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「Optimization and Performance Tuning for Intel® Xeon Phi™ Coprocessors - Part 1: Optimization Essentials」の日本語参考訳です。

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 - パート 1: 最適化の基本

概要

インテル® Xeon Phi™ コプロセッサーは、インテル® プロセッサー・ファミリー/プラットフォームに新しく追加された、インテル® メニー・インテグレーテッド・コア (インテル® MIC) アーキテクチャー・ファミリーの最初の製品です。この最初のインテル® Xeon Phi™ コプロセッサーには、広い SIMD (ベクトル) レジスターを備えた多くのコアが搭載されています。新しいプロセッサー上で実行するソフトウェアは、広い SIMD (ベクトル) 演算と多くのコアを活用するべきです。この記事では、インテル® Xeon Phi™ コプロセッサーでより高速に実行できるようにソフトウェアをチューニングする方法について説明します。より詳しい情報を参照できる他の資料へのリンク集は、パート 2 で紹介します。

はじめに

インテル® Xeon Phi™ コプロセッサーは、高い計算パフォーマンスを提供します。この高いパフォーマンスを得るには、高度にスケーラブルで、高度にベクトル化され、メモリーを効率良く利用する、適切にチューニングされたソフトウェアが必要です。この記事では、インテル® Xeon Phi™ コプロセッサー向けソフトウェアのチューニングと最適化に必要な基本プロセスを開発者に紹介します。インテル® Xeon Phi™ コプロセッサー・システムのパフォーマンスに最も影響するのは、3 つの基本要素であるスケーラビリティー、ベクトル化、メモリーの使用率です。

インテル® Xeon Phi™ コプロセッサー

最初のインテル® Xeon Phi™ 製品ファミリー (開発コード名: Knights Corner) の主な特長を以下に示します。

- インテルの命令セット・アーキテクチャー (x86 インテル® アーキテクチャー命令セット) を実行可能な 57-61 個 (注: 2013 年 8 月時点) のコア
- 物理コアあたり 4 スレッド
- SIMD 演算 (ベクトル演算) 用の 512 ビット・レジスター
- コアあたり 512KB の L2 キャッシュ
- 高速な双方向リングですべてのコアを接続

コプロセッサーの物理コアはインテル® Xeon® プロセッサーよりもシンプルです。インテル® Xeon® プロセッサーのアウトオブオーダー実行モデルとは対照的に、コプロセッサーは 2 つのイ

ンオーダー実行パイプラインを備えています。アプリケーションは、インテル® MIC システムで 200 以上のアクティブなスレッドを利用するため、物理コアごとに 4 つのハードウェア・スレッドを用意して、インオーダー命令実行のレイテンシーを隠蔽することが重要になります。高い計算能力は広い 512 ビットのレジスターから得られます。インテル® Xeon Phi™ コプロセッサーでアプリケーションが、目的のパフォーマンス・レベルを達成するにはこれらの広いベクトル幅の SIMD 命令を利用する必要があります。最高のパフォーマンスは、コア、スレッド、および SIMD (ベクトル) 演算のすべてが効率良く使用された場合にのみ達成できます。コアあたり 512KB の L2 キャッシュは、合計で 25MB 以上になります。また、双方向リングによって、インテル® Xeon® プラットフォームよりも高い処理能力がもたらされます。インテル® Xeon Phi™ コプロセッサーのメインメモリーは、コプロセッサーと同じ物理カード上に搭載され、完全に独立しており、ホストシステムのメモリーとは同期されません。インテル® Xeon Phi™ コプロセッサーは、カード上で Linux* OS を実行します。開発者は、OS と開発ツールが HPC や企業顧客向けの標準的な環境であることに気付くでしょう。インテル® Xeon Phi™ コプロセッ+サーのアーキテクチャーの詳細は、http://www.isus.jp/article/mic-article/intel-xeon-phi-coprocessor-codename-knights-corner/ を参照してください。

スケーラビリティー – 多くのコアを使用

アムダールの法則

このセクションでは、アムダールの法則とグスタフソンの見解について解説し、スケーラビリティーを実現する基本的なチューニングのヒントを紹介します。ここでは「スケーラビリティー」を、プロセッサー数に反比例して実行時間を減らす能力と見なしています。例えば、完璧にスケーリングされたコードは、1000 コアでは 1 コアで実行する場合の 1/1000 の時間で実行されます。ただし、これは理論上最適なケースです。実際のモデルは、アムダールの法則により以下のように表現できます。

$$T_P = T_S(P/n + S) + OVH$$

意味:

- T_P は並列に実行する時間
- T_S はシーケンシャル (シリアル) に実行する時間
- P は、シーケンシャルに実行したときに並列領域を実行する時間の割合
- n は、ワークが分散されるプロセッサー数 (ホモジニアス・プロセッサーを仮定)
- S は、シーケンシャルに実行したときにシーケンシャル領域を実行する時間の割合
- OVH は、並列タスクの設定、同期コスト、並列処理に伴うその他の操作に関連するオーバーヘッド

アムダールの法則は期待値を設定できる点が優れています。P が 90% で S が 10% のケースを考えてみましょう。S が 10% 残っているため、この固定問題サイズに無限数のコアが適用されても、最大のスピードアップは 10 倍を超えることがありません。10 倍以上のスピードアップを達成するには、(プロセッサーまたはコア数を 10 以上にして) S を 10% 未満にする必要があります。開発者は、要件とすべてのリソースを考慮する必要があります (プロセッサー・コアは考慮

すべき 1 つのリソースにすぎません)。S が 10% の場合、開発者は 10 コア以上のケースを考慮しなくてもかまわないでしょうか? いいえ、そんなことはありません。例えば、T_S が 100 秒で、タスクを 13 秒未満で完了しなければいけない場合、31 コアあれば完了できます。31 のコアをすべて効率良く利用していなくても、設定されたパフォーマンス要件は満たされます。多くの開発者は、インテル® Xeon Phi™ コプロセッサーで、ワークの並列領域の割合 P を大きく (つまり S を小さく) 設定します。異なる開発プロジェクト、要件、目標で、S の最小値を指定しないでください。各プロジェクト・チームは、スケーラビリティーの目標と必要なレベルを定義した後、達成が可能な数値をおおまかに予測する必要があります。多くのハイパフォーマンス・コンピューティング・タスクは、単一システムのメモリーに収まらない問題を解きます。プロセッサー、コア、メモリー、インターコネクト、ディスク容量、時間など、システムやクラスターのすべてのリソースが問題を解くために利用されます。これらのケースでは、問題全体をシリアルに実行できないため、T_S は現実的な数になりません。多くのインテル® Xeon Phi™ コプロセッサーは、大規模なハイパフォーマンス・クラスターの一部として構成されます。

グスタフソンの見解

ここから導き出されたのが、グスタフソンの見解です。John Gustafson 氏は、並列に解かれる問題は、シリアルシステムで解かれる問題サイズよりも多くなることを指摘しました。*n* (コアまたはプロセッサーの数) の増加とともに問題サイズが増加し、問題サイズの増加とともに並列領域 (P) の時間の割合が増加すれば、スケーリングは増加することを述べています。アムダールの法則の短所は、P と S の変化を問題サイズの増分として見なさないことです。多くの場合、小さな問題をシーケンシャルに解くのと同じ時間内で、さらに大きな問題を並列に解くことが目標となります。グスタフソンは、問題サイズの増加とともに、並列に処理されるワークの割合がシリアル領域で費やされる時間の割合よりも増加することに気付きました。つまり、問題サイズの増加とともに、スケーリング、相対的なスピードアップ、効率が向上します。簡単な例として、部分ピボットを使用した LU 行列の因数分解について考えてみましょう。行列サイズ *n* の増加とともに、必要なメモリーの量は n^2 の次数で増加しますが、必要な計算は n^3 の次数で増加するため、この計算は並列実行に適しています。その結果、LU 行列の因数分解は、スーパーコンピューターの最も一般的な並列ベンチマークとして用いられるようになりました。アムダールの法則は無視すべきではありませんが、P は問題サイズの増加とともに増加し、スケーリングが増加することを理解するべきです。

スケーラブルな最適化のヒント: 粒度、バランス、バリア、フォルス・シェアリング

粒度

問題サイズが一定である場合、コア数の増加につれて、各コアのワークの量は減少します (“細粒度” になります)。オーバーヘッドがワーク単位で一定であれば、より小さな単位のワークで実行時のオーバーヘッドが大きくなります。つまり、粒度が小さくなると、効率は低下します。シーケンシャル・システムでこの点をチェックする 1 つの方法は、問題サイズと並列処理されるワークの量を考えることです。シーケンシャルに処理されるワークが、並列ワークロードの各コアで行われるワークの総量と一致する、より小さなシーケンシャル問題を検討してください。アムダールの法則を用いて比率を決定し、スケーリングするかどうかを調べます。

ワークロードのバランス

インテル® MIC アーキテクチャーでは、200 を超えるアクティブなスレッドが処理されます。すべてのスレッドが SIMD 命令を実行している場合、処理は最も効率良くなります。しかし、少ないスレッドのチームがより多くの計算を行う場合、残りのスレッドはこれらのスレッドが完了するまで待機しなければいけないため、効率は低下します。つまり、アプリケーションは利用可能なリソースを活用するスケーリングに失敗しています。負荷バランスの悪い 1 つのタスクが割り当てられると、これもまた多くのスレッドやプロセスがアイドル状態になり、システムのパフォーマンスは低下します。各ループ反復で全く同じ計算を実行する、反復回数が多く固定の for ループで構成されるコードは、一般的に、優れたワークロード・バランスを達成し、OpenMP* 構文で適切に処理できます。ワークロード・バランスが均一でない場合、インテル® Cilk™ Plus 並列拡張やインテル® スレッディング・ビルディング・ブロック (インテル® TBB) のワークスチールを利用すると、優れたワークロード・バランスを達成できる可能性があります。アプリケーションでワークロードのバランスが取れていない場合、これらの手法を試してください。インテル® Cilk™ Plus の手法やインテル® TBB に詳しくない方は、この機会にぜひ以下の製品情報をご覧ください。
インテル® TBB: http://www.isus.jp/article/intel-tbb
インテル® Cilk™ Plus: http://www.isus.jp/article/intel-cilk-plus

バリア

フォーク、ジョイン、mutex、ロック、バリアは効率を低下させる可能性があります。コードにデータ競合が存在しないことを保証するため、必要最低限のロック/バリアまたはコントロールを選択すべきです。ただし、データ競合が発生するリスクを冒してまでバリアを削除すべきではありません。デスクトップ・システムでインテル® Parallel Advisor XE 2013 のようなツールを利用すると、スレッド化モデルを構築する前に、シリアル領域を並列化した場合の共有データを識別できます。コードがスレッド化されたら、インテル® Inspector XE を使用して一般的なデータ競合とメモリー問題を調べます。16 スレッド間のバリアによる同期の時間は、200 以上のスレッドを同期する時間よりも少ないことを覚えておいてください。数百スレッドで実行するアプリケーションのスケーリングを達成するには、グローバルなデータへのロック/バリアを最小限にする (または融合する) ことが重要です。グローバルアドレスへの参照をロックするのではなく、各スレッドに重要なデータのローカルコピーを割り当てて、特定の場所で同期を行うことにより、パフォーマンスに重要な領域でロックを排除できる場合があります。使用するロックの種類を変更したほうが良い場合もあります。例えば、多くのスレッドがデータにアクセスするがほとんど変更しない場合、2 から 16 スレッドであれば汎用 mutex が適していますが、200 以上のスレッドには適していません。この場合、reader-writer (読み取り/書き込み) ロックが適しています。読み取りが多く、更新や変更 (書き込み) がほとんど行われない場合、複数のスレッドがデータにアクセスすることができます。インテル® TBB では、読み取り/書き込みロック、ユーザー空間最適化バリアおよびコントロールがサポートされています。

システムコールは多くの開発者がよく見落とすバリアです。スケーラビリティーに影響する、最も一般的な意図しない 2 つの呼び出しは、*malloc* と *gettimeofday* です。*malloc* を呼び出すと内部でロックされ、呼び出し元がシリアル化され、実行がシリアル化されます。スレッドが大きなメモリーブロックを割り当ててから長時間処理するのであれば、このオーバーヘッドはそれほど問題になりません。*malloc* を多く呼び出すアプリケーションは、より効率良いメモリー・アロケーターを使うことでスケーラビリティーが向上します。インテル® TBB は、この目的のためスケーラビリティー

に優れたメモリー割り当て呼び出しをサポートしています 標準の *malloc* よりも優れたメモリー割り当てを行う、サードパーティーのメモリー割り当てライブラリーを利用することもできます。また、200 以上のスレッドが同時に *gettimeofday* を呼び出した場合も、シーケンシャル領域のように動作します。1 つのスレッドのみが *gettimeofday* を呼び出すようにしてください。スレッド内のタイミング調整にはローカルタイマーを使用します。gettimeofday の呼び出しはローカル・コア・カウンターまたはグローバル・プロセッサー・カウンターを返すように設定できます。ここでは、ローカルコア tsc カウンターではなくグローバル・プロセッサー・カウンターを使用していると仮定します。バリアとなるほかのシステムコールやライブラリーの利用を最小限に抑えるか、ほかの方法を検討してください。

フォルス・シェアリング

考慮すべきもう 1 つの項目はフォルス・シェアリングです。フォルス・シェアリングは、2 つ以上の異なるコアが、同じメモリーアドレスのデータをそれぞれのキャッシュラインに格納し、キャッシュライン内の隣接するデータをそれぞれが読み書きするときに発生します。次に例を示します。

```
float a[16];
#pragma omp parallel for num_threads(16)
for (int i = 0; i < 100000; ++i)
     a[omp_get_thread_num()]+= 1.0;
```

上記の例では、配列がキャッシュライン境界で始まり、16 個のスレッドが異なる 16 個のコアで動作すると仮定しています (コンパイラーの OpenMP* 拡張 API を使用。詳細はコンパイラーのドキュメントを参照してください)。すべてのコアは異なる要素にアクセスするため、プログラムから見ると共有は行われません。しかし、キャッシュラインは 16 個の 4 バイト浮動小数点を連続して格納するため、16 のコアが 1 つのキャッシュラインに割り当てられるメモリーをアクセスします。多くのコアが各自のデータ要素を更新するために次々に書き込みを行うと、異なるコアのキャッシュと整合性を保持するため、キャッシュラインの無効化、複製が絶えず繰り返されるため、パフォーマンスは非常に悪くなります。フォルス・シェアリングは通常、キャッシュライン境界にデータをパディングするか、またはプライベート変数を用いて回避します。

並列プログラミングの参考文献:

並列プログラミングの習得に役立つ文献を以下に示します。

マルチスレッド・アプリケーション開発のためのガイド - http://www.isus.jp/article/intelguide/index/

『構造化並列プログラミング』(Structured Parallel Programming: Patterns for Efficient Computation)、Michael McCool、James Reinders、Arch Robison 共著、Morgan Kaufman、2012

『The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications』、Clay Breshears 著、O'Reilly Media、2009 (英語)

インテル® スレッディング・ビルディング・ブロック・チュートリアル (90 ページ)
www.threadingbuildingblocks.org (英語)

インテル® スレッディング・ビルディング・ブロック・デザインパターン（47 ページ）
http://jp.xlsoft.com/documents/intel/tbb/Design_Patterns.pdf

『インテル スレッディング・ビルディング・ブロック―マルチコア時代の C++並列プログラミング』(Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism)、James Reinders 著、O'Reilly Media、2007

ベクトル化 – SIMD レジスターと SIMD 演算の使用

配列表記 (アレイ・ノーテーション)

前述したように、インテル® Xeon Phi™ コプロセッサーの優れたパフォーマンスの 1 つの鍵は、512 ビットのレジスターとそれを使用した SIMD 演算です。このセクションでは、ベクトル化または 512 ビット SIMD 演算を効率良く利用してソフトウェアをチューニングするアプローチについて説明します。

既存のコードをチューニングする際に最初に行うことは、どこで時間が費やされているかを知ることです。最も計算サイクルを消費するコードのセクションは hotspot と呼ばれ、チューニングを最初に行うべき箇所です。クロック数を測定してソースコードにマップするインテル® VTune™ Amplifier XE を利用すると、最も CPU サイクルを費やしている場所が簡単に分かります。インテル® VTune™ Amplifier XE で特定された hotspot に注目し、下記の手順に従って、そのような領域で最適化が適切に行われることを確認します。

512 ビット幅の SIMD 命令を利用する最適な手法は、インテル® Cilk™ Plus や Fortran 90 で利用可能な配列表記を使用してコードを記述することです。配列表記が使われると、コンパイラーはベクトル化を行うか SIMD 命令セットを使用します。Fortran は配列表記構文を以前から採用しており、Fortran 開発者はこの Fortran90 の機能を利用できます。インテルでは、インテル® Cilk™ Plus に配列構文を追加しました。また、コンパイラーが異なる命令セットでベクトル化できるコードを記述する手法を普及させるため、インテルはインテル® Cilk™ Plus の仕様を公開しています。配列表記およびスレッド化を含むインテル® Cilk™ Plus の機能は、gcc 4.7 コンパイラーのブランチで利用できます。

インテル® Cilk™ Plus で配列または配列のセグメントを参照する構文は、[<lower bound> : <length> : <stride>] です。*lower bound* は配列の最初の要素、*length* は配列の要素の数、*stride* は配列の要素間の距離 (通常は 1) です。配列全体を処理する場合、値は省略できます。例えば、A と B が長さ *n* の 2 つの 1 次元配列で、*c* がスカラーの場合、以下の 2 つの式は等価です。

```
A[:] = c * B[:] ; // for (i=0;i<n;i++) A[i] = c * B[i] ;
A[0:n:1] = c * B[0:n:1] ; // for (i=0;i<n;i++) A[i] = c * B[i] ;
```

配列表記構文は、インテル® MIC アーキテクチャー (およびほかのインテル® プラットフォーム) でコンパイラーが SIMD 演算を効率良く利用することを保証するための推奨される手法です。

要素関数

Fortran がユーザー定義の要素関数をサポートしているように、インテル® Cilk™ Plus の C/C++ 向け並列拡張で利用できるユーザー定義の要素関数をサポートしています。要素関数は、スカラーまたは配列要素から並列に呼び出すことができる通常の関数です。要素関数が宣言されると、通常のスカラー関数と、(*for* ループから、またはベクトル入力で関数が呼び出されたときに起動する) データ並列バージョンの 2 つのコードブロックが生成されます。要素関数は、下記の例のように、宣言子に属性ベクトルを追加して定義します。関数 MyVecMult は以下のように記述できます。

```
__attribute__(vector (optional clauses))void MyVecMult(double *a,
double *b, double *c)
{ c[0] = a[0] * b[0] ; return ;}
```

は、以下のように呼び出します。

```
for (i=0;i<n;i++) MyvecMult(a[i],b[i],c[i]) ;
```

または

```
MyvecMult(a[:],b[:],c[:]);
```

ベクトル長と追加のヒントをコンパイラーに与えるには、ベクトル節を追加します。ユーザー定義要素関数は、SIMD 命令を利用できる演算を表現するために開発者が用いる手法の 1 つです。ユーザー定義要素関数にはいくつかの制限があります (スイッチ文、for ループで goto が使用できないなど。詳細は、http://software.intel.com/sites/default/files/article/181418/whitepaperonelementalfunctions.pdf (英語) を参照してください)。インテル® コンパイラーのドキュメントにも、インテル® Cilk™ Plus の配列表記に関する情報が掲載されています。

宣言子とプラグマ

多くのソフトウェアは、配列表記でベクトル化を行うように書き直す必要はありません。インテル® コンパイラーは多くの for ループと構造を自動的にベクトル化します。しかし、自動ベクトル化のみに頼るのは不十分です。効率良いベクトルコードを生成するように開発者がコンパイラーを支援する必要があります。具体的には、プラグマや宣言子を追加して、コードをベクトル化するのに必要な情報をコンパイラーに伝えます。コードを変更する前に、インテル® VTune™ Amplifier XE でパフォーマンス・データを収集し、特定された hotspot に取り組みます。さらに、コンパイル時に (–vec-report3 オプションなどを指定して) ベクトルレポートを生成します。特定された hotspot を含むソースファイルのレポートを見て、hotspot がベクトル化されていることを確認してください。ベクトル化されていない場合、配列表記を用いてコードを書き直すか、コンパイラーが効率良くベクトル化を行えるようにプラグマや宣言子を追加します。

可能であれば、最初に、特定されたすべての hotspot をコンパイラーがベクトル化できることを確認します。上記の for ループに追加する最も単純なプラグマ/宣言子は、*#pragma ivdep* (C/C++) と cDIR$ IVDEP (Fortran) です (この “c” は固定形式の Fortran コメント文字であり、自由形式では “!” を使用することに注意してください)。このプラグマ/宣言子は、潜在的または仮定されたポインターの依存関係を無視するようにコンパイラーに指示します。ポインターが独立したメモリー領域を常に逆参照することが分かっている場所にのみ、このプラグマ/宣言子を使用してください。一般に使われるプラグマ/宣言子は、#pragma simd と cDIR$ SIMD です。このプラグマ/宣言子は、可能な場合、すべての競合を無視して SIMD 演算を含むコードを生成するようにコンパイラーに許可します。一義化できない競合を無視するようにコンパイラーに指示するため、開発者はこれを使用する前に、コードの潜在的な依存関係を明らかにしておく必要があります。

効率良いベクトル化も重要です。コンパイラーにより多くの情報を伝えることは、コンパイラーがより適切にベクトル化したコードを生成するのに役立ちます。コンパイラー・レポートで、あるコード領域がベクトル化されたことを確認したら、さらに効率良くベクトル化されているかどうかをチェックします。コードが分割ロードや分割ストアを使用せず、ギャザー/スキャッター操作を避けていることを確認してください (疎行列コードのように、ギャザー/スキャッターが有効な方法でデータを格納するコードを除く)。ベクトル化においてデータのアライメントは重要です。配列を宣言するとき、例えば、__declspec(align(64)) float A[1000] のように、64 バイト・アドレスでアライメントされていることを確認します。Fortran では、cDIR$ ATTRIBUTE ALIGN 宣言子を利用します。C の動的メモリー割り当てでは _aligned_malloc() を利用します。関数またはルーチンへポインターを渡すとき、ループ中のすべてのポインター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、ループの直前に *#pragma vector aligned* を挿入します。ルーチンのすべてのループで全ポインターがアライメントされている場合、各ループの前にプラグマを挿入する代わりに、ポインターに __assume_aligned() 組込み関数を用いてコンパイラーに知らせます。Fortran では、CDIR$ ASSUME_ALIGNED 宣言子を使用します。これらの使用方法の詳細は、コンパイラーのドキュメントを参照してください。

入れ子になった 2 つのループのループカウントが小さい場合、それらのループを 1 つの for ループにマージすると、コンパイラーはより効率良くベクトル化できます。これを手動で行う開発者もいますが、宣言子/プラグマで同じことを行えるため、開発者が行う必要はありません。プラグマ/宣言子 *#pragma nounroll_and_jam* と cDIR$ NOUNROLL_AND_JAM、および *unroll_and_jam*/ と UNROLL_AND_JAM はどちらも 2 つのループを 1 つのループにマージします。後者はさらにループをアンロールします。

メモリーとキャッシュ

アドレス指定とプリフェッチ

データがメモリーからシーケンシャルなアドレス順でアクセスされた場合、コードは最も効率良く動作します。このリニア・アクセス・パターンになるようにデータ構造を変更することがあります。ベクトル化に適した一般的な変換は、構造体配列 (AoS) から配列構造体 (SoA) です。

データのアクセスは常に重要であり、プリフェッチは計算に必要なデータの取得を待つ遅延を最小限に抑えます。インテル® コンパイラーは、ベクトル化するデータを自動的にプリフェッチします。コンパイル時に明らかでないメモリー・アクセス・パターンがループに含まれる場合、プリフェッチを指定するとコードの効率が高まる可能性があります。プリフェッチを指示するには、プラグマ/宣言子を追加するか、組込み関数を使います。開発者が for ループにプリフェッチ命令を挿入することを決めた場合、通常は、ループの現在の反復ではなく、ループの将来の反復をプリフェッチするようにします。実装の詳細は、インテル® コンパイラーのドキュメントを参照してください。

ブロッキングとタイリング

データがプロセッサーのレジスターまたはキャッシュに残されている間にそのデータを再利用すると、コードをより速く実行できます。データがキャッシュから退避される前に再利用されるように操作をブロッキングまたはタイリングすることができます。配列表記を用いた 2 つの例を以下に示します。

例 1 – 大きな N のデータ再利用のためにブロッキングしない。

```
A[0:N]=B[0:N]+C[0:N];
D[0:N]=E[0:N]+A[0:N];
```

例 2 – 再利用するためにベクトル A へのアクセスをブロッキングまたはタイリング。

```
#define VLEN 4
for(i=0;i<N;i+=VLEN){
    A[i:VLEN]= B[i:VLEN]+C[i:VLEN];
    D[i:VLEN]= E[i:VLEN]+A[i:VLEN];
}
```

小さなセットのアプリケーションでは、インテル® Xeon Phi™ コプロセッサー上で n ファクター以上のスピードアップを達成します。このようなアプリケーションは「驚異的並列 (Embarrassingly Parallel)」と呼ばれ、わずかな同期とインテル® Xeon Phi™ コプロセッサーのキャッシュに収まるデータセットを使用します。これらのワークロードは、Sandy Bridge マイクロアーキテクチャー・システムではキャッシュに収まらないことがあります。その場合、メインメモリーからのデータ転送が遅延します。これらの機能は一般的ではありませんが、利用できる場合は利用することを推奨します。全体をキャッシュに残すことができるとは限らない場合でも、データ再利用のためのブロッキングやタイリングは効果的です。

一部のアプリケーションは大きなページサイズによる利点を得ることができます。アプリケーションが大きなページサイズによる利点を得られるかどうかを判断するために収集するデータについては、イベントベースのチューニング・ガイドを参照してください。

帯域幅

帯域幅の制限を受けるアプリケーションは、インテル® Xeon Phi™ コプロセッサー・システムでより高速に実行できます。この場合のスピードアップは、追加コア数の比率ではなく、インテル®

Xeon® プロセッサー (Sandy Bridge マイクロアーキテクチャー・ベース) の帯域幅を超える、インテル® Xeon Phi™ コプロセッサーの帯域幅の合計と密接に関連しています。適切なメモリー・アクセス・パターン (ユニットストライド 1) とプリフェッチにより、メモリー帯域幅を最大限に利用できます。

まとめ

インテル® Xeon Phi™ コプロセッサーには、512 ビット SIMD レジスター、SIMD ベクトル演算、50 以上のコアが搭載されています。開発者は、この新しいコプロセッサー向けにソフトウェアをチューニングするメリットに気付くでしょう。チューニングされたコードはインテル® Xeon® ホスト・プラットフォームでも適切に動作するため、開発者は単一のコードを保守するだけで済みます。この記事で説明したステップは、ソフトウェアをチューニングする最初のアプローチです。より詳細な最適化とパフォーマンスの理解については、この後のパート 2「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 - パート 2: ハードウェア・イベントの理解と使用」をお読みください。パート 2 では、イベントベースのアプローチによるチューニングを行い、重要なチューニング・ステップについて詳細に説明します。

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。