (現在は 16) で割り切れないワークグループのサイズの利用を避けてください。

ワークアイテム ID が均一でない制御フロー

ここでは、暗黙のベクトル化のコンテキストにおいて、均一な制御フローと均一でない制御フローの違いについて説明します。均一な制御フローはパフォーマンスに若干悪影響を及ぼすことがあるため、この違いを理解することは重要です。ただし、均一でない制御フローは最内 NDRange 次元内で大幅なパフォーマンス・オーバーヘッドを引き起こします。ベクトル化されたループ (次元 0) の均一性は重要です。

ワークグループ内のすべてのワークアイテムが分岐を一方に実行することが静的に保証されている場合、分岐は均一です。

均一な分岐の例:

```
//isSimple is a kernel argument
```

```
Int LID = get_local_id(0);
If (isSimple == 0)
       Res = buff[LID]
```

均一でない分岐の例:

```
Int LID = get_local_id(0);
If (LID == 0)
       Res = -1;
```

別の均一な分岐の例:

```
Int LID = get_local_id(1);
//Uniform as the IF is based on dimension one, while vectorization on dimension on.
If (LID == 0)
       Res = -1;
```

コンパイラーは、ベクトル化を行う一方、予測に基づいて均一でない制御フローのコードを線形化 (平坦に) しなければいけません。予測の最初の大きなコストは分岐の両方を実行することです。マスク実行によりペナルティーが発生します。

次のカーネルコードを仮定します。

```
Int gid = get_global_id(0);
If(gid % 32 == 0)
       Res = HandleEdgeCase();
Else
       Res = HandleCommonCase();
End
```

ベクトル化 (および予測) 後、コードは次のようになります。

```
int16 gid = get16_global_id(0);
uint mask;
Mask = compare16int((gid % broadcast16(32)), 0)
res_if = HandleEdgeCase();
res_else = HandleCommonCase();
Res = (res_if & mask) | (res_else & not(mask));
```

IF と ELSE はどちらもすべてのワークアイテムに対して実行されることに注意してください。

推奨: 分岐を避けてください (特に次元 0 で均一でない分岐)。

データ・アライメント

さまざまな理由により、ベクトルサイズにアライメントされたメモリーアクセスは、アライメントされていないメモリーアクセスよりも高速です。インテル® Xeon Phi™ コプロセッサーでは、OpenCL* バッファーはベクトルサイズにアライメントされたアドレスで開始することが保証されています。しかし、これは最初のワークグループがアライメントされたアドレスから開始されることを保証している

だけです。すべてのワークグループが適切にアライメントされていることを保証するには、ワークグループのサイズ (ローカルサイズ) が 16 (将来の実装を考慮する場合は 32) で割り切れる必要があります。ローカルサイズ NULL で EnqueueNDRange を呼び出すと、OpenCL* ドライバーが最適なワークグループのサイズを選択します。ドライバーはアライメント要件に合致するワークグループのサイズを選択するはずですが、開発者は、ワークグループへの分割が効率良く行われるように、グローバルサイズが VECTOR_SIZE で分割でき、商が十分大きいことを確認する必要があります。"十分大きい" とは、小さなカーネルでは 1,000,000、カーネルに 1000 の反復ループを含む大きなカーネルでは 1000 と考えてください。NDRange のオフセットがアライメントを変更する場合もあります。

推奨 1: NDrange のオフセットを使用しないでください。オフセットを使用しなければいけない場合は、32 の倍数 (少なくとも 16 の倍数) にします。

推奨 2: 32 の倍数 (少なくとも 16 の倍数) のローカルサイズを使用してください。

インテル® Xeon Phi™ コプロセッサーのメモリー・サブシステムのメリットが得られるようにアルゴリズムを設計する

インテル® Xeon Phi™ コプロセッサーはインオーダー・マイクロアーキテクチャーであるため、メモリーアクセスのレイテンシーに大きく影響されます。アプリケーション・レベルでメモリー関連の最適化を行うと、パフォーマンスが 2-4 倍向上することがあります。

ワークグループ内のデータの再利用

キャッシュに格納されているデータの再利用を最大限にするようにアプリケーションを設計することが、最初に取り組むべきメモリー最適化です。ただし、データを再利用する必要があるのは特定のアルゴリズムのみです。例えば、2 つの行列の加算はデータを再利用しません。しかし、2 つの行列の乗算 (GEMM) は多くのデータを再利用します。そのため、ブロッキング/タイリング最適化の明白な候補と言えます。詳細は、「インテル® SDK for OpenCL* Applications XE – 最適化ガイド」(英語) を参照してください。

データ再利用のメリットを得るには、前述したように、ワークグループの暗黙のループを考慮する必要があります。開発者は、ローカルサイズを定義してこれらのループを制御できます。また、カーネルに明示的にループを追加できます。

ワークグループ間のデータの再利用

ワークグループ間のデータの再利用は、より困難です。現在、インテル® Xeon Phi™ コプロセッサーの OpenCL* ではワークグループのスケジューリングを十分に制御できません。そのため、ワークグループ間のデータの再利用はほぼ不可能です。この問題については将来の開発で検討される予定です。

データ・アクセス・パターン

連続するデータアクセスを行うと、メモリーシステムのパフォーマンスは最適になります。この連続データアクセスを検討する際、ワークグループの暗黙のループ構造を理解することが重要です。最も内側の暗黙のループは、次元 0 のループです。カーネルにそれ以上 (暗黙の) ループがない場合、暗黙の次元 0 のループに注意して連続するメモリーアクセスを行ってください。次に例を示します。

次のコードはメモリーの 2D バッファーに連続してアクセスします (推奨)。

```
__kernel ABC(…){
int ID1 = get_global_id(1);
int ID0 = get_global_id(0);
res[ID1][ID0] = param1 * buffer[ID1][ID0];
}
```

次のコードはメモリーの 2D バッファーに連続してアクセスしません (非推奨)。

```
__kernel ABC(…){
int ID1 = get_global_id(1);
int ID0 = get_global_id(0);
res[ID0][ID1] = param1 * buffer[ID0][ID1];
}
```

2 つ目のコード例は 2D バッファーを "列優先" でスキャンします。ここでベクトル化の際、2 つの問題が起こります。
1) 入力ベクトルデータは 16 の連続する行から列とともに収集する必要があります。結果はスキャッター命令を用いて 16 の異なる行にストアされます。これらの処理により実行が遅くなります。
2) メモリーアクセスは各反復で連続していません。

2 つの問題はどちらも TLB の負荷を増やしプリフェッチを妨げます。

簡単な 1 次元の例を次に示します。

次の例は連続してアクセスします (推奨)。

```
Int id = get_global_id(0);
A[id]= B[id];
```

次の例は連続してアクセスしません (非推奨)。

```
Int id = get_global_id(0);
A[id*4] = B[id*4]
```

推奨: ID(0) を使用して行内でメモリーを連続してインデックスします。
明示的な 2D バッファーの場合: buffer[ID1][ID0]
1D バッファーへの 2D インデックスの場合: buffer[STRIDE * ID1 + ID0]

カーネルに明示的なループが含まれる場合、暗黙のベクトル化は、ID(0) の暗黙のループに基づくことに注意してください。そのため、OpenCL* ID によるバッファーのアクセスは上記の推奨

(buffer[ID1][ID0]) に従ってください。ベクトルアクセスが連続することで、効率が向上します。内部ループ・インデックス (idx) でバッファーにアクセスすると、内部ループ (buffer[ID1][idx]) 内のアクセスは連続し、ベクトル化されたループに対して均一になります。ただし、ID0 と idx の組み合わせは避けてください。例えば、buffer[ID0][idx] はベクトル化されたループごとにストライドされるため、ギャザー/スキャッター命令が生成されます。

データ形式

ピュア **SOA** (配列構造体) データ形式は、シンプルで効率良いベクトルのロードとストアを行います。しかし、空間の局所性は低く、TLB の負荷は高く、同時に使用されるページ数は多くなります。

AOS (構造体配列) データ形式は、生成されるベクトル化されたカーネルでギャザー/スキャッター命令によりデータをロードおよびストアする必要があるため、シンプルなベクトルのロードおよびストアよりも効率は低下します。ただし、ランダム・アクセス・パターンでは、空間の局所性が優れている AOS データ形式のほうが SOA データ形式よりも効率が高くなることがあります。SOA データ形式のランダムアクセスでもギャザー/スキャッター命令が生成されることに注意してください。

3 つ目のオプションは **AOSOA** (小さな配列の構造体配列) です。インテル® Xeon Phi™ コプロセッサー向けには、配列のサイズを 32 にして 32 要素までベクトル化できるようにします。

```
struct Point32 {     float x[32], y[32], z[32]; };
__kernel void ABC(__global Point32* ptrData)
```

AOSOA は、シンプルなベクトルロードを利用して、TLB をオーバーロードせず、多くのページにアクセスすることなく、効率良いベクトル化を行います。AOSOA の短所はコードの判読が難しくなることです。ほとんどの開発者は AOSOA によるアクセスを考慮せずに利用しています。

データ・プリフェッチ

インテル® Xeon Phi™ コプロセッサーはインオーダー・マイクロアーキテクチャーであるため、データをコアの近くのキャッシュに格納して、ほかの計算と並列に処理するには、データ・プリフェッチが不可欠です。並列システムでも、ロードとストアは連続して実行されます。例えば、2 つのロード命令は連続して実行されます。プリフェッチ命令は例外で、 プリフェッチ命令は、ほかの命令と並列に実行されます。そのため、時間通りに完了しなかったプリフェッチ命令を、ほかの命令と並列に実行することでパフォーマンスを向上できます。キャッシュミスが発生すると、スレッドがストールするだけでなく、命令を再発行するため、数サイクルのペナルティーが追加されます。インテル® Xeon Phi™ コプロセッサーには、L2 へのシンプルな自動ハードウェア・プリフェッチャーが備わっています。ハードウェア・プリフェッチャーが動作するには多少の時間が必要です。また、4KB の仮想ページ境界ごとに再起動する必要があります。

L1 と L2 への自動ソフトウェア・プリフェッチは、(解析によって) 挿入するメリットがあると判断された場合は、将来の反復でアクセスされるデータのために OpenCL* コンパイラーによって挿入されます。ベータリリースでは、自動ソフトウェア・プリフェッチを一部サポートしています。

プログラマーは、PREFETCH ビルトインを使って、手動プリフェッチを OpenCL* カーネルに挿入することができます。現在、手動プリフェッチはプログラマーが要求した位置とアドレスに正確に挿入されますが、これらは L2 プリフェッチャーによって制限されます。将来、OpenCL* コンパイラーには PREFETCH ビルトイン向けに L2 プリフェッチャーと L1 プリフェッチャーの両方が追加される予定です。また、プログラマーによって示される位置とストライドも改善される予定です。手動プリフェッチは、データが実際に使われる時点より少なくとも 500 サイクル前に挿入してください。通常、主入力/出力バッファーのみをプリフェッチする必要があります。

ローカルメモリーとバリア

従来の GPU には手動で管理する共有ローカルメモリー (SLM) が含まれていますが、インテル® Xeon Phi™ コプロセッサーには CPU と同様の 2 レベルのキャッシュシステム (自動) が備わっています。そのため、インテル® Xeon Phi™ コプロセッサーで OpenCL* SLM を利用するメリットはありません。また、コプロセッサーのローカルメモリーは通常の GDDR メモリーに割り当てられ、キャッシュシステムでサポートされています。データのコピーと管理が冗長になるため、オーバーヘッドが発生します。

推奨: インテル® Xeon Phi™ コプロセッサーの共有ローカルメモリーを使用しないでください。

インテル® Xeon Phi™ コプロセッサーにはバリアをサポートする特別なハードウェア機能はありません。そのため、バリアはコプロセッサー上で OpenCL* によってエミュレートされます。バリアの使用を避けることを推奨します。また、カーネルを 2 つに分割する手法はバリアよりも遅くなるため、推奨されません。

ベータリリースの時点では、バリアと 16 で割り切れないワークグループのサイズを組み合わせると、スカラーカーネルが実行されます。この組み合わせは避けてください。現在、OpenCL* コンパイラー内でこの組み合わせの最適化が重要であるとは考えられていません。

まとめ

インテル® Xeon Phi™ コプロセッサー向けの OpenCL* アプリケーションを設計するときは、次の点に注意してください。

1. 各 NDRange 内に十分な数のワークグループを含めてください。1000 以上を推奨します。
2. 軽量のワークグループは避けてください。最大のローカルサイズ (現在は 1024) を使用してください。ワークグループのサイズは 32 の倍数にします。
3. ID(0) 依存の制御フローを避けてください。効率良い暗黙のベクトル化が行われます。
4. 連続するデータアクセスを行ってください。
5. データ形式は、まばらなランダムアクセスの場合は AOS、その他の場合はピュア SOA または AOSOA(32) にしてください。
6. ワークグループ内のキャッシュでデータを再利用してください (ブロッキング/タイリング)。
7. 自動プリフェッチが動作しなかった場合、PREFETCH ビルトインを用いてグローバルデータを使う前に 500-1000 サイクルキャッシュに格納してください。
8. ローカルメモリーを使用しないでください。バリアも避けてください。

関連ドキュメント

インテル® SDK for OpenCL Applications XE

インテル® Xeon Phi™ コプロセッサー向けの最適化とパフォーマンス・チューニング

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。