インテル® Xeon Phi™ プロセッサーの最適化チュートリアル

この記事は、インテル® デベロッパー・ゾーンに公開されている「Tutorial on Intel® Xeon Phi™ Processor Optimization」の日本語参考訳です。

サンプルコード [TAR 20KB]

1. はじめに

このチュートリアルでは、インテル® Xeon Phi™ プロセッサー上で実行するアプリケーションを最適化するいくつかの可能性を示します。ここで紹介する最適化プロセスは、3 つのパートで構成されています。

- パート 1 では、コードのベクトル化 (データ並列処理) に使用される一般的な最適化手法について述べます。
- パート 2 では、スレッドレベルの並列化により、プロセッサーのすべてのコアを利用する方法を説明します。
- パート 3 では、インテル® Xeon Phi™ プロセッサー上でメモリーを最適化することで、コードを最適化します。

そして、最後に各最適化ステップによるパフォーマンスの向上をグラフに示します。

このチュートリアルでは、次の作業を行います。まず、シリアルの最適ではないサンプルコードをベースとして使用します。次に、コードにいくつかの最適化手法を適用してベクトルバージョンを生成し、ベクトルバージョンにスレッドによる並列処理を追加して並列バージョンを生成します。最後に、インテル® VTune™ Amplifier XE で並列コードのメモリー帯域幅を解析し、高帯域幅メモリーを使用することでパフォーマンスをさらに向上します。コードの 3 つのバージョン (*mySerialApp.c*、*myVectorizedApp.c*、および *myParallelApp.c*) は、この記事の最初にあるサンプルコード (*samplecode.tar*) に含まれています。

このサンプルコードは、入出力データ用に 2 つの大きなバッファーを使用するストリーミング・アプリケーションです。入力バッファーには、2 次方程式の係数が含まれます。出力バッファーには、各 2 次方程式の根を格納します。ここでは分かりやすいように、2 次方程式の実根が常に 2 つになるように係数を選択しています。

次の 2 次方程式について考えてみます。

$$ax^2 + bx + c = 0$$

2 つの根は、次の公式の解になります。

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

2 つの実根が異なるためには、次の条件を満たしていなければなりません: $a \neq 0$ および $b^2 > 4ac$

2. ハードウェアとソフトウェア

プログラムの実行には、リリース前のインテル® Xeon Phi™ プロセッサー 7250、68 コア、1.40GHz、96GB DDR4 RAM、16GB MCDRAM を使用します。コアごとに 4 つのハードウェア・スレッドがあるため、このシステムは合計 272 ハードウェア・スレッドで実行することができます。システムに Red Hat* Enterprise Linux* 7.2、インテル® Xeon Phi™ Processor Software 1.3.1、インテル® Parallel Studio XE 2016 Update 3 をインストールします。

システムに搭載されているプロセッサーのタイプと数を確認するには、/proc/cpuinfo を実行します。次に例を示します。

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 87
model name      : Intel(R) Xeon Phi(TM) CPU 7250 @ 1.40GHz
stepping        : 1
microcode       : 0xffff0180
cpu MHz         : 1515.992
cache size      : 1024 KB
physical id     : 0
siblings        : 272
core id         : 0
cpu cores       : 68
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdt
scp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc ap
erfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 fma cx16
 xtpr pdcm sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f1
6c rdrand lahf_lm abm 3dnowprefetch ida arat epb pln pts dtherm tpr_shadow vnmi
flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms avx512f rdsee
d adx avx512pf avx512er avx512cd xsaveopt
bogomips        : 2793.59
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:
……………………………………
```

出力結果から、このシステムの CPU (ハードウェア・スレッド) 数は 272 であることが分かります。flags フィールドには、インテル® Xeon Phi™ プロセッサーでサポートされる命令拡張 avx512f、avx512pf、avx512er、avx512cd が表示されています。

lscpu を実行して CPU 情報を表示することもできます。

```
$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                272
```

```
On-line CPU(s) list:    0-271
Thread(s) per core:     4
Core(s) per socket:     68
Socket(s):              1
NUMA node(s):           2
Vendor ID:              GenuineIntel
CPU family:             6
Model:                  87
Model name:             Intel(R) Xeon Phi(TM) CPU 7250 @ 1.40GHz
Stepping:               1
CPU MHz:                1365.109
BogoMIPS:               2793.59
Virtualization:         VT-x
L1d cache:              32K
L1i cache:              32K
L2 cache:               1024K
NUMA node0 CPU(s):      0-271
NUMA node1 CPU(s):
```

出力結果から、システムには 1 ソケット、68 コア、272 CPU があることが分かります。また、NUMA ノードは 2 つあり、272 CPU はすべて NUMA node 0 に属しています。NUMA に関する詳細は、「Knights Landing (開発コード名) 上の MCDRAM (高帯域メモリー) の紹介」を参照してください。

サンプルプログラムの解析と最適化を行う前に、プログラムをコンパイルし、バイナリーを実行して、ベースラインのパフォーマンスを取得します。

3. ベースライン・コードのベンチマーク

ソリューションの単純な実装が添付のプログラム *mySerialApp.c* です。係数 *a*、*b*、および *c* は *Coefficients* 構造体に、根 x_1 と x_2 は *Roots* 構造体にそれぞれ格納されます。係数と根は単精度浮動小数点数です。各係数タプルには、対応する 1 つの根タプルがあります。プログラムは、*N* 個の係数タプルと *N* 個の根タプルを割り当てます。N は大数 (512*1024*1024 = 536,870,912 = 約 512M 要素) です。Coefficients 構造体と Roots 構造体を以下に示します。

```
struct Coefficients {
        float a;
        float b;
        float c;
    } coefficients;

struct Roots {
        float x1;
        float x2;
    } roots;
```

このプログラムは、前述の公式に従って実根 x_1 と x_2 を計算します。ここでは、標準のシステムタイマーを使用して、計算時間も測定します。バッファーの割り当てと初期化にかかった時間は測定されません。このプログラムは、計算プロセスを 10 回繰り返し実行します。

インテル® C++ コンパイラーでベースライン・コードをコンパイルして、アプリケーションのベンチマークを行います。

```
$ icc mySerialApp.c
```

デフォルトでは、実行速度を最適化する -O2 コンパイラー・オプションでコンパイルされます。アプリケーションを実行します。

```
$ ./a.out
No. of Elements : 512M
Repetitions = 10
Start allocating buffers and initializing ....
SERIAL
Elapsed time in msec: 461,222 (after 10 iterations)
```

出力結果から、大きな入力 (N = 512M 要素) に対して、このシステムがデータのストリーミング、根の計算、結果の格納を 10 回繰り返すのにかかった時間は 461,222 ミリ秒であることが分かります。係数タプルごとに、プログラムは根タプルを計算します。このベースライン・コードは、シリアルおよびスカラーモードで実行する (一度に 1 スレッドが 1 つのタプル要素のみを処理する) ため、システムに搭載されている多数のコアや SIMD 命令を利用していません。つまり、1 つのハードウェア・スレッド (CPU) のみが実行し、残りのすべての CPU はアイドル状態のままです。これは、-qopt-report=5 -qopt-report-phase:vec コンパイラー・オプションを指定してベクトル化レポート (**.optrpt*) を生成することで確認できます。

```
$ icc mySerialApp.c -qopt-report=5 -qopt-report-phase:vec
```

ベースライン・コードのパフォーマンスを測定したら、コードのベクトル化を行います。

4. コードのベクトル化

4.1. 構造体配列 (AOS) から配列構造体 (SOA) への変更 (バッファーの割り当てで複数のレイヤーを使用しない)

コードのパフォーマンスを向上するため、最初に構造体配列 (AOS) から配列構造体 (SOA) へ変換します。SOA は、ユニットストライドでアクセスできるデータ量を増やします。多数の係数タプル (*a*、*b*、*c*) と根タプル (x1、x2) を定義する代わりに、データ構造を変更することで、*a*、*b*、*c*、x1、および x2 という 5 つの大きな配列にデータを割り当てることができます (変更後のプログラムは *myVectorizedApp.c* を参照)。さらに、malloc でメモリーを割り当てる代わりに、_mm_malloc を使用してデータを 64 バイト境界でアライメントします (次のセクションを参照)。

```
float *coef_a  = (float *)_mm_malloc(N * sizeof(float), 64);
float *coef_b  = (float *)_mm_malloc(N * sizeof(float), 64);
float *coef_c  = (float *)_mm_malloc(N * sizeof(float), 64);
float *root_x1 = (float *)_mm_malloc(N * sizeof(float), 64);
float *root_x2 = (float *)_mm_malloc(N * sizeof(float), 64);
```

4.2. その他の改善点: 型変換の排除、データ・アライメント

次のステップでは、不要な型変換を排除します。例えば、関数 sqrt() は倍精度の入力を受け付けます。しかし、このプログラムは入力として単精度を渡すため、コンパイラーは単精度から倍精度へ変換する必要があります。このデータ型変換を排除するには、sqrt() の代わりに sqrtf() を使用します。同様に、整数の代わりに単精度数を使用します。例えば、4 の代わりに 4.0f を使用します。4.0 (サフィックス f を付けない場合) は倍精度浮動小数点値を表し、4.0f は単精度浮動小数点値を表します。

データ・アライメントは、メモリーからデータを読み書きする必要がある場合に、データを効率良く移動できるようにします。インテル® Xeon Phi™ プロセッサーでは、インテル® Xeon Phi™ コプロセッサーと同様に、データの開始アドレスが 64 バイト境界にある場合、データの移動を最適に行うことができます。コンパイラーがベクトル化しやすいように、64 バイトでアライメントされたメモリーを割り当て、データ使用時にプラグマ/ディレクティブを

用いてメモリーアクセスがアライメントされていることをコンパイラーに伝える必要があります。ベクトル化は、データが適切にアライメントされている場合に最適に動作します。ここで言うベクトル化とは、1 つの命令で複数のデータを処理できる能力 (SIMD) を指します。

上記の例では、ヒープに割り当てられるデータをアライメントするため、_mm_malloc() と _mm_free() を使用して配列を割り当てています。_mm_malloc() は malloc() のように動作しますが、第 2 引数としてアライメント引数 (バイト単位) を受け付けます。インテル® Xeon Phi™ プロセッサーの場合、この引数の値を 64 に指定します。配列 *a* がアライメントされていることをコンパイラーに知らせるため、a を使用する前に assume_aligned(a, 64) 節を追加する必要があります。ループ内のすべての配列がアライメントされていることをコンパイラーに知らせるには、ループの前に #pragma vector aligned 節を追加します。

4.3. コンパイラー・オプションによる自動ベクトル化の使用、コンパイラー・レポートの生成、ベクトル化の無効化

ベクトル化とは、一度に複数の値に対して操作を実行可能なベクトル処理ユニット (VPU) を利用するプログラミング手法です。自動ベクトル化とは、ループのベクトル化の可能性を見つけ、ベクトル化するコンパイラーの機能を指します。インテル® コンパイラーの自動ベクトル化機能は、デフォルトの最適化レベル -O2 以上で有効になります。

例えば、*mySerialApp.c* サンプルコードをインテル® コンパイラー (icc) でコンパイルすると、コンパイラーはデフォルトでループがベクトル化可能かどうか調査します。コンパイラーは、特定のルール (ループの反復数が判明していなければならない、ループの入口と出口はそれぞれ 1 つずつでなければならない、直列型コードでなければならない、入れ子の最内ループでなければならない、など) に従ってループをベクトル化します。追加の情報を提供することで、コンパイラーがループをベクトル化しやすくなるように手助けすることができます。

コードがベクトル化されたかどうか判断するには、-qopt-report=5 -qopt-report-phase:vec コンパイラー・オプションを指定して ベクトル化レポート (**.optrpt*) を生成します。ベクトル化レポートから、各ループがベクトル化されたかどうか、およびループのベクトル化に関する簡単な説明が分かります。ベクトル化レポートオプションは -qopt-report=<n> で、n は詳細レベルを指定します。

4.4. 最適化レベル -O3 でコンパイルする

最適化レベル -O3 を指定してプログラムをコンパイルします。この最適化レベルは、デフォルトの -O2 よりも強力な実行速度の最適化を行います。

自動ベクトル化により、一度に 1 つの要素を処理する代わりに、各ループ反復でコンパイラーは 16 の単精度浮動小数点数をベクトルレジスターにパックし、ベクトル操作を実行します。

```
$ icc myVectorizedApp.c -O3 -qopt-report -qopt-report-phase:vec -o
myVectorizedApp
```

コンパイラーは、バイナリー *myVectorizedApp* およびベクトル化レポート *myVectorizedApp.optrpt* を生成します。バイナリーを実行します。

```
$ ./myVectorizedApp
No. of Elements : 512M
Repetitions = 10
Start allocating buffers and initializing ....
Elapsed time in msec: 30496 (after 10 iterations)
```

バイナリーは、1 つのスレッドのみで実行されますが、ベクトル化されています。*myVectorizedApp.optrpt* レポートは、すべての内部ループがベクトル化されたことを示しているはずです。

比較のため、-no-vec オプションを指定してプログラムをコンパイルします。

```
$ icc myVectorizedApp.c -O3 -qopt-report -qopt-report-phase:vec -o
myVectorizedApp-noVEC -no-vec
icc: リマーク #10397: 最適化レポートは出力先の *.optrpt ファイルに生成されます。
```

myVectorizedApp-noVEC バイナリーを実行します。

```
$ ./myVectorizedApp-noVEC
No. of Elements : 512M
Repetitions = 10
Start allocating buffers and initializing ....
Elapsed time in msec: 180375 (after 10 iterations)
```

この実行では、*myVectorizedApp.optrpt* レポートは、想定どおり、自動ベクトル化が無効なためループがベクトル化されなかったことを示します。

それぞれのパフォーマンスについて見てみましょう。オリジナルバージョン (461,222 ミリ秒) からベクトル化なしバージョン (180,375 ミリ秒) の向上は、基本的に一般的な最適化手法によるものです。ベクトル化なしバージョン (180,375 ミリ秒) からベクトル化ありバージョン (30,496 ミリ秒) の向上は、自動ベクトル化によるものです。

パフォーマンスは向上しましたが、1 つのスレッドのみが計算を実行していることに変わりはありません。マルチコア・アーキテクチャーの利点を活かし、複数のスレッドが並列に実行できるように、さらなる改善が可能です。

5. マルチスレッド化

5.1. スレッドレベルの並列処理: OpenMP*

インテル® Xeon Phi™ プロセッサーに搭載されている多数のコア (このシステムでは 68 コア) を利用するため、OpenMP* スレッドを並列に実行してアプリケーションをスケーリングできます。OpenMP* は、共有メモリー向けの標準 API およびプログラミング・モデルです。

OpenMP* スレッドを使用するには、ヘッダーファイル "omp.h" をインクルードし、-qopenmp オプションを指定してコードをリンクする必要があります。*myParallelApp.c* プログラムでは、for ループの前に次のディレクティブを追加します。

```
#pragma omp parallel for simd
```

for ループの前にこの pragma ディレクティブを追加することで、スレッドのチームを生成し、for ループのワークを複数のチャンクに分割するようにコンパイラーに指示します。各スレッドは、OpenMP* ランタイム・スケジュールに従ってワークのチャンクを実行します。*SIMD* 構文は、*SIMD* 命令を使用してループの複数の反復を同時に実行できることを示します。この構文は、ループにベクトル依存性が存在すると推定されてもそれを無視するようにコンパイラーに指示するため、使用には注意が必要です。

このプログラムでは、同じループでスレッド並列処理とベクトル化が行われます。各スレッドは、ループに対するそれぞれの下限から開始します。OpenMP* (スタティック・スケジューリング) の調整が適切に行われるように、並列ループの数を制限し、残りのループはシリアルに処理されるようにします。

```
#pragma omp parallel
#pragma omp master
    {
        int tid = omp_get_thread_num();
 numthreads = omp_get_num_threads();

        printf("thread num=%d\n", tid);
        printf("Initializing\r\n");

// Assuming omp static scheduling, carefully limit the loop-size to N1 instead of
N
        N1 = ((N / numthreads)/16) * numthreads * 16;
        printf("numthreads = %d, N = %d, N1 = %d, num-iters in remainder serial
loop = %d, parallel-pct = %f\n", numthreads, N, N1, N-N1, (float)N1*100.0/N);
    }
```

根を計算する関数は、次のようになります。

```
for (j=0; j<ITERATIONS; j++)
    {
#pragma omp parallel for simd
#pragma vector aligned
        for (i=0; i<serial; i++)   // Perform in parallel fashion
        {
            x1[i] = (- b[i] + sqrtf((b[i]*b[i] - 4.0f*a[i]*c[i])) ) /
(2.0f*a[i]);
            x2[i] = (- b[i] - sqrtf((b[i]*b[i] - 4.0f*a[i]*c[i])) ) /
(2.0f*a[i]);
        }

#pragma vector aligned
        for( i=serial; i<vectorSize; i++)
        {
            x1[i] = (- b[i] + sqrtf((b[i]*b[i] - 4.0f *a[i]*c[i])) ) /
(2.0f*a[i]);
            x2[i] = (- b[i] - sqrtf((b[i]*b[i] - 4.0f *a[i]*c[i])) ) /
(2.0f*a[i]);
        }
    }
```

-qopenmp を指定して、プログラムをコンパイルおよびリンクします。

```
$ icc myParallelApp.c -O3 -qopt-report=5 -qopt-report-phase:vec,openmp -o
myParallelAppl -qopenmp
```

myParallelApp.optrpt レポートを調べて、OpenMP* によりすべてのループがベクトル化および並列化されたことを確認します。

5.2. 環境変数を使用してスレッド数とアフィニティーを設定する

OpenMP* 実装は、複数のスレッドを並列に開始できます。デフォルトでは、スレッド数はシステムの最大ハードウェア・スレッド数に設定されます。このテストシステムでは、デフォルトで 272 OpenMP* スレッドが実行されます。この OpenMP* スレッド数は、OMP_NUM_THREADS 環境変数を使用して設定することもできます。例えば、次のコマンドは 68 個の OpenMP* スレッドを開始します。

```
$ export OMP_NUM_THREADS=68
```

スレッド・アフィニティー (OpenMP* スレッドを CPU にバインドする) の設定には、KMP_AFFINITY 環境変数を使用します。スレッドをシステム全体にわたって均等に分配するには、scatter を指定します。

```
$ export KMP_AFFINITY=scatter
```

システムのすべてのコアを使用し、異なるコアごとのスレッド数でプログラムを実行してみましょう。以下は、テストシステムでコアごとのスレッド数を 1、2、3、および 4 に設定して実行した結果です。

コアごとのスレッド数が 1 の場合:

```
$ export KMP_AFFINITY=scatter
$ export OMP_NUM_THREADS=68
$ ./myParallelApp
No. of Elements : 512M
Repetitions = 10
Start allocating buffers and initializing ....
thread num=0
Initializing
numthreads = 68, N = 536870912, N1 = 536870336, num-iters in remainder serial
loop = 576, parallel-pct = 99.999893
Starting Compute on 68 threads
Elapsed time in msec: 1722 (after 10 iterations)
```

コアごとのスレッド数が 2 の場合:

```
$ export OMP_NUM_THREADS=136
$ ./myParallelApp
No. of Elements : 512M
Repetitions = 10
Start allocating buffers and initializing ....
thread num=0
Initializing
numthreads = 136, N = 536870912, N1 = 536869248, num-iters in remainder serial
loop = 1664, parallel-pct = 99.999690
Starting Compute on 136 threads
Elapsed time in msec: 1781 (after 10 iterations)
```

コアごとのスレッド数が 3 の場合:

```
$ export OMP_NUM_THREADS=204
$ ./myParallelApp
No. of Elements : 512M
Repetitions = 10
Start allocating buffers and initializing ....
thread num=0
Initializing
numthreads = 204, N = 536870912, N1 = 536869248, num-iters in remainder serial
loop = 1664, parallel-pct = 99.999690
Starting Compute on 204 threads
Elapsed time in msec: 1878 (after 10 iterations)
```

コアごとのスレッド数が 4 の場合:

```
$ export OMP_NUM_THREADS=272
$ ./myParallelApp
No. of Elements : 512M
Repetitions = 10
```

```
Start allocating buffers and initializing ....
thread num=0
Initializing
numthreads = 272, N = 536870912, N1 = 536867072, num-iters in remainder serial
loop = 3840, parallel-pct = 99.999285
Starting Compute on 272 threads
Elapsed time in msec: 1940 (after 10 iterations)
```

上記の結果から、68 コアすべてを使用し、コアごとに 1 スレッドを実行した場合に最高のパフォーマンスが得られることが分かります。

6. インテル® Xeon Phi™ プロセッサー向けのコードの最適化

6.1. メモリー帯域幅の最適化

システムには 2 種類のメモリーがあります: オンパッケージの 16GB MCDRAM と従来のオンプラットフォームの 6 チャネル 96GB DDR4 RAM (最大 384GB に拡張可能)。MCDRAM の帯域幅は約 500GB/秒で、DDR4 の最大帯域幅は約 90GB/秒です。

MCDRAM には 3 つのメモリーモードがあります: フラットモード、キャッシュモード、ハイブリッド・モード。フラットモード (アドレス指定可能メモリー) では、ユーザーが明示的に MCDRAM にメモリーを割り当てることができます。キャッシュモードでは、MCDRAM 全体が L2 キャッシュと DDR4 メモリーの間の最終キャッシュとして使用されます。ハイブリッド・モードでは、MCDRAM の一部がキャッシュとして使用され、残りはアドレス指定可能メモリーとして使用されます。以下の表は、それぞれのメモリーモードの長所と短所をまとめたものです。

メモリーモード	長所	短所
フラット	• ユーザーが MCDRAM を制御して高帯域幅メモリーの利点を活かすことができる	• ユーザーは numactl を使用するか、コードを変更する必要がある
キャッシュ	• ユーザーに対して透過的 • キャッシュレベルを拡張	• DDR4 のメモリーロード/ストアのレイテンシーが増える可能性がある
ハイブリッド	• フラットモードとキャッシュモードの両方の利点が得られる	• フラットモードとキャッシュモードの両方の短所がある

NUMA (Non Uniform Memory Access) アーキテクチャーに関して、インテル® Xeon Phi™ プロセッサーは MCDRAM の設定に応じて 1 つまたは 2 つのノードと見なされます。キャッシュモードの場合は 1 NUMA ノードと見なされ、フラットモードまたはハイブリッド・モードの場合は 2 NUMA ノードと見なされます。クラスターモードでは、インテル® Xeon Phi™ プロセッサーは 8 NUMA ノードになることもありますが、ここではクラスターモードについては取り上げません。

numactl ユーティリティーでシステムの NUMA ノード表示できます。例えば、テストシステムで MCDRAM をフラットモードに設定し、“numactl -H” を実行すると、2 つの NUMA ノードが表示されます。ノード 0 は 272 CPU と 96GB DDR4 で構成され、ノード 1 は 16GB MCDRAM で構成されています。

```
$ numactl -H
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52
53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104
105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124
125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144
145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164
165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184
185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204
205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224
225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244
245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264
265 266 267 268 269 270 271
node 0 size: 98200 MB
node 0 free: 92888 MB
node 1 cpus:
node 1 size: 16384 MB
node 1 free: 15926 MB
node distances:
node 0 1
0: 10 31
1: 31 10
```

"numactl" ツールを使用して、特定の NUMA ノードにメモリーを割り当てることができます。この例では、ノード 0 にすべての CPU とオンプラットフォーム・メモリーの DDR4 があり、ノード 1 にはオンパケット・メモリーの MCDRAM があります。-m、または--membind オプションを指定することで、プログラムが特定の NUMA ノードにメモリーを割り当てるように強制することができます。

アプリケーションが DDR メモリー (ノード 0) に割り当てるように強制するには、次のコマンドを実行します。

```
$ numactl -m 0 ./myParallelApp
```

これは、次のコマンドと等価です。

```
$ ./myParallelApp
```

68 個の OpenMP* スレッドでアプリケーションを実行します。

```
$ export KMP_AFFINITY=scatter
$ export OMP_NUM_THREADS=68

$ numactl -m 0 ./myParallelApp
No. of Elements : 512M
Repetitions = 10
Start allocating buffers and initializing ....
thread num=0
Initializing
numthreads = 68, N = 536870912, N1 = 536870336, num-iters in remainder serial
loop = 576, parallel-pct = 99.999893
Starting Compute on 68 threads
Elapsed time in msec: 1730 (after 10 iterations)
```

NUMA ノードを表示する別の方法として "lstopo" コマンドがあります。このコマンドは、NUMA ノードだけでなく、それぞれのノードに関連付けられている L1 および L2 キャッシュも表示します。

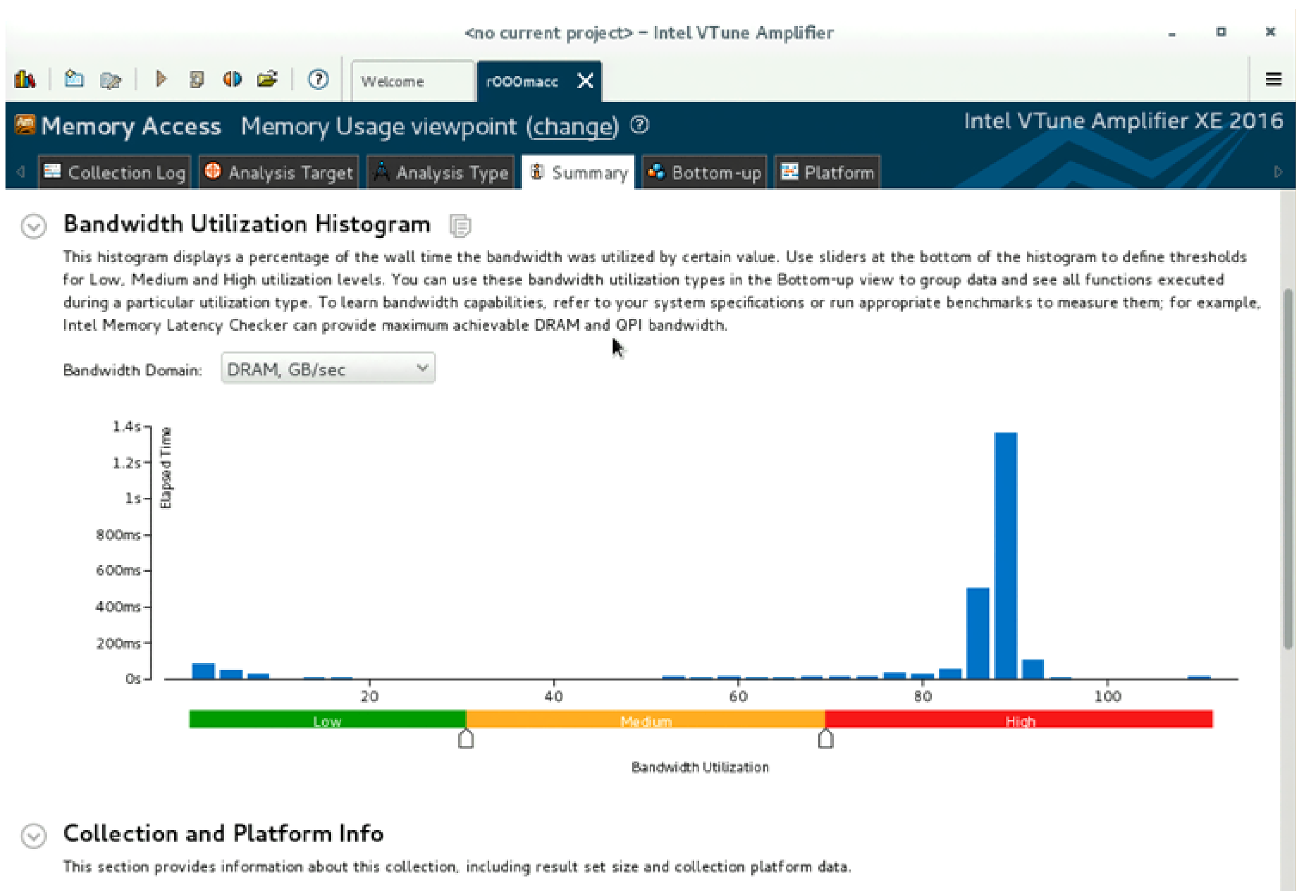
6.2. メモリー使用の解析

このアプリケーションが帯域幅に依存しているかどうか、インテル® VTune™ Amplifier XE を使用してメモリーアクセスを解析します。DDR4 DRAM の最大帯域幅は約 90GB/秒であるのに対し、MCDRAM の最大帯域幅は約 500GB/秒です。

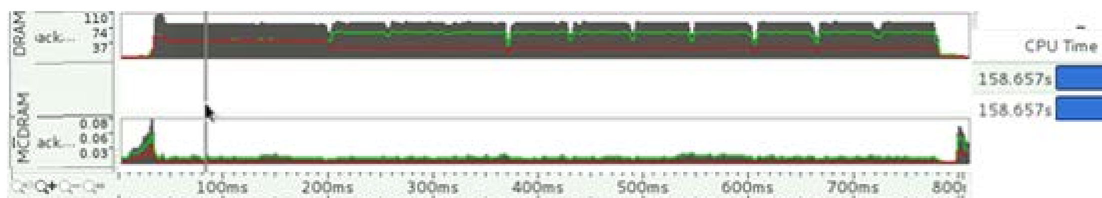
システムにインテル® VTune™ Amplifier XE をインストールし、次のコマンドを実行して、アプリケーションが DDR にメモリーを割り当てる際のメモリーアクセス情報を収集します。

```
$ export KMP_AFFINITY=scatter; export OMP_NUM_THREADS=68; amplxe-cl -collect
memory-access -- numactl -m 0 ./myParallelApp
```

[Bandwidth Utilization Histogram] フィールドにアプリケーションの帯域幅の使用状況が表示されます。ヒストグラムから、DDR 帯域幅の使用率が高いことが分かります。



メモリー・アクセス・プロファイルから、DDR4 のピーク帯域幅が最大帯域幅の 90GB/秒に近い 96GB/秒に達することが見てとれます。この結果は、アプリケーションがメモリー帯域幅に依存していることを示しています。



アプリケーションのメモリー割り当てを確認すると、512M 要素の 5 つの配列 (512 * 1024 * 1024 要素) を割り当てていることが分かります。各要素は単精度浮動小数点数 (4 バイト) なので、各配列のサイズは約 4*512M (つまり 2GB) になります。全体では、2GB * 5 = 10GB のメモリー割り当てが行われています。このサイズであれば MCDRAM (16GB) に収まるため、MCDRAM にメモリーを割り当てるフラットモードを使用することで、アプリケーションのパフォーマンスを向上できます。

MCDRAM (ノード 1) にメモリーを割り当てるには、次のように引数 -m 1 を numactl コマンドに渡します。

```
$ numactl -m 1 ./myParallelApp
No. of Elements : 512M
Repetitions = 10
Start allocating buffers and initializing ....
thread num=0
Initializing
numthreads = 68, N = 536870912, N1 = 536870336, num-iters in remainder serial
loop = 576, parallel-pct = 99.999893
Starting Compute on 68 threads
Elapsed time in msec: 498 (after 10 iterations)
```

MCDRAM にメモリーを割り当てることで、アプリケーションのパフォーマンスが大幅に向上したことが分かります。

比較のため、次のコマンドを実行して、アプリケーションが MCDRAM にメモリーを割り当てる際のメモリーアクセス情報を収集します。

```
$ export KMP_AFFINITY=scatter; export OMP_NUM_THREADS=68; amplxe-cl -collect
memory-access -- numactl -m 1 ./myParallelApp
```

ヒストグラムから、DDR 帯域幅の使用率は低く、MCDRAM 帯域幅の使用率は高いことが分かります。

<no current project> – Intel VTune Amplifier

Welcome | r001macc

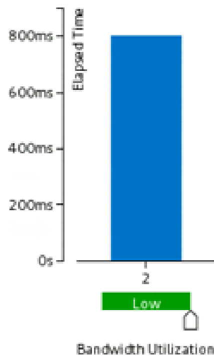
Memory Access Memory Usage viewpoint (change) — Intel VTune Amplifier XE 2016

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Platform

Bandwidth Utilization Histogram

This histogram displays a percentage of the wall time the bandwidth was utilized by certain value. Use sliders at the bottom of the histogram to define thresholds for Low, Medium and High utilization levels. You can use these bandwidth utilization types in the Bottom-up view to group data and see all functions executed during a particular utilization type. To learn bandwidth capabilities, refer to your system specifications or run appropriate benchmarks to measure them; for example, Intel Memory Latency Checker can provide maximum achievable DRAM and QPI bandwidth.

Bandwidth Domain: DRAM, GB/sec



Collection and Platform Info

This section provides information about this collection, including result set size and collection platform data.

<no current project> – Intel VTune Amplifier

Welcome | r001macc

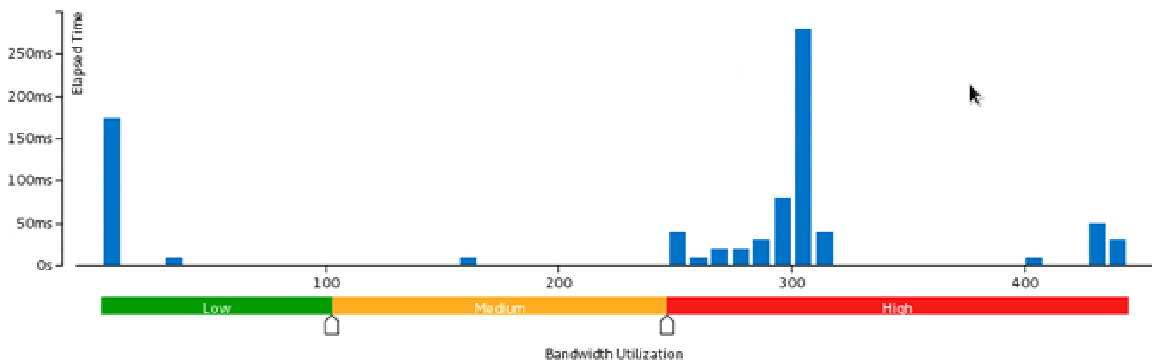
Memory Access Memory Usage viewpoint (change) — Intel VTune Amplifier XE 2016

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Platform

Bandwidth Utilization Histogram

This histogram displays a percentage of the wall time the bandwidth was utilized by certain value. Use sliders at the bottom of the histogram to define thresholds for Low, Medium and High utilization levels. You can use these bandwidth utilization types in the Bottom-up view to group data and see all functions executed during a particular utilization type. To learn bandwidth capabilities, refer to your system specifications or run appropriate benchmarks to measure them; for example, Intel Memory Latency Checker can provide maximum achievable DRAM and QPI bandwidth.

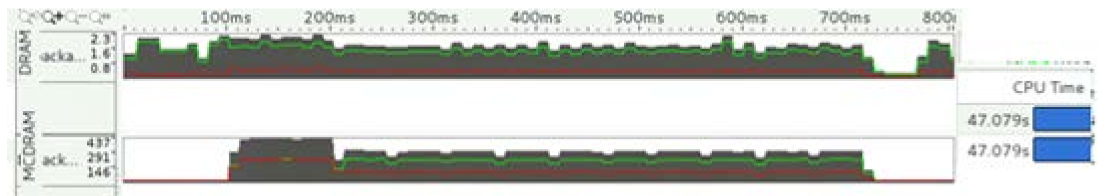
Bandwidth Domain: MCDRAM Flat, GB/sec



Collection and Platform Info

This section provides information about this collection, including result set size and collection platform data.

メモリー・アクセス・プロファイルから、DDR4 のピーク帯域幅は 2.3GB/秒に達し、MCDRAM のピーク帯域幅は 437GB/秒に達することが見てとれます。



6.3. -xMIC-AVX512 コンパイラー・オプションでコンパイルする

インテル® Xeon Phi™ プロセッサーは、x87、インテル® ストリーミング SIMD 拡張命令 (インテル® SSE)、インテル® SSE2、インテル® SSE3、インテル® ストリーミング SIMD 拡張命令 3 補足命令(インテル® SSSE3)、インテル® SSE4.1、インテル® SSE4.2、インテル® アドバンスト・ベクトル・エクステンション (インテル® AVX)、インテル® AVX2、インテル® AVX-512 命令セット・アーキテクチャー (ISA) をサポートします。インテル® トランザクショナル・シンクロナイゼーション・エクステンション (インテル® TSX) はサポートしません。

インテル® AVX-512 は、インテル® Xeon Phi™ プロセッサーで実装されました。インテル® Xeon Phi™ プロセッサーは、次のグループをサポートします: インテル® AVX-512F、インテル® AVX-512CD、インテル® AVX-512ER、およびインテル® AVX-FP。インテル® AVX-512F (インテル® AVX-512 基本命令) には、512 ビット・ベクトル・レジスター向けのインテル® AVX およびインテル® AVX2 の SIMD 命令拡張が含まれます。インテル® AVX-512CD (インテル® AVX-512 競合検出命令) は、効率良い競合検出によりさらに多くのループのベクトル化を可能にします。インテル® AVX-512ER (インテル® AVX-512 指数および逆数命令) は、基数 2 の指数関数、逆数、逆平方根を提供します。インテル® AVX-512PF (インテル® AVX-512 プリフェッチ命令) は、メモリー操作のレイテンシー軽減に役立ちます。

インテル® AVX-512 を利用するには、-xMIC-AVX512 コンパイラー・オプションを指定してプログラムをコンパイルします。

```
$ icc myParallelApp.c -o myParallelApp-AVX512 -qopenmp -O3 -xMIC-AVX512

$ export KMP_AFFINITY=scatter
$ export OMP_NUM_THREADS=68

$ numactl -m 1 ./myParallelApp-AVX512
No. of Elements : 512M
Repetitions = 10
Start allocating buffers and initializing ....
thread num=0
Initializing
numthreads = 68, N = 536870912, N1 = 536870336, num-iters in remainder serial
loop = 576, parallel-pct = 99.999893
Starting Compute on 68 threads
Elapsed time in msec: 316 (after 10 iterations)
```

次のコマンドを実行して、アセンブリー・ファイル myParallelApp.s を生成します。

```
$ icc -O3 myParallelApp.c -qopenmp -xMIC-AVX512 -S -fsource-asm
```

アセンブリー・ファイルを調べてインテル® AVX-512 ISA が生成されたことを確認します。

6.4. -no-prec-div -fp-model fast=2 最適化オプションの使用

高い精度が必要ない場合、浮動小数点数に対してより多くの最適化を提供する (ただし、安全ではない) -fp-model fast=2 オプションを指定してコンパイルすることで、浮動小数点モデルを緩和できます。コンパイラーは、より高速で精度が低い平方根と除算の実装を使用します。次に例を示します。

```
$ icc myParallelApp.c -o myParallelApp-AVX512-FAST -qopenmp -O3 -xMIC-AVX512 -no-prec-div -no-prec-sqrt -fp-model fast=2
$ export OMP_NUM_THREADS=68

$ numactl -m 1 ./myParallelApp-AVX512-FAST
No. of Elements : 512M
Repetitions = 10
Start allocating buffers and initializing ....
thread num=0
Initializing
numthreads = 68, N = 536870912, N1 = 536870336, num-iters in remainder serial loop = 576, parallel-pct = 99.999893
Starting Compute on 68 threads
Elapsed time in msec: 310 (after 10 iterations)
```

6.5. MCDRAM をキャッシュモードに設定する

BIOS 設定で MCDRAM をキャッシュモードに設定してシステムを再起動します。キャッシュモードに設定された MCDRAM は numactl ユーティリティーに対して透過的なため、numactl を使用して NUMA ノードが 1 つのみであることを確認します。

```
$ numactl -H
available: 1 nodes (0)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271
node 0 size: 98200 MB
node 0 free: 94409 MB
node distances:
node 0
0: 10
```

プログラムを再コンパイルします。

```
$ icc myParalledApp.c -o myParalledApp -qopenmp -O3 -xMIC-AVX512 -no-prec-div -no-prec-sqrt -fp-model fast=2
```

実行します。

```
$ export OMP_NUM_THREADS=68
$ ./myParalledApp-AVX512-FAST
```

```
No. of Elements : 512M
Repetitions = 10
Start allocating buffers and initializing ....
thread num=0
Initializing
numthreads = 68, N = 536870912, N1 = 536870336, num-iters in remainder serial
loop = 576, parallel-pct = 99.999893
Starting Compute on 68 threads
Elapsed time in msec: 325 (after 10 iterations)
```

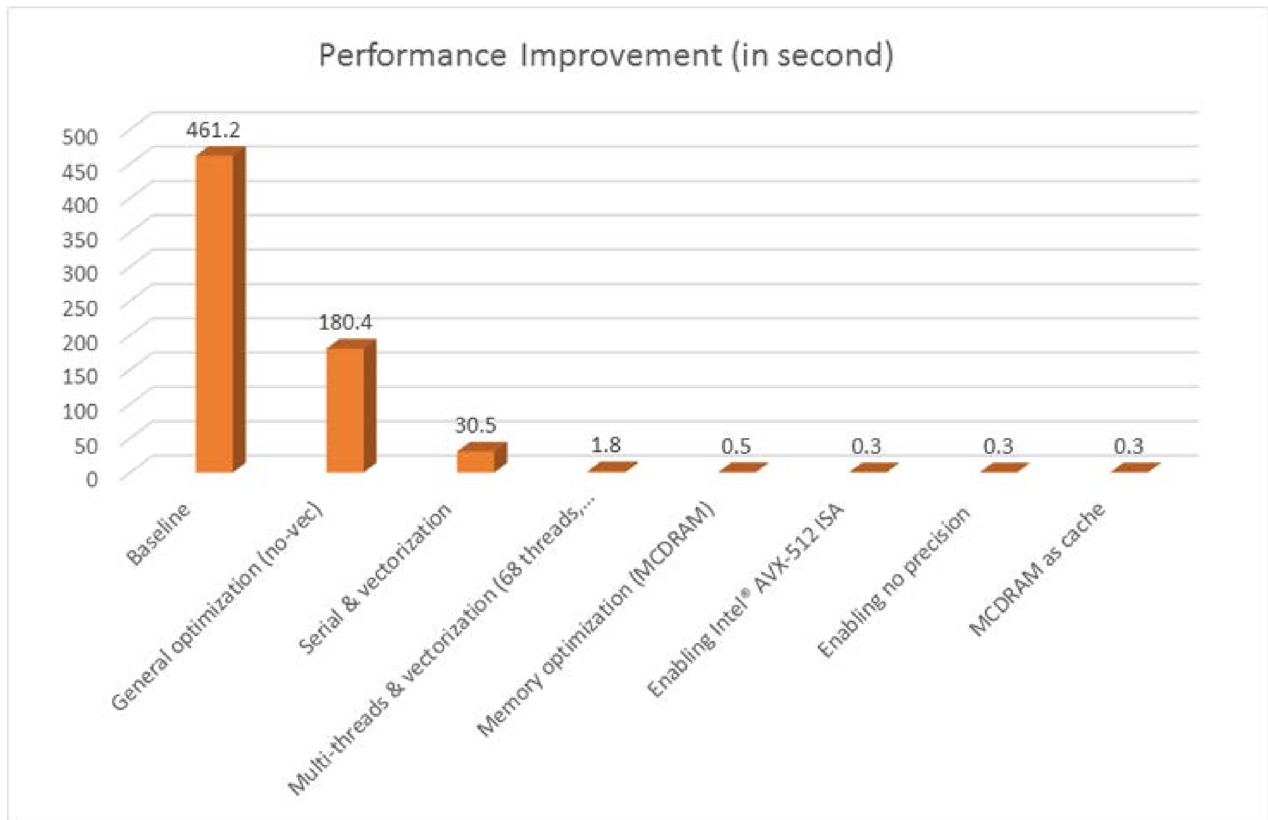
このアプリケーションでは、MCDRAM をキャッシュモードで使用しても利点がないことが分かります。

7. まとめ

このチュートリアルでは、次のトピックを説明しました。

- メモリー・アライメント
- ベクトル化
- コードの解析を支援するコンパイラー・レポートの生成
- コマンドライン・ユーティリティー cpuinfo、lscpu、numactl、lstopo の使用
- OpenMP* によるスレッドレベルの並列化
- 環境変数の設定
- インテル® VTune™ Amplifier XE による帯域幅の使用状況のプロファイル
- numactl を使用した MCDRAM メモリーの割り当て
- インテル® AVX-512 オプションを指定したコンパイルによるパフォーマンスの向上

次のグラフは、ベースライン・コードの各ステップでのパフォーマンスの向上を示します: データ・アライメントによる一般的な最適化、ベクトル化、スレッドレベルの並列化、フラットモードでの MCDRAM メモリーの割り当て、インテル® AVX-512 オプションによるコンパイル、浮動小数点演算の精度を緩和するオプションによるコンパイル、キャッシュモードでの MCDRAM の使用。



ここでは、利用可能なすべてのコア、インテル® AVX-512 のベクトル化、MCDRAM の帯域幅を利用することで、実行時間を大幅に短縮することができました。

参考文献

- インテル® Xeon Phi™ プロセッサーおよびインテル® AVX-512 ISA 向けのコンパイル (英語)
- Knights Landing (開発コード名) 上の MCDRAM (高帯域メモリー) の紹介
- インテル® C++ コンパイラーのベクトル化ガイド
- Stream Triad 上の Knights Landing (開発コード名) でのメモリー帯域幅の最適化 (英語)
- インテル® Xeon Phi™ コプロセッサー上でのストリーミング (英語)
- ベクトル化の可能性を高めるデータ・アライメント
- インテル® Xeon Phi™ コプロセッサーで最適なパフォーマンスを達成するためのメモリー管理: アライメントとプリフェッチ (英語)
- インテル® MIC アーキテクチャー向けの高度な最適化
- OpenMP* 4.0 を使用してプログラムで SIMD を有効にする
- インテル® Xeon Phi™ プロセッサー - メモリーモードとクラスターモード: 設定と使用例 (英語)

著者紹介

Loc Q Nguyen。ダラス大学で MBA を、マギル大学で電気工学の修士号を、モントリオール理工科大学で電気工学の学士号を取得しています。現在は、インテル コーポレーションのソフトウェア & サービスグループのソフトウェア・エンジニアで、コンピューター・ネットワーク、並列コンピューティング、コンピューター・グラフィックスを研究しています。

性能に関するテストに使用されるソフトウェアとワークロードは、性能がインテル® マイクロプロセッサー用に最適化されていることがあります。SYSmark* や MobileMark* などの性能テストは、特定のコンピューター・システム、コンポーネント、ソフトウェア、操作、機能に基づいて行ったものです。結果はこれらの要因によって異なります。製品の購入を検討される場合は、他の製品と組み合わせた場合の本製品の性能など、ほかの情報や性能テストも参考にして、パフォーマンスを総合的に評価することをお勧めします。詳細については、http://www.intel.com/performance (英語) を参照してください。

インテル・サンプル・ソース・コード使用許諾契約書 (英語)

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。