インテル® ISPC パフォーマンス・ガイド

このドキュメントは ispc GitHub に公開されている『Intel® ISPC Performance Guide』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) が提供する SPMD プログラミング・モデルは、プロセッサーの SIMD ベクトル・ハードウェアを効率良く使用することで、多くのワークロードに対し優れたパフォーマンスをごく自然に発揮します。このガイドでは、インテル® ISPC を最大限に活用する方法を詳しく説明します。

目次

主要な概念

この節では、ハイパフォーマンスのインテル® ISPC プログラムを作成する際に、理解すべき最も重要な 4 つの概念について説明します。『インテル® ISPC ユーザーガイド』で説明されているトピックに精通していることを前提とします。

foreach による効率良い反復

foreach 並列反復構造は、語彙的には通常の for() ループと同等ですが、パフォーマンス的には大きな利点があります (構文とセマンティクスについては、『インテル® ISPC ユーザーガイド』の foreach の説明を参照)。配列内の複数の要素を繰り返し処理しそれぞれに対し計算を行う簡単な関数の例を考えてみましょう。

```
export void foo(uniform int a[], uniform int count) {
    for (int i = programIndex; i < count; i += programCount) {
        // a[i] の計算を行います
    }
}
```

実行中の計算内容に応じて、この関数から生成されるコードがループを通過するたびに、プログラム・インスタンスのギャング全体で常にパックド計算でデータを処理するようにコードを変更することで改善される可能性があります。これにより、インテル® ISPC は、実行マスクが「すべてオン」であることが判明している場合、効率良いコードを生成することができます。そして最後に、if 文でギャングを埋めきれなかった余りのデータを処理します。

```
export void foo(uniform int a[], uniform int count) {
    // 最初に、ループ反復の開始時にギャング内のすべての
    // プログラム・インスタンスがアクティブになるまでループします
    uniform int countBase = count & ~(programCount-1);
    for (uniform int i = 0; i < countBase; i += programCount) {
        int index = i + programIndex;
        // a[index] の計算を行います
    }
    // 最後に残りビットを処理します
    int index = countBase + programIndex;
    if (index < count) {
        // a[index] の計算を行います
    }
}
```

上記のコードのパフォーマンスは、最初のバージョンよりも高い可能性がありますが、ループ本体のコードが重複しています (または、別のユーティリティー関数に移動する必要があります)。

次のような foreach ループ構造を使用すると、最初のバージョンの簡潔さとともに、2 番目のバージョンのパフォーマンスの恩恵も受けられます。

```
export void foo(uniform int a[], uniform int count) {
    foreach (i = 0 ... count) {
        // a[i] の計算を行います
    }
}
```

foreach_tiled による制御フローの一貫性を向上

実行する計算によっては、foreach よりも foreach_tiled の方が高いパフォーマンスを発揮する場合があります (foreach_tiled の構文とセマンティクスについては、『インテル® ISPC ユーザーガイド』を参照)。次のような多次元配列の反復について考えてみます。

```
foreach (i = 0 ... width, j = 0 ... height) {
    // 要素 (i,j) の計算を行います
}
```

foreach 文が使用されると、プログラム・インスタンスのギャングの要素は、j の単一値で i 全体の programCount 要素のスパンを取得することで、i と j の値にマップされます。例えば、上記の foreach 文は、次の for ループに対応します。

```
for (uniform int j = 0; j < height; ++j)
    for (int i = 0; i < width; i += programCount) {
        // 計算を実行します
}
```

多次元領域を反復処理する場合、foreach_tiled 文は領域の n 次元正方セグメントを選択することで、プログラム・インスタンスをデータにマップします。例えば、8 レーンのプログラム・インスタンスのギャングを持つターゲットでは、次のコードと同様にドメインを反復処理するコードを生成します (ただし、より効率的)。

```
for (int j = programIndex/4; j < height; j += 2)
    for (int i = programIndex%4; i < width; i += 4) {
        // 計算を実行します
}
```

これは、プログラム・インスタンスの各ギャングは、ドメインの 2x4 タイルで動作します。例えば、16 分割の 2D 反復処理では 4x4 のタイルを、8 分割の 4D 反復処理では 1x2x2x2 のタイルを処理するなど、高次元の反復処理と異なるギャングサイズでも同様のマッピングが行われます。

foreach_tiled を使用すると、ドメインのコンパクトな領域を反復処理する最適化により、パフォーマンスが向上します (foreach は、線形メモリーアクセスによりドメインを反復処理します)。ドメインのコンパクトな領域を処理するには 2 つの利点があります。

まず、ギャングのプログラム・インスタンスの制御フローの一貫性を向上させます。データ依存の制御フローが処理するドメイン内のデータ値に関連し、データ値に一貫性がある場合、コンパクトな領域で反復処理すると制御フローの一貫性が向上します。

次に、コンパクトな領域を処理することで、ギャング内のプログラム・インスタンスがアクセスするデータの一貫性が向上し、キャッシュヒット率が高まるためパフォーマンス上の利点が得られる可能性があります。

例えば、インテル® ISPC ディストリビューションに含まれるレイトレーサーの例 (examples/rt ディレクトリーにあります) では、foreach よりも foreach_tiled を使用してピクセルを反復処理すると 20% パフォーマンスが向上します。これは、プログラム・インスタンスのギャングであるレイセットがシーンごとにコヒーレントな領域をアクセスするためです。

一貫性のある制御フロー構造を使用する

『インテル® ISPC ユーザーガイド』のインテル® ISPC 並列実行モデルの説明を思い出してください。uniform を評価する if 文は、varying を評価する if 文よりも効率良くコードをコンパイルできます。一貫性のある cif 文は、varying な評価を行う場合、uniform な if 文により多くの利点を得られます。

この場合、コンパイラーが if 文の評価に生成するコードは、次のようなコードになります。

```
bool expr = /* cif 条件を評価 */
if (all(expr)) {
    // if の評価が "true" の場合にのみ実行
} else if (!any(expr)) {
    // if の評価が "false" の場合にのみ実行
} else {
    true と false の両方で実行し、マスクを適切に更新します
}
```

実行中の異なる SPMD プログラム・インスタンスが、ブール値の if 評価に一貫した値を持っていない場合、cif を使用することで all および any の評価と対応する分岐によるオーバーヘッドが発生します。プログラム・インスタンスが頻繁に同じブール値を計算する場合、このオーバーヘッドは隠匿される可能性があります。制御フローに一貫性がない場合は、このオーバーヘッドはパフォーマンスにのみ影響します。

同様に、インテル® ISPC は cfor、cwhile、および cdo 文を提供します。これらの文は、対応する「c」プリフィクスのない文と同じ意味を持ちます。

適切であれば常に uniform を使用する

ギャング内のすべてのプログラム・インスタンスで常に同じ値を持つ変数は、uniform 修飾子を使用して変数を宣言します。これにより、インテル® ISPC はさまざまな方法により適切なコードを生成できます。

簡単な例として、常に同じ回数だけ反復を行う for ループを考えてみます。

```
for (int i = 0; i < 10; ++i)
    // 10 回処理を行います
```

上記のように、可変変数として i を使用する場合、コンパイラーはすべてのプログラム・インスタンスが同じ制御フローをたどらないケースを処理する命令を生成するため、ループを制御するコードにオーバーヘッドが発生します (ループの上限 10 が可変である場合)。

上記のループの代わりに i を uniform とします。

```
for (uniform int i = 0; i < 10; ++i)
    // 10 回処理を行います
```

これは、より優れたコードを生成できます。さらに、ループがアンロールされる可能性もあります。

コンパイラーは、単純なケースであればこのような状況を検出できる可能性がありますが、実際の計算形式を理解できるよう、コンパイラーをできるだけアシストすることは常に最良の結果につながります。

可能であれば「配列構造体」レイアウトを使用する

一般に、メモリーアクセスのパフォーマンスは、実行中のプログラム・インスタンスが連続したメモリー領域にアクセスすると最も高くなります。この場合、ギャザーやスキャッターではない効率的なベクトルロードとストア命令が利用できます。この問題の例として、単純な Point データタイプの配列を、AOS (構造体配列) レイアウトにしてアクセスするケースを考えてみます。

```
struct Point { float x, y, z; };
uniform Point pts[...];
float v = pts[programIndex].x;
```

このコードでは、pts[programIndex].x は、メモリー内で x 値の後に y と z があるため、非連続なメモリー位置にアクセスします。v の値を取得するにはギャザーが必要であり、それに伴ってパフォーマンスも低下します。

Point を配列構造体 (SOA) タイプとして定義することで、効率良くアクセスできます。

```
struct Point8 { float x[8], y[8], z[8]; };
uniform Point8 pts8[...];
int majorIndex = programIndex / 8;
int minorIndex = programIndex % 8;
float v = pts8[majorIndex].x[minorIndex];
```

この場合、それぞれの Point8 は、8 つの y 値の前にメモリー内で連続する 8 つの x 値があり、y の後には 8 つの z 値が続きます。ギャングサイズが 8 以下の場合、v へのアクセスはすべてのプログラム・インスタンスで majorIndex が同じ値になり、x[8] 配列の連続した要素をベクトルロードでアクセスできます (ギャングサイズが大きくなると、8 レーンのベクトルロードが 2 つ発行されますが、これもかなり効率的です)。

しかし、上記のコードの構文は複雑です。この方法で SOA データにアクセスするのは、AOS レイアウトでデータにアクセスするコードに比べエレガントとはいえません。インテル® ISPC の soa 修飾子を使用すると、AOS レイアウトに対応する洗練されたデータアクセス構造を維持したまま、Point タイプに対応する変換を行うことができます。

```
soa<8> Point pts[...];
float v = pts[programIndex].x;
```

SOA レイアウトを言語のタイプシステムの第 1 コンセプトとしたため、レイアウト間でデータを変換する関数を容易に作成できるようになりました。例えば、次の aos_to_soa 関数は、与えられた Point タイプの count 要素を AOS から 8 レーンの SOA レイアウトに変換します。これは、呼び出し側が pts_soa 出力配列に十分な領域を確保していることを前提としています。

```
void aos_to_soa(uniform Point pts_aos[], uniform int count,
                soa<8> pts_soa[]) {
    foreach (i = 0 ... count)
        pts_soa[i] = pts_aos[i];
}
```

同様に、SOA から AOS に戻す関数も作成できます。

ヒントとテクニック

この節では、ispc プログラムを記述する際に役立つ、いくつかのヒントやテクニックを紹介します。

ギャザーとスキャッターを理解する

プログラム・インスタンスからのメモリーリードやライトは、不規則な場所 (連続した場所や単一の場所ではなく) にアクセスするため、相対的に効率が悪くなることがあります。例えば、次のような単純な配列のインデックス計算について考えてみます。

```
int i = ....;
uniform float x[10] = { ...};
float f = x[i];
```

インデックス i は可変値であるため、ギャングのプログラム・インスタンスは、一般に配列 x の異なる位置を読み取ります。すべての CPU がギャザー命令をサポートするわけではないので、インテル® ISPC はこれらのメモリーリードをシリアル化し、実行中のプログラム・インスタンスごとに異なるメモリーをロードして、結果を f にパックする必要があります (類似のケースが x[i] への書き込みでも発生します)。

多くの場合、このようなギャザーは避けられません。プログラム・インスタンスは、一貫性のないメモリー位置にアクセスするだけで済みます。しかし、配列インデックス i がすべてのプログラム・インスタンスで同じ値を持つ場合、もしくは配列の連続したデータセットへのアクセスする場合は、それぞれギャザーとスキャッターの代わりに効率的なロード命令とストア命令を生成できます。

多くの場合、インテル® ISPC は、可変インデックスでアクセスされるメモリー位置がすべて同一であるか、均一であることを推測します。次の例について考えてみます。

```
uniform int x = ...;
int y = x;
return array[y];
```

コンパイラーは、y が uniform 変数でなくても、すべてのプログラム・インスタンスで同じ位置からロードされると判断できます。この場合、コンパイラーは、このロードを一般的なギャザーではなく、通常のベクトルロードに変換します。

実行中のプログラム・インスタンスが、リニアに連続したメモリー位置にアクセスすることがあります。これは、ビルトインの programIndex 変数をベースに配列インデックスを使用する場合に最も頻繁に起こります。このような場合、コンパイラーもこのケースを検出してベクトルロードを行います。次の例について考えてみます。

```
for (int i = programIndex; i < count; i += programCount)
  // array[i] を処理します
```

array[i] のアクセスは、通常のベクトルロードとベクトルストアで行われます。

どちらのケースも、コンパイル時にインデックスが同じ値を持つことをコンパイラーが静的に判断できる場合です。コンパイル時には判断できなくても、実際にはそのようなケースも多くあります。この場合、標準ライブラリーの reduce_equal() 関数を利用できます。この関数は、指定された値が実行中のすべてのプログラム・インスタンスで同一であるかどうかを調査し、同じ場合は true と均一値を返します。

次の関数は、reduce_equal() を使用して、実行時にインデックスが等しいかどうかをチェックし、スカラーロードとブロードキャストを行うか、一般的なギャザーを行うか決定するコードです。

```
uniform float array[..]= { ...};
float value;
int i = ...;
uniform int ui;
if (reduce_equal(i, &ui) == true)
    value = array[ui]; // スカラーロード + ブロードキャスト
else
    value = array[i];  // ギャザー
```

上記のような単純なケースでは、reduce_equal() によるチェックのオーバーヘッドは、常にギャザーを行う場合と比較すると好ましくないかもしれません。インデックスをベースにして多くのアクセスを行うような複雑なケースでは、価値があるかもしれません。このテクニックをパフォーマンスに適したインスタンスで使用する方法については、インテル® ISPC ディストリビューションに含まれる examples/volume_rendering の例を参照してください。

メモリーリードの結合について理解する

この節は今後追加される予定です。

可能な限り 64 ビット・アドレス計算を避ける

64 ビット・アーキテクチャーのターゲットにコンパイルする場合でも、インテル® ISPC はデフォルトで 32 ビットの精度でアドレス計算を行います。この動作は、--addressing=64 コマンドライン・フラグで変更できます。このオプションは、インテル® ISPC コードで 4GB 以上のメモリーアドレスを指定する必要がある場合にのみ使用してください。これにより、生成されるコードのメモリーアドレス計算のコストがほぼ 2 倍になるためです。

8 ビットおよび 16 ビット整数タイプの計算を避ける

一般に、8 ビットと 16 ビットの整数タイプから生成されるコードは、32 ビットの整数タイプから生成されるコードほど効率が良くありません。例え最終結果が小さな整数タイプに格納されるとしても、中間の計算には 32 ビット整数タイプを使用すべきです。

効率良くリダクションを実装する

例えば、配列に格納されるすべての値を加算し、その最小値を求めるデータセットには、リダクション計算が必要になることがあります。インテル® ISPC は、このようなリダクション計算を効率良く行う機能を備えています。ただし、最高の結果を得るには、これらの機能を適切に使用する必要があります。

例えば、配列に含まれるすべての値の合計を計算するワークを考えてみます。C のコードでは次のようになります。

```
/* C による実装の sum のリダクション */
float sum(const float array[], int count) {
    float sum = 0;
    for (int i = 0; i < count; ++i)
        sum += array[i];
    return sum;
}
```

この計算はベクトル化の恩恵は得られませんが、インテル® ISPC で純粋な 均一計算として表現することもできます。

```
/* 非効率的な ispc の実装による sum のリダクション */
uniform float sum(const uniform float array[], uniform int count) {
    uniform float sum = 0;
    for (uniform int i = 0; i < count; ++i)
        sum += array[i];
    return sum;
}
```

最初の試みとして、インテル® ISPC 標準ライブラリーの reduce_add() 関数を使用してみるのも良いでしょう。これは、varying 値を受け取り、アクティブなプログラム・インスタンスのすべてにその値の合計を返します。

```
/* 非効率的な ispc の実装による sum のリダクション */
uniform float sum(const uniform float array[], uniform int count) {
    uniform float sum = 0;
    foreach (i = 0 ... count)
        sum += reduce_add(array[i]);
    return sum;
}
```

この実装では、配列からペアのプログラム・インスタンスごとに値をロードし、reduce_add() を使用してプログラム・インスタンス間でリダクションを行い合計値の更新を行います。残念ながら、この方法ではベクトル化の恩恵をほとんど得られません。値をベクトルロードすることで得られる恩恵よりも、プログラム間の reduce_add() 呼び出しで多くのワークを実行するコストが上回るためです。

最も効率の良い方法は、2 段階に分けて計算を行うことです。uniform 変数を使用して sum を格納するのではなく、varying 変数を保持することで、各プログラム・インスタンスが配列からロードした配列値のサブセットを使用してローカルの部分 sum を効率良く計算できるようにします。配列要素のループが終了すると、reduce_add() を 1 回だけ呼び出すことで、各プログラム・インスタンスの sum の要素の最終的なリダクションが行われます。この方法は、各ループ反復の値に対して 1 つのベクトルロードと加算を効率良くコンパイルして最終的に非常に高効率のコードになります。

```
/* 効率的な ispc の実装による sum のリダクション */
uniform float sum(const uniform float array[], uniform int count) {
    float sum = 0;
    foreach (i = 0 ... count)
        sum += array[i];
    return reduce_add(sum);
}
```

効率良く "foreach_active" を使用する

例えば、『インテル® ISPC ユーザーガイド』に示される foreach_active のコード例について考えてみます。

```
uniform float array[...]= { ...};
int index = ...;
foreach_active (i) {
    ++array[index];
}
```

ここで、index は複数のプログラム・インスタンスで同一値になると仮定し、index 値が競合した際に未定義の結果にならないよう、foreach_active 文で array[index] の更新をシリアル化します。

この場合、array[index] でアクセスされるのは配列の 1 つの要素のみであり、ギャザーやスキャッターを使用する必要がないことを明確にすることで、コンパイラーは生成するコードを改善できます。具体的には、標準ライブラリーの extract() 関

数を使用してプログラム・インスタンスの index 値を uniform 変数に抽出し、それを使用して以下のように array へのインデックスを作成することで、効率良いコードを生成できます。

```
foreach_active (instanceNum) {
    uniform int unifIndex = extract(index, instanceNum);
    ++array[unifIndex];
}
```

低レベルのベクトル手法の利用

インテル® ISPC コードでは、多くの低レベルのインテル® SSE およびインテル® AVX コード構造を実装できます。この場合、インテル® ISPC 標準ライブラリー関数の intbits() と floatbits() が役立ちます。intbits() は float 値を受け取り、整数として返します。整数のビットは、float のメモリー上のビット表現と同一です (つまり、整数から浮動小数点への数値的な変換は行いません)。floatbits() はその逆を行います。

これらの関数を使用した以下のコードは、指定された値の符号を効率良く反転します。

```
float flipsign(float a) {
    unsigned int i = intbits(a);
    i ^= 0x80000000;
    return floatbits(i);
}
```

このコードは単一の XOR 命令にコンパイルされます。

Fast math オプション

インテル® ISPC には、数値の精度が重要であるコードで精度に影響する多くの最適化を行う --opt=fast-math コマンドライン・フラグがあります。しかし、多くのグラフィックス・アプリケーションなどでは、それらの最適化による近似値を許容できます。--opt=fast-math が指定されると、次の 2 つの最適化が行われます。デフォルトで --opt=fast-math フラグはオフです。

- x / y のような式 (y はコンパイル時定数) は、x * (1./y) に変換され、y の逆数がコンパイル時に事前計算されます。
- x / y の式で y が定数でない場合は、x * rcp(y) に変換され、rcp() はインテル® ISPC 標準ライブラリーの近似逆数命令にマップされます。

積極的な inline 展開

関数を積極的にインライン展開することは、通常インテル® ISPC のパフォーマンスには有益です。数行程度の短い関数は、必ず inline 修飾子を使用し、長い関数では検証を行ってください。

システム数学ライブラリーを避ける

インテル® ISPC のデフォルト数学ライブラリーに含まれる超越関数は、プログラム・インスタンス間でベクトル化されていることと、最終的なコードで関数をインライン展開できるためはるかに効率的ですが、システムの数学ライブラリーよりも誤差が大きくなります (通常、システムの数学ライブラリーの超越数の誤差は 1ulp 以下ですが、インテル® ISPC 数学ライブラリーでは 10 ulp の誤差があります)。

インテル® ISPC プログラムのコンパイル時に、--math-lib=system コマンドライン・オプションを指定すると、システムの数学ライブラリーを呼び出すコードが生成されます。このオプションは、高い精度を使用するためパフォーマンスへの影響が大きいことから、高精度を必要とする場合にのみ使用してください。

使用中のスコープで変数を宣言する

最初に使用するブロックのスコープで変数を宣言すると、パフォーマンスが若干向上します。例えば、以下のコードで foo のライフタイムが if 節の範囲しかない場合は、次のように記述できます。

```
float func() {
    ....
    if (x < y) {
        float foo;
        ... foo を使用する ...
    }
}
```

次のようなコードを記述してはなりません。

```
float func() {
    float foo;
    ....
    if (x < y) {
        ... foo を使用 ...
    }
}
```

こうすることで、コンパイラーが生成するマスクが必要なストア命令の数を減らすことができます。

ランタイムの動作を理解するため インテル® ISPC プログラムをインストルメントする

インテル® ISPC には、オプションのインストルメント機能があり、パフォーマンスの問題を理解するのに役立ちます。プログラムを --instrument フラグを付けてコンパイルすると、コンパイラーは次のシグネチャーを持つ関数呼び出しを、プログラムの中のそれぞれの場所 (例えば、スキャッターやギャザーが発生した制御フローの場所) で行います。

```
extern "C" {
    void ISPCInstrument(const char *fn, const char *note,
                        int line, uint64_t mask);
}
```

この関数には、実行中のインテル® ISPC ファイル名、状況を示す短い説明、ソースファイルの行番号、およびギャング内のアクティブなプログラム・インスタンスのマスク値が渡されます。この関数を実装してアプリケーションと連携させる必要があります。

例えば、インテル® ISPC プログラムが実行される際に、この関数は次のように呼び出されることもあります。

```
ISPCInstrument("foo.ispc", "function entry", 55, 0xfull);
```

これは、実行中のプログラムが、ファイル foo.ispc の 55 行目で定義されている関数が、すべてのレーンをマスクして開始されたことを示します (4 レーンのギャングサイズのターゲットマシンを想定)。

この関数の有効性を示す詳細については、インテル® ISPC ディストリビューションに含まれる examples/aobench_instrumented を参照してください。この例には、プログラムの実行動作の集計データを収集する ISPCInstrument() 関数の実装が紹介されています。

この例を実行すると、インストルメントにかなりのオーバーヘッド・コストが掛かるため、低解像度のイメージを生成することを ao 実行ファイルに指示する必要があります。

例:

```
% ./ao 1 32 32
```

ao プログラムの実行が終了すると、次のようなサマリーレポートが出力されます。最初の行で、呼び出されたソース中の行番号と状況の説明が表示され、次の行に関数が呼び出された回数と関数のエントリーでアクティブであった SIMD レーンの平均パーセンテージが表示されます。

```
ao.ispc(0067) - function entry: 342424 calls (0 / 0.00% all off!), 95.86% active
lanes
ao.ispc(0067) - return: uniform control flow: 342424 calls (0 / 0.00% all off!),
95.86% active lanes
ao.ispc(0071) - function entry: 1122 calls (0 / 0.00% all off!), 97.33% active lanes
ao.ispc(0075) - return: uniform control flow: 1122 calls (0 / 0.00% all off!), 97.33%
active lanes
ao.ispc(0079) - function entry: 10072 calls (0 / 0.00% all off!), 45.09% active lanes
ao.ispc(0088) - function entry: 36928 calls (0 / 0.00% all off!), 97.40% active lanes
...
```

ターゲットのベクトル幅を選択

デフォルトでは、インテル® ISPC はターゲットの命令セットに従った自然なベクトル幅でコンパイルします。例えば、インテル® SSE2 やインテル® SSE4 では 4 レーン、インテル® AVX では 8 レーンでコンパイルします。プログラムによっては、2 倍のベクトル幅 (インテル® SSE では 8、インテル® AVX では 16) にコンパイルしたほうが、高いパフォーマンスを発揮することがあります。

レジスターの利用が少ないワークロードでは、この方法は、命令レベルの並列性を高め、より多くのプログラム・インスタンスでさまざまなオーバーヘッドを償却することで、実行効率を大幅に向上できます。レジスター負荷の高いワークロードでは、速度が低下する可能性があります。主要なカーネルで両方のアプローチを試す価値はあるかもしれません。

このオプションは、インテル® SSE2、インテル® SSE4、インテル® AVX のターゲットに対してのみ有効です。それぞれ、--target=sse2-x2、--target=sse4-x2、および --target=avx-x2 オプションで選択できます。

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。