基本的な N 体シミュレーションによる インテル® Xeon Phi™ コプロセッサーのテスト

Andrey Vladimirov (スタンフォード大学)

Vadim Karpusenko (Colfax International)

2013 年 1 月 7 日

概要

インテル® Xeon Phi™ コプロセッサーは、特定の並列アプリケーションで、インテル® Xeon® プロセッサーよりも高いパフォーマンスと優れたエネルギー効率を達成できます [1]。このホワイトペーパーでは、天体物理学と生物物理学のさまざまな計算アプリケーションの基礎となる、基本的な N 体シミュレーションを使って、インテル® Xeon Phi™ コプロセッサーへの移植と最適化を調査します。一般的な C コードを使用して、ホスト・プロセッサーとコプロセッサーで N 体シミュレーションのベンチマークをテストします。テストの結果から、このシミュレーションは、1 つのインテル® Xeon Phi™ コプロセッサーで、2 つのインテル® Xeon® プロセッサー E5 シリーズの 2.3 倍から 5.4 倍の速度で動作することが分かります。パフォーマンスは、超越演算の精度設定に依存します。また、コンパイラーが C コードから生成したアセンブリー・コードを参考にして、インテル® Xeon® プロセッサー・ファミリーのデバイス向けに効率的で自動的にベクトル化される C/C++ プログラムを設計するための手法を示します。

目次

[1] Intel、インテル、Intel ロゴ、Xeon、Intel Xeon Phi は、アメリカ合衆国および / またはその他の国における Intel Corporation の商標です。

1 インテル® Xeon Phi™ コプロセッサーがもたらすもの

インテル® Xeon Phi™ コプロセッサーは、PCI Express (PCIe) デバイス形式の対称マルチプロセッサーです。Intel の James Reinders は、インテル® Xeon Phi™ コプロセッサーについて次のように述べています [2]。「インテル® Xeon Phi™ コプロセッサーは、インテル® Xeon® プロセッサー・ベースのシステムのスケーリング能力を最大限に引き出し、利用可能なプロセッサーのベクトル機能やメモリー帯域幅を最大限に活用する能力を備えた、これまでにないアプリケーションを作成できるように設計されています。」インテル® Xeon Phi™ コプロセッサーは、スタンドアロン・プロセッサーとして使用することはできません。しかし、最大 8 つのインテル® Xeon Phi™ コプロセッサーを単一シャーシで使用できます [3]。各コプロセッサーには、64 ビット x86 命令をサポートする、クロック周波数 1GHz 以上のコアが 50 以上搭載されています。コア数は、製品のモデルおよび世代により異なります。これらのインオーダー・コアは、4 ウェイ・ハイパースレッディングをサポートしているため、論理コア数は 200 以上になります。インテル® Xeon Phi™ コプロセッサーの各コアは高速な双方向リングにより相互接続され、コアの L2 キャッシュは 25MB 以上の大きなコヒーレント・キャッシュに結合されます。コプロセッサーには、6GB 以上のオンボード GDDR5 メモリーも装備されています。インテル® Xeon Phi™ コプロセッサーの速度とエネルギー効率は、そのベクトルユニットによるものです。各コアには、インテル® Initial Many-Core Instructions (インテル® IMCI) と呼ばれる新しい命令セットをサポートする、512 ビット幅の SIMD ベクトルを処理可能なベクトル演算ユニットが用意されています。インテル® IMCI には、物理モデリングや統計分析で一般的に使用される、積和演算、逆数、平方根、累乗、指数演算、その他の命令が含まれています。インテル® Xeon Phi™ コプロセッサーの理論的なピーク・パフォーマンスは、倍精度で 1 TFLOP/秒です。このパフォーマンスを、2 つのインテル® Xeon® プロセッサー (300 GFLOP/秒) と同じ消費電力で達成しています。

インテル® Xeon Phi™ コプロセッサー (およびインテル® Xeon® プロセッサー・ベースのシステム) の能力を完全に引き出すには、次の複数のレベルの並列処理を行う必要があります。

1. 複数のコプロセッサーまたは複数の計算ノードでアプリケーションをスケールする、分散メモリーのタスク並列処理。
2. 200 以上の論理コアを使用する、共有メモリーのタスク並列処理。
3. 512 ビットのベクトルユニットを使用するデータ並列処理。

インテル® Xeon Phi™ コプロセッサーの開発者にとって、インテル® Xeon® プロセッサーとインテル® Xeon Phi™ コプロセッサー間のプログラミング・モデルの継続性は目新しいものでしょう。

1. 同じ C、C++、Fortran コードから、インテル® Xeon® プロセッサー向けのアプリケーションとインテル® Xeon Phi™ コプロセッサー向けのアプリケーションをコンパイルできます。そのため、コードを再コンパイルするだけで、クロスプラットフォームの移植が可能です。
2. 同じ並列フレームワークがインテル® Xeon® プロセッサーとインテル® Xeon Phi™ コプロセッサーでサポートされています。独立したクラスターノードとして動作するインテル® Xeon Phi™ コプロセッサーと MPI により、分散メモリーまたは共有メモリーでスケーリングが可能です。OpenMP*、インテル® Cilk™ Plus、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)、その他の共有メモリー・フレームワークを使用して、プロセッサーやコプロセッサーのコア間で作業を均等に分割できます。インテル® マス・カーネル・ライブラリー (インテル® MKL) は、両方のプラットフォームに高度に最適化された標準関数を提供します。

[2] http://software.intel.com/sites/default/files/blog/337861/reindersxeonandxeonphi-v20121112a.pdf
[3] http://www.prweb.com/releases/2012/11/prweb10124930.htm

3. 同じソフトウェア開発ツールを両方のプラットフォームに使用できます。インテル® コンパイラーとインテル® VTune™ Amplifier XE によりアプリケーションのプロファイルと最適化を行い、インテル® デバッガーとインテル® Inspector によりクリティカルな問題と論理問題の診断を行うことができます。
4. 同様の最適化手法により、どちらのプラットフォームでもアプリケーションのパフォーマンスが向上します。

Colfax International は、幸いにも、インテル® Xeon Phi™ コプロセッサーを早くから利用することができました。科学アプリケーションを新しいアーキテクチャーに移植する過程を通して、インテル® Xeon® プロセッサーとインテル® Xeon Phi™ コプロセッサーはアーキテクチャーに多少の違いがあるものの、一般にどちらでも同様の最適化手法が役立つことが分かりました。このホワイトペーパーでは、有効であると判明した最適化手法のいくつかを紹介し、それぞれのアーキテクチャーのパフォーマンスに与える影響を示します。まず、最適化されていない一般的な多体問題の並列実装から始めます。次に、ユニットストライド形式のベクトル化によりパフォーマンスを向上させます。最後に、浮動小数点演算の精度を制御し、コードの単純性を損なうことなく、追加のスピードアップを達成します。

2 N 体シミュレーション: 基本的なアルゴリズム

N 体シミュレーションは、粒子間相互作用の重力または静電気の運動方程式を解く計算問題を指します。これらの問題は、天体物理学で太陽系、銀河、宇宙構造のような自己重力系の運動をモデル化するため、あるいは分子物理学で錯体分子や原子構造の動態をモデル化するために使用されます。

多体問題の粒子間相互作用の一般形態は、方程式 (1) で示されます。

$$\vec{F}_i = KC_i \sum_{j \neq i} C_j \frac{\vec{R}_j - \vec{R}_i}{\left|\vec{R}_j - \vec{R}_i\right|^3}. \tag{1}$$

ここで、$\vec{R}_i$ は、粒子の位置ベクトルです。この方程式は、ほかのすべての粒子により粒子 i に及ぼされる力、$\vec{F}_i$ を表現しています。相互作用は距離の逆 2 乗に依存します。つまり、粒子 j により粒子 i に及ぼされる力の大きさは、これらの粒子間の距離 $\left|\vec{R}_j - \vec{R}_i\right|$ の 2 乗に反比例します。重力多体問題では、K は万有引力定数 $G \approx 6.674 \cdot 10^{-11}\ \mathrm{m^3\ kg^{-1}\ s^{-2}}$、$C_i$ および C_j は i 番目と j 番目の粒子の質量 (キログラム) を表します。静電気問題では、K はクーロン定数 $k_e \approx 8.988 \cdot 10^9\ \mathrm{N\ m^2\ C^{-2}}$ (国際単位系、略称 SI)、C_i およびC_j は i 番目と j 番目の電荷 (クーロン) を表します。

ここでは、粒子の質量または電荷はすべて同じ値である ($C_i = C_j$) と仮定し、C が粒子間で異なるケースは解析しません。一般性を失うことなく、$K = 1$ および $C_i = 1$ を仮定できます。実際、これが真である単位系を選択することは常に可能です。次の式で、$\vec{R}$、$\vec{V}$ および t は、SI ではなく、この特別な単位系で計測された量です。

微分方程式の積分に前進オイラー法を使用すると、最後のシミュレーション時間ステップ Δt の粒子速度 $\vec{v}_i$ のコンポーネントは次のように表現できます。

$$v_{x,i}(t+\Delta t) = v_{x,i}(t) + \Delta t \sum_{j\neq i} \frac{(x_j - x_i)}{\left[(x_j-x_i)^2+(y_j-y_i)^2+(z_j-z_i)^2\right]^{\frac{3}{2}}}, \tag{2}$$

$$v_{y,i}(t+\Delta t) = v_{y,i}(t) + \Delta t \sum_{j\neq i} \frac{(y_j - y_i)}{\left[(x_j-x_i)^2+(y_j-y_i)^2+(z_j-z_i)^2\right]^{\frac{3}{2}}}, \tag{3}$$

$$v_{z,i}(t+\Delta t) = v_{z,i}(t) + \Delta t \sum_{j\neq i} \frac{(z_j - z_i)}{\left[(x_j-x_i)^2+(y_j-y_i)^2+(z_j-z_i)^2\right]^{\frac{3}{2}}}, \tag{4}$$

また、最後の時間ステップの座標のコンポーネントは次のように表現できます。

$$x_i(t+\Delta t) = x_i(t) + v_{x,i}(t+\Delta t)\Delta t, \tag{5}$$

$$y_i(t+\Delta t) = y_i(t) + v_{y,i}(t+\Delta t)\Delta t, \tag{6}$$

$$z_i(t+\Delta t) = z_i(t) + v_{z,i}(t+\Delta t)\Delta t. \tag{7}$$

ここで示される量は、粒子の座標と粒子の速度のコンポーネント、つまり、位置ベクトル $\vec{R}_i \equiv (x_i, y_i, z_i)$ と速度ベクトル $\vec{R}_i = \vec{V}_i \equiv (v_{x,i}, v_{y,i}, v_{z,i})$ です。

アルゴリズム (2) ～ (7) は、天体物理学や生物物理学以外のアプリケーションでも見かけることができ、いくつかのソフトウェア開発キットでテスト・ワークロードとして使用されています。このアルゴリズムは、実用的な大規模シミュレーションには効率的でなく、精度も十分でないことに注意してください。このアルゴリズムの計算量は $O(N^2)$ です。前進オイラー法を使用しているため、時間の精度は 1 次であり、大規模な時間ステップでは不安定です。実用的な科学アプリケーションでは、各粒子の重力の計算は通常、バーンズハット法のようなツリー・アルゴリズムで行われます。ツリー・アルゴリズムを利用すると、計算量を $O(N \log N)$ まで減らすことができます。前進オイラー法の代わりに陽的/陰的ルンゲクッタ法で運動方程式を解くこともできます。これらの手法は、時間の精度を 4 次に向上させ、アルゴリズムの安定性が大幅に向上します。

図 1: N 体シミュレーション仮想化のスナップショット。ビデオは、Colfax の YouTube チャネル (http://www.youtube.com/embed/KxaSEcmkGTo?rel=0) から入手できます。

このホワイトペーパーは、インテル® Xeon Phi™ コプロセッサーでのパフォーマンス最適化の調査を目的としているため、高度なアルゴリズムは取り上げていません。しかし、ここで説明するコード設計および最適化手法は、実用の高度なアルゴリズムにも適用できます。例えば、バーンズハット法では、空間はゾーンに分割され、粒子 i のカを計算するため、離れたゾーンは単一の大きな粒子として近似されます。一方、ローカルゾーンの相互作用は、ここで説明する $O(N^2)$ アルゴリズムを使用して計算されます。

3 最適化されていない N 体シミュレーションの C コード

ここでは、アルゴリズム (2) ～ (7) を C コードで実装しています。コード `nbody-1.c` をリスト 1 に示します。ベンチマーク情報をコンパイル、実行、生成するという点では、このコードで十分です。コードを 1 ページに収めて読みやすくするため、初期条件のセットアップを単純にし、出力機能を省き、仮想化を行わず、1 つの共有メモリー計算デバイスのみに制限しています。

コードは、OpenMP* フレームワークを使用して共有メモリーで並列化されています。インデックス `i` の外側の `for` ループは、各粒子に作用する力を計算し、その力に応じて粒子の運動量を変更します。このループの反復空間は、`#pragma omp` により複数のスレッドに分配されます。

インデックス `j` の内側の `for` ループは、粒子 `i` に作用する力を計算します。このループは、複数のスレッドではなく、別の方法により並列処理されます。デフォルトの最適化レベルで、コンパイラーは SIMD 並列処理を利用できるように、このループを自動的にベクトル化しようと試みます。ここで SIMD 並列処理が利用できることは明白です。`x`、`y`、`z` のすべてのセットで、同じ演算シーケンスを使用して、変数 `dx`、`dy`、`dz`、`drSquared`、`drPowerN32` の計算を個別に行うことができます。コンパイラーにより `nbody-1.c` の `j` ループがベクトル化されることは、`-vec-report3` コンパイラー・オプションで確認できます。

```c
#include <math.h>
#include <mkl_vsl.h>
#include <omp.h>
#include <stdio.h>

struct ParticleType { float x, y, z, vx, vy, vz; };

int main(const int argc, const char** argv) {
  const int nParticles = 30000, nSteps = 10; // Simulation parameters
  const float dt = 0.01f; // Particle propagation time step
  // Particle data stored as an array of structures
  ParticleType* particle = (ParticleType*) malloc(nParticles*sizeof(ParticleType));

  // Initialize particles
  VSLStreamStatePtr rnStream;  vslNewStream( &rnStream, VSL_BRNG_MT19937, 1 );
  vsRngUniform(VSL_RNG_METHOD_UNIFORM_STD, rnStream, 6*nParticles, (float*)particle, -1.0f, 1.0f);

  // Propagate particles
  printf("Propagating particles using %d threads...\n", omp_get_max_threads());
  double rate = 0.0, dRate = 0.0; // Benchmarking data
  const int skipSteps = 1; // Skip first iteration is warm-up on Xeon Phi coprocessor
  for (int step = 1; step <= nSteps; step++) {
    const double tStart = omp_get_wtime(); // Start timing

    #pragma omp parallel for schedule(dynamic)
    for (int i = 0; i < nParticles; i++) { // Parallel loop over particles that experience force
      float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f; // Components of force on particle i

      for (int j = 0; j < nParticles; j++) { // Vectorized loop over particles that exert force
        if (j != i) {
          // Newton's law of universal gravity calculation.
          const float dx = particle[j].x - particle[i].x;
          const float dy = particle[j].y - particle[i].y;
          const float dz = particle[j].z - particle[i].z;
          const float drSquared    = dx*dx + dy*dy + dz*dz;
          const float drPowerN32   = 1.0f/(drSquared*sqrtf(drSquared));

          // Reduction to calculate the net force
          Fx += dx * drPowerN32;  Fy += dy * drPowerN32;  Fz += dz * drPowerN32;
        }
      }
      // Move particles in response to the gravitational force
      particle[i].vx += dt*Fx; particle[i].vy += dt*Fy; particle[i].vz += dt*Fz;
    }
    for (int i = 0 ; i < nParticles; i++) { // Not much work, serial loop
      particle[i].x  += particle[i].vx*dt;
      particle[i].y  += particle[i].vy*dt;
      particle[i].z  += particle[i].vz*dt;
    }

    const double tEnd = omp_get_wtime(); // End timing
    if (step > skipSteps) // Collect statistics
      { rate  += 1.0/(tEnd - tStart); dRate += 1.0/((tEnd - tStart)*(tEnd-tStart)); }
    printf("Step %d: %.3f seconds\n", step, (tEnd-tStart));
  }
  rate/=(double)(nSteps-skipSteps); dRate=sqrt(dRate/(double)(nSteps-skipSteps)-rate*rate);
  printf("Average rate for iterations %d through %d: %.3f +- %.3f steps per second.\n",
        skipSteps+1, nSteps, rate, dRate);
  free(particle);
}
```

リスト 1: 最適化されていない N 体シミュレーション `nbody-1.c`

コードで使用しているデータ構造についても指摘しておく必要があります。各粒子は、派生型 `ParticleType` のオブジェクトにより表現されます。この型は、粒子の座標と速度コンポーネントを含む構造体です。この種のデータ構造は、後述するように、最適化に適した表現ではありません。しかし、オブジェクト指向のパラダイムで記述された実用コードで同様の構成を見かけたため、このデータ構造を採用しました。

インテル® ソフトウェア開発ツールでこのコードをコンパイルおよび実行するためのプロシージャーをリスト 2 に示します。ここでは、`-std=c99` (省略型 C 構文を有効にする)、`-openmp` (OpenMP* でプラグマによる並列化を有効にする)、`-mkl` (乱数生成のためにインテル® MKL をリンクする) の 3 つのコンパイラー・オプションを使用しています。このリストには、パフォーマンス結果も含まれています。パフォーマンスは、2 つのインテル® Xeon® プロセッサー E5-2680 (開発コード名: Sandy Bridge) (8 コア、動作周波数 2.7GHz (ターボ・ブースト利用時の最大周波数 3.5GHz)) を搭載した Colfax CX2265i-XP5 サーバー[44] で測定しました。ベンチマークの粒子数は N = 30000 に設定しました。N = 30000 のパフォーマンスは、1 ステップあたり T = 0.122 秒 (毎秒 8.2 ステップ) でした。

```
andrey@dublin$ icc -std=c99 -openmp -mkl -o nbody-1 nbody-1.c
andrey@dublin$ ./nbody-1
Propagating particles using 32 threads...
Step 1: 0.136 seconds
Step 2: 0.124 seconds
Step 3: 0.122 seconds
Step 4: 0.122 seconds
Step 5: 0.122 seconds
Step 6: 0.122 seconds
Step 7: 0.122 seconds
Step 8: 0.122 seconds
Step 9: 0.122 seconds
Step 10: 0.123 seconds
Average rate for iterations 2 through 10: 8.178 +- 0.049 steps per second.
andrey@dublin$
```

リスト 2: ホストシステムで `nbody-1.c` (リスト 1) をコンパイルして実行

このコードの効率を推定するため、算術演算数と、1 つのシミュレーション・ステップにかかった CPU クロックサイクル数を比較しました。`j` ループの反復あたりの算術演算数は、ループ本体の算術演算数をカウントして推定できます。この数は、$A = 15$ 以上になります (積和演算は 2 つの演算としてカウント)。これらの演算の少なくとも 1 つは超越演算 (平方根) です。2 つのインテル® Xeon® プロセッサー E5-2680 により発行される 1 秒あたりの CPU クロックサイクル数は、$C = 12 \times 8 \times 3.5 \cdot 10^9 = 5.6 \cdot 10^{10}$ を超えることはありません。ここで、8 はプロセッサーあたりの物理コア数、$3.5 \cdot 10^9$ Hz はターボ・ブースト利用時の最大周波数です。ここでは、クロックサイクルの最大数を推定するためにクロック周波数の上限を使用しました。このため、最低効率 (サイクルごとの演算数) の推定値は、次のようになります。

$$\mathrm{OPC} \gtrsim \frac{A \cdot N^2}{C \cdot T} = \frac{15 \cdot (30000)^2}{5.6 \cdot 10^{10} \cdot 0.122} = 1.98. \tag{8}$$

演算の 1 つが長いレイテンシーの超越命令であっても、この数が 1 以上になる理由は 2 つあります。まず、インテル® Xeon® プロセッサーは、命令パイプラインが深いため、サイクルあたり最大 2 つの命令を発行できます。次に、SIMD 命令により一度に複数の算術演算を行うことができます。後者は、インテル® C コンパイラーの自動ベクトル化機能により可能です。

[4] http://www.colfax-intl.com/xeonphi/CX2265i-XP5.html

4 ネイティブ・アプリケーションとしての N 体シミュレーション

インテル® Xeon Phi™ コプロセッサーでは、Linux* オペレーティング・システムが動作しているため、ネイティブ・アプリケーションを実行できます。インテル® Xeon Phi™ コプロセッサー向けネイティブ・アプリケーションは、インテル® Xeon® プロセッサーとバイナリー互換ではありません。ネイティブ実行以外にもコプロセッサーを利用する方法はあります。より複雑なアプリケーションには、オフロードモデルやハイブリッド MPI ジョブが適しています。ここでは、パフォーマンスと最適化手法の評価を目的としているため、ネイティブ実行について詳しい説明は行いません。

コプロセッサー向け実行ファイルを生成するには、`-mmic` オプションを指定してコードをコンパイルします。コンパイル後、インテル® Xeon Phi™ コプロセッサーでアプリケーションを実行するため、実行ファイルをコプロセッサーのファイルシステムにコピーします。アプリケーションによっては、ライブラリーをコプロセッサーにコピーするか NFS 共有する必要があります。N 体シミュレーション `nbody-1.c` では、Linux* の `scp` (セキュアコピー) コマンドを使用して、インテル® MKL およびインテルの OpenMP* ライブラリーのファイルをコプロセッサーにコピーします。

インテル® Xeon Phi™ コプロセッサーで N 体シミュレーションをコンパイルし実行する手順をリスト 3 に示します。より単純な手順を使用することも可能ですが (例えば、`micnativeloadex` ユーティリティーを利用するなど)、インテル® Xeon Phi™ コプロセッサーの Linux* 機能を説明するため、ここではあえてこの手順を使用しています。`scp` (セキュアコピー) コマンドでファイルを転送し、`ssh` を使用してコプロセッサー・システムにログインして、シェルからコマンドを実行します。

```
andrey@dublin$ icc -std=c99 -openmp -mkl -mmic -o nbody-1-mic nbody-1.c
andrey@dublin$ scp nbody-1-mic mic0:~/
nbody-1-mic                                   100%  217KB 217.2KB/s   00:00
andrey@dublin$ ssh mic0
andrey@dublin-mic$ ./nbody-1-mic
Propagating particles using 228 threads...
Step 1: 0.349 seconds
Step 2: 0.117 seconds
Step 3: 0.116 seconds
Step 4: 0.117 seconds
Step 5: 0.117 seconds
Step 6: 0.117 seconds
Step 7: 0.117 seconds
Step 8: 0.117 seconds
Step 9: 0.116 seconds
Step 10: 0.117 seconds
Average rate for iterations 2 through 10: 8.568 +- 0.021 steps per second.
andrey@dublin-mic$ exit
andrey@dublin$
```

リスト 3: インテル® Xeon Phi™ コプロセッサーで `nbody-1.c` (リスト 1) をコンパイルして実行

ベンチマークは、コードを再コンパイルしてコプロセッサーで実行するだけでパフォーマンスが少し向上することを示しています。シミュレーションにかかる時間は、1 ステップあたり $T = 0.117$ 秒 (毎秒 8.6 ステップ) になりました。しかし、パフォーマンスの向上は約 5% に過ぎません。再コンパイルしただけではそれほどスピードアップしない理由を説明する前に、コプロセッサーで達成されたサイクルあたりの命令数を推定してみましょう。

ベンチマークには、インテル® Xeon Phi™ コプロセッサーのエンジニアリング・サンプル SKU B1QS-3115A を使用しました。このコプロセッサーは 57 のアクティブコア (1.1GHz) を搭載しており、1 秒あたり $57 \times 1.1 \cdot 109 = 6.3 \cdot 10^{10}$ の命令を発行できます。このため、コプロセッサーで達成されるサイクルあたりの算術命令数は、次のようになります。

$$\text{OPC} \gtrsim \frac{A \cdot N^2}{C \cdot T} = \frac{15 \cdot (30000)^2}{6.3 \cdot 10^{10} \cdot 0.117} = 1.22. \tag{9}$$

コプロセッサーは SIMD 並列処理も使用します。しかし、インテル® Xeon Phi™ コプロセッサーの SIMD ベクトルレジスターが 512 ビットで、インテル® Xeon® プロセッサー E5 の AVX レジスターが 256 ビットであることを考えると、このアプリケーションは、ホスト・プロセッサーと比べてコプロセッサーで効率的ではありません。インテル® Xeon Phi™ コプロセッサーでパフォーマンスが大幅に向上しない理由は、セクション 5 で説明します。

5 最適化: ユニットストライド形式のベクトル化

インテル® Xeon Phi™ コプロセッサーのパフォーマンスを引き出す鍵は、スレッド並列処理とデータ並列処理を組み合わせることです。そこで、インテル® Xeon Phi™ コプロセッサー向けにスレッド並列の N 体シミュレーションを再コンパイルしました。コンパイラーにより自動的にベクトル化が行われましたが、パフォーマンスはあまり向上しませんでした。アプリケーションのスレッド並列部分は重要ではない (同期が含まれていない) ため、アプリケーションのデータ並列部分で行われていることを調査しました。

リスト 1 の内側の `j` ループを詳しく調査したところ、データ・アクセス・パターンに重要な特性が欠けていることがわかりました。その特性とは、ユニットストライド形式のデータアクセスです。量 `particle[j].x` の使用について考えてみましょう。プロセッサー・コアまたはコプロセッサー・コアが、`j=0` から `j=15` の反復でデータ並列計算を行おうとしていると仮定します。演算 `dx = particle[j].x` 、`particle[i].x` にベクトル命令を適用するには、`particle[0].x`、`particle[1].x`、 …、`particle[15].x` をベクトルレジスター (レジスターファイルの 512 バイトの連続する領域) にロードしなければいけません。しかし、ロードされるデータはアプリケーションのメモリー空間では連続していません。`particle[0].x` と `particle[1].x` 間の距離は `sizeof(ParticleType)=24` バイトです。このデータ・アクセス・パターンは非効率的です。各メモリーアクセスで、コアは 1 バイトではなく、キャッシュ全体 (64 バイト幅) をロードします。反復 `j=0` と `j=1` の `x` 座標をロードするには、コアは同じキャッシュラインを 2 回、または 2 つの異なるキャッシュラインをロードしなければいけません。その結果、不要な命令やキャッシュミスが発生し、合理的なデータ・アクセス・パターンを達成できません。より便利なデータ・アクセス・パターンは、ユニットストライド 形式のアクセスです。この形式では、連続する反復のスカラーが連続するメモリー位置に配置されるため、コアは 1 つの命令でメモリー (またはキャッシュ) からベクトルレジスターに 64 バイトのデータ全体をロードすることができます。これは、1 回のメモリーアクセスで `particle[0].x` から `particle[15].x` をロードすることに相当します。

N 体シミュレーションでユニットストライド形式のアクセスを行うには、データ格納構造の設計を変更する必要があります。通常、ユニットストライド形式のアクセスには、構造体配列の代わりに配列構造体を使用します。これは、抽象化やオブジェクト指向プログラミングの目指す方向とは矛盾しますが、マルチコアやメニーコア並列プロセッサーに適切なデータ構造を提供しないことは、パフォーマンスの大幅な損失につながります。

リスト 4 は、N 体シミュレーションの最適化された実装を示しています。この新しいコードでは、粒子の座標と速度の `x`、`y`、`z` コンポーネントをすべて配列に格納し、6 つの配列をまとめて 1 つの `ParticleSystemType` 型の構造体にしています。これにより、内側の `j` ループのデータ・アクセス・パターンはユニットストライド形式になり、`p.x[0]` と `p.x[1]` がメモリーで連続するようになります。データのアライメントが正しい場合、コプロセッサー・コアは 1 回のメモリーアクセスで `p.x[0]` から `p.x[15]` をベクトルレジスターにロードすることができます。ユニットストライド形式のデータ構造により、メモリーからキャッシュに粒子データをプリフェッチするタスクも単純化されます。データ・アライメントにより `p.x[0]` がキャッシュラインの先頭に配置されている場合でも、コンパイラーは、ループの先頭のいくつかの反復をピールし、ほとんどの反復空間でアライメントされたメモリーアクセスを使

用するコードパスを実装します。ピールされた反復はスカラー命令で (つまり、ベクトル化なしで) 実行されます。

リスト 5 は、ホスト・プロセッサーとインテル® Xeon Phi™ コプロセッサーのパフォーマンス結果を示しています。ホストのパフォーマンスは毎秒 11.5 ステップ、コプロセッサーのパフォーマンスは毎秒 29.5 ステップになり、最適化されていないバージョンと比べて、ホストで 40%、コプロセッサーで 140% パフォーマンスが向上しました。コプロセッサー・バージョンは、ホストバージョンの 2.3 倍以上高速です。2 つのインテル® Xeon® プロセッサーのパワー・エンベロープが 1 つのインテル® Xeon Phi™ コプロセッサーとほぼ同じであることを考えると、これは非常に大きなスピードアップと言えます。

演算精度の要件を緩和してコプロセッサーのパフォーマンスをさらに向上させることもできます。この最適化については、セクション 6 で説明します。

```
#include <math.h>
#include <mkl_vsl.h>
#include <omp.h>
#include <stdio.h>

struct ParticleSystemType { float *x, *y, *z, *vx, *vy, *vz; };

int main(const int argc, const char** argv) {
  const int nParticles = 30000, nSteps = 10; // Simulation parameters
  const float dt = 0.01f; // Particle propagation time step
  ParticleSystemType p; // Particle system stored as a structure of arrays
  float *buf = (float*) malloc(6*nParticles*sizeof(float)); // Malloc all data
  p.x  = buf+0*nParticles; p.y  = buf+1*nParticles; p.z  = buf+2*nParticles;
  p.vx = buf+3*nParticles; p.vy = buf+4*nParticles; p.vz = buf+5*nParticles;

  // Initialize particles
  VSLStreamStatePtr rnStream;  vslNewStream( &rnStream, VSL_BRNG_MT19937, 1 );
  vsRngUniform(VSL_RNG_METHOD_UNIFORM_STD, rnStream, 6*nParticles, buf, -1.0f, 1.0f);

  // Propagate particles
  printf("Propagating particles using %d threads...\n", omp_get_max_threads());
  double rate = 0.0, dRate = 0.0; // Benchmarking data
  const int skipSteps = 1; // Skip first iteration is warm-up on Xeon Phi coprocessor
  for (int step = 1; step <= nSteps; step++) {
    const double tStart = omp_get_wtime(); // Start timing

    #pragma omp parallel for schedule(dynamic)
    for (int i = 0; i < nParticles; i++) { // Parallel loop over particles that experience force
      float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f; // Components of force on particle i

      for (int j = 0; j < nParticles; j++) { // Vectorized loop over particles that exert force
        if (j != i) {
          // Newton's law of universal gravity calculation.
          const float dx = p.x[j] - p.x[i];
          const float dy = p.y[j] - p.y[i];
          const float dz = p.z[j] - p.z[i];
          const float drSquared    = dx*dx + dy*dy + dz*dz;
          const float drPowerN32   = 1.0f/(drSquared*sqrtf(drSquared));

          // Reduction to calculate the net force
          Fx += dx * drPowerN32;  Fy += dy * drPowerN32;  Fz += dz * drPowerN32;
        }
      }
      // Move particles in response to the gravitational force
      p.vx[i] += dt*Fx;         p.vy[i] += dt*Fy;         p.vz[i] += dt*Fz;
    }
    for (int i = 0; i < nParticles; i++) { // Not much work, serial loop
      p.x [i] += p.vx[i]*dt; p.y [i] += p.vy[i]*dt; p.z [i] += p.vz[i]*dt;
    }

    const double tEnd = omp_get_wtime(); // End timing
    if (step > skipSteps) // Collect statistics
      { rate  += 1.0/(tEnd - tStart); dRate += 1.0/((tEnd - tStart)*(tEnd-tStart)); }
    printf("Step %d: %.3f seconds\n", step, (tEnd-tStart));
  }
  rate/=(double)(nSteps-skipSteps); dRate=sqrt(dRate/(double)(nSteps-skipSteps)-rate*rate);
  printf("Average rate for iterations %d through %d: %.3f +- %.3f steps per second.\n",
         skipSteps+1, nSteps, rate, dRate);
  free(buf);
}
```

リスト 4: ユニットストライド形式のデータアクセスで最適化された N 体シミュレーション `nbody-2.c`

```
andrey@dublin$ # First, compile and run the optimized simulation on host processors
andrey@dublin$ icpc -openmp -mkl -o nbody-2 nbody-2.c
andrey@dublin$ ./nbody-2
Propagating particles using 32 threads...
Step 1: 0.099 seconds
Step 2: 0.090 seconds
Step 3: 0.086 seconds
Step 4: 0.090 seconds
Step 5: 0.086 seconds
Step 6: 0.087 seconds
Step 7: 0.086 seconds
Step 8: 0.086 seconds
Step 9: 0.086 seconds
Step 10: 0.088 seconds
Average rate for iterations 2 through 10: 11.473 +- 0.211 steps per second.
andrey@dublin$
andrey@dublin$ # Now compile and run a native version for Xeon Phi coprocessors
andrey@dublin$ icc -std=c99 -openmp -mkl -mmic -o nbody-2-mic nbody-2.c
andrey@dublin$ scp nbody-2-mic mic0:~/
nbody-1-mic                                   100%  217KB 217.2KB/s   00:00
andrey@dublin$ ssh mic0
andrey@dublin-mic$ ./nbody-2-mic
Propagating particles using 228 threads...
Step 1: 0.262 seconds
Step 2: 0.034 seconds
Step 3: 0.034 seconds
Step 4: 0.034 seconds
Step 5: 0.034 seconds
Step 6: 0.034 seconds
Step 7: 0.034 seconds
Step 8: 0.034 seconds
Step 9: 0.034 seconds
Step 10: 0.034 seconds
Average rate for iterations 2 through 10: 29.502 +- 0.260 steps per second.
andrey@dublin-mic$ exit
andrey@dublin$
```

リスト 5: ホストシステムとインテル® Xeon Phi™ コプロセッサーで `nbody-2.c` (リスト 4) をコンパイルして実行

6 最適化: 精度の制御

リスト 4 の N 体シミュレーションは、逆数平方根超越関数がボトルネックになっています。コードをリスト 5 のようにコンパイルすると、コンパイラーはこの関数の SVML (Short Vector Math Library) 実装を呼び出します。ここでは、インテル® VTune™ Amplifier XE を使用して、コプロセッサーで N 体シミュレーションを実行したときの Lightweight Hotspots 解析を行いました。累計イベントカウントに対する初期化オーバーヘッドの影響を抑えるため、シミュレーションを 10 ステップではなく 1000 ステップで実行しました。結果を図 2 に示します。サマリーパネルは、CPU 時間のほぼ半分が `_svml_invsqrtf16` 関数で費やされていることを示しています。

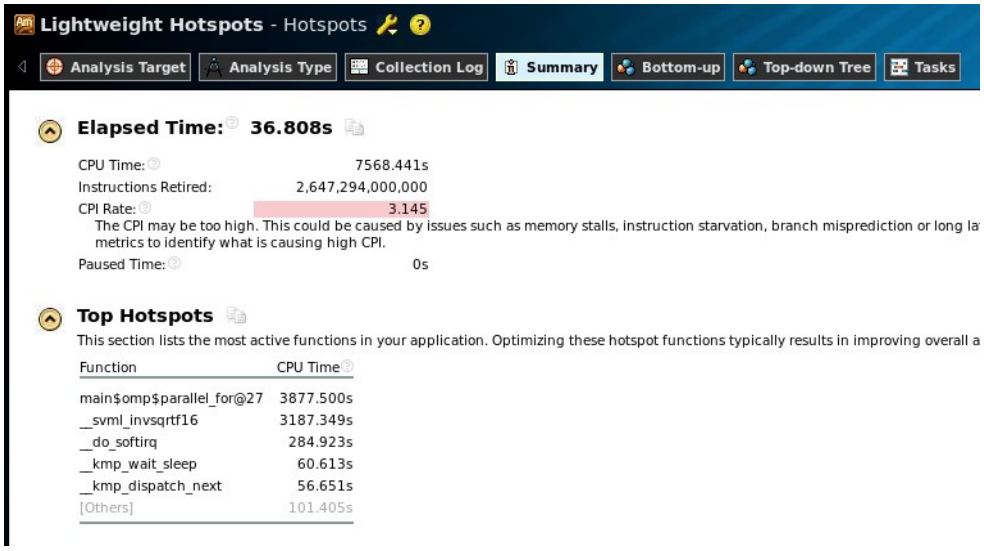


図 2: 反復回数を 1000 回にしてリスト 4 のコードをコプロセッサーで実行した場合のプロファイル結果

この N 体シミュレーションのような超越演算を含む計算集約型のコードでは、精度の制御がインテル® Xeon Phi™ コプロセッサーのパフォーマンスに大きく影響します。インテル® IMCI 命令セットには、特定の超越関数の特別な実装が含まれています。アプリケーションでいくつかの IEEE 754 浮動小数点演算の要件を安全に無視できる場合、これらの超越ベクトル命令を使用することができます。これらの命令を使用すると、SVML の IEEE 754 準拠の超越関数を実装した場合と比べて、パフォーマンスが向上します。

N 体シミュレーションで、IEEE 754 非準拠の超越関数を実装するために必要なのは、`-fimf-domain-exclusion=8` コンパイラー・オプションを追加することだけです。このオプションは、デノーマル数の処理に精度を求めないことを意味します。デノーマル浮動小数点数とは、正規数でアンダーフローになるゼロ周辺の値で、その仮数部は 1 つ以上の上位ビットがゼロになります。この最適化の結果をリスト 6 に示します。ホストのパフォーマンスは毎秒 11.6 ステップとほぼ同じですが、コプロセッサーのパフォーマンスは毎秒 62.9 ステップと、ホストの約 5.4 倍高速になりました。

インテル® Xeon Phi™ コプロセッサーでの精度の制御に関する詳細は、「Advanced Optimizations for Intel® MIC Architecture, Low Precision Optimizations」[5] (英語) を参照してください。

[5] http://software.intel.com/en-us/articles/advanced-optimizations-for-intel-mic-architecture-low-precision-optimizations

```
andrey@dublin$ # First, compile and run the optimized simulation on host processors
andrey@dublin$ icpc -openmp -mkl -fimf-domain-exclusion=8 -o nb-2a nbody-2.c
andrey@dublin$ ./nb-2a
Propagating particles using 32 threads...
Step 1: 0.104 seconds
Step 2: 0.097 seconds
Step 3: 0.085 seconds
Step 4: 0.085 seconds
Step 5: 0.085 seconds
Step 6: 0.085 seconds
Step 7: 0.085 seconds
Step 8: 0.085 seconds
Step 9: 0.085 seconds
Step 10: 0.085 seconds
Average rate for iterations 2 through 10: 11.604 +- 0.446 steps per second.
andrey@dublin$
andrey@dublin$ # Now compile and run a native version for Xeon Phi coprocessors
andrey@dublin$ icpc -openmp -mkl -fimf-domain-exclusion=8 -mmic -o nb-2a-mic nbody-2.c
andrey@dublin$ scp nb-2a-mic mic0:~/
nb-2-a-mic                                    100%  217KB 217.2KB/s   00:00
andrey@dublin$ ssh mic0
andrey@dublin-mic$ ./nb-2a-mic
Propagating particles using 228 threads...
Step 1: 0.254 seconds
Step 2: 0.016 seconds
Step 3: 0.016 seconds
Step 4: 0.016 seconds
Step 5: 0.016 seconds
Step 6: 0.016 seconds
Step 7: 0.016 seconds
Step 8: 0.016 seconds
Step 9: 0.016 seconds
Step 10: 0.016 seconds
Average rate for iterations 2 through 10: 62.944 +- 0.963 steps per second.
andrey@dublin-mic$ exit
```

リスト 6: 浮動小数点演算の精度を緩和してホストシステムとインテル® Xeon Phi™ コプロセッサーで `nbody-2.c` (リスト 4) をコンパイルして実行

7 自動的にベクトル化されたコードのアセンブリー・リストの確認

このセクションでは、N 体シミュレーションの C コードをコンパイルして生成されるアセンブリー・コードについて説明します。また、インテル® コンパイラーで使用される最適化手法にも触れます。この「アセンブリー・コード」についての知識は、高水準プログラミング言語でパフォーマンス・クリティカルなアプリケーションを設計する際に役立ちます。リスト 4 のコードを (セクション 6 で説明したように) 算術精度を緩和してコンパイルした、インテル® Xeon Phi™ コプロセッサー向けネイティブ・アプリケーションのアセンブリー・コードについて考えます。

-S オプションを指定してコンパイルすることで、完全なアセンブリー・リストを含むファイルを生成することができます。通常、アセンブリー・リストからコードのパフォーマンス・クリティカルな部分に相当する行を特定することは困難です。これは、コンパイラーがいくつかのループで複数のバージョンを生成するためです。これらのバージョンは、異なる問題サイズ N と異なるランタイム・メモリー・アライメントに対応しています。例えば、コンパイラーは、問題サイズがベクトル化には小さすぎる場合に実行するループのスカラーバージョンを生成します。また、ループカウントがベクトルレジスターのサイズの倍数でない場合、または最初のデータ要素のメモリーアドレスが適切にアライメントされていない場合、コンパイラーは、ループの先頭または最後のいくつかの反復をピールします。このように複数のバージョンが存在する場合であっても、インテル® VTune™ Amplifier XE を利用することで、コードの各行の実行にかかる CPU クロックサイクル数を測定し、ランタイムに実行されるコードパスを特定できます。

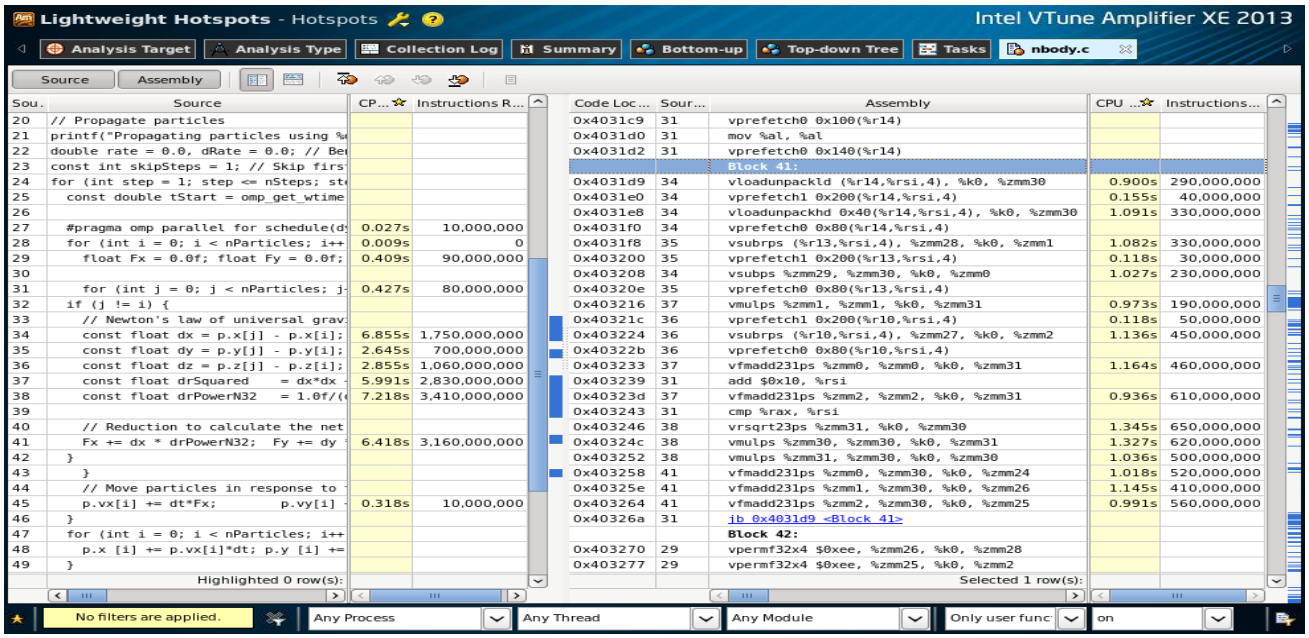
図 3 は、インテル® VTune™ Amplifier XE インターフェイスの 2 つのスクリーンショットです。どちらのスクリーンショットにも、計算負荷の最も高い j ループの本体が含まれています。図 3 では、インテル® C コンパイラーの多くの機能が明らかになっています。

a) j ループの分岐 “if (j != i)” を処理するため、実行時間がほぼ同じ 2 つのブロック (スクリーンショットのブロック 41 と 95) を生成しています。1 つのブロックは、ループの先頭から条件 (j == i) に近いポイントまで、ベクトル化された計算を実行します。もう 1 つのブロックは、ループの残りを実行します。2 つのブロックに分割することで、すべての反復で分岐条件を確認する必要をなくし、ベクトル命令を使用できるようにしています。
b) “if (j != i)” 条件を処理するため、また、N の任意の値に対応するため、コンパイラーはピーリングを行います。ピーリングとは、ループの先頭から、またはループの最後からいくつかの反復を切り離すことです。ピーリングの目的は、(a) ループの反復数がベクトルレジスターのサイズの倍数になるようにすること、および (b) データ配列が適切にアライメントされたメモリーアドレスで始まるようにすることです。
c) コンパイラーは、プリフェッチ命令を挿入します (例えば、コード位置 0x4031e0 の vprefetch1)。これらの命令は、メモリーからキャッシュ、あるいは高レベルキャッシュから低レベルキャッシュへのデータ転送をリクエストします。プリフェッチの距離 (つまり、どのくらい前のループ反復でプリフェッチを発行するか) は、コンパイラーにより自動的に判断されます。
d) ベクトル命令は、コードのベクトル化された領域で使用されます。例えば、コード位置 0x403208 では、インテル® IMCI 命令セットの SIMD 減算命令 vsubps が使用されています。
e) コンパイラーには、最適化されたベクトル命令を使用するため、算術式を組み合わせる機能があります。例えば、コード位置 0x403233 の積和演算命令 vfmadd231ps はコード行 37 に対応しています。別の例として、コード位置 0x403246 では vrsqrt23ps 命令が使用されています。この命令は、指定された引数の逆数平方根の SIMD 計算を行います。この命令とその後の 2 つの乗算命令 vmulps は、drPowerN32 = 1.0f/(drSquared*sqrtf(drSquared)); の値を計算します。ここでは、コンパイラーによって、3 つの命令 (平方根、乗算、逆数) がより効率的な 3 つの命令 (逆数平方根と 2 つの乗算) に置換されています。
f) リダクションは、合理的な方法でコンパイラーにより実装されます。コード位置 0x403258、0x40325e、0x403264 で、コンパイラーはベクトルレジスターの変数 Fx、Fy、Fz のリダクション用にデータを集計しています。これらのベクトルレジスターは、ループ計算の最後でスカラーにリダクションされます。スカラーへのリダクションは、スクリーンショットには含まれていません。

上記の最適化はすべて、インテル® MIC アーキテクチャー向けの自動ベクトル化や最適化の過程で、インテル® コンパイラーが複雑かつ効果的な決定を下していることを示しています。実際、自動的にベクトル化されたコードのパフォーマンスは非常に優れています。一般的なソースコードから生成されるインテル® Xeon® プロセッサー向けアプリケーションとインテル® MIC アーキテクチャー向けアプリケーションは、それぞれのプラットフォームの能力の上限に達することが可能です。また、ホストシステムと比べて、インテル® Xeon Phi™ コプロセッサーの N 体シミュレーションのパフォーマンスがより優れていることからも明らかです。

しかし、コンパイラーが最適化できる高水準言語コードを用意するのはプログラマーの責任です。この作業には、ユニットストライド形式のアクセスが可能なデータ構造の設計も含まれます。空間と時間のデータの局所性も一般に有効であり、タイリングやキャッシュを意識しない再帰アルゴリズムを使用して改善できます。プログラマーは、変数、定数、関数の型や精度に注意しなければいけません。例えば、N 体シミュレーションでは sqrt() の代わりに sqrtf() を使用することが非常に重要です。sqrtf() 関数は高度に最適化された単精度平方根関数を呼び出します。また、単精度の定数には文字 “f” を付けます (例えば、“1.0f”)。

そうでない場合、倍精度の定数として扱われます。型キャストはコストがかかるため、算術式で異なる型の変数を組み合わせないようにしてください。



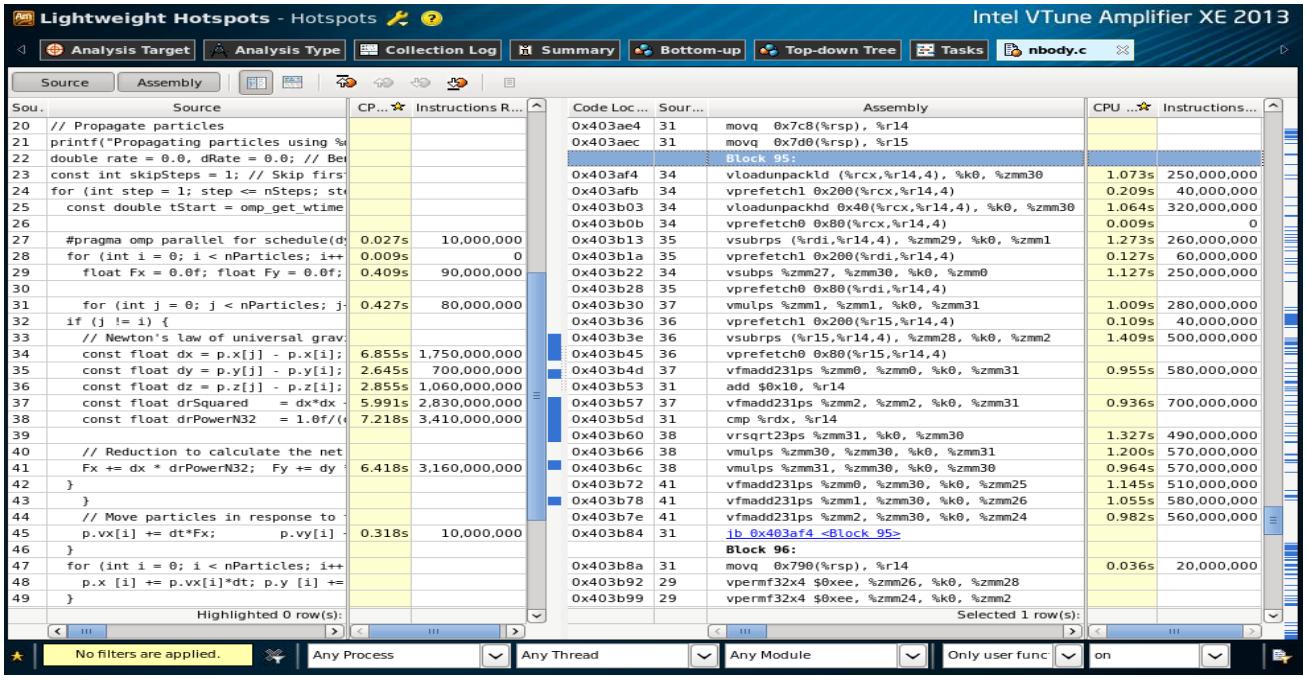


図 3: インテル® Xeon Phi™ コプロセッサーでのネイティブ実行用に演算の精度を緩和してコンパイルされたコード nbody-2.c のアセンブリー・リスト (コードの詳細はセクション 6、アセンブリー・リストの説明はセクション 7 を参照)。

8 まとめ

図 4 は、ここで説明した次の 3 つのケースのパフォーマンス結果を要約したものです。

1) 非ユニットストライド形式のデータアクセスを使用する最適化されていない並列コード

2) ユニットストライド形式のデータアクセスを使用する最適化された並列コード、

3) ユニットストライド形式のデータアクセスを使用する、演算の精度が緩和された最適化された並列コード

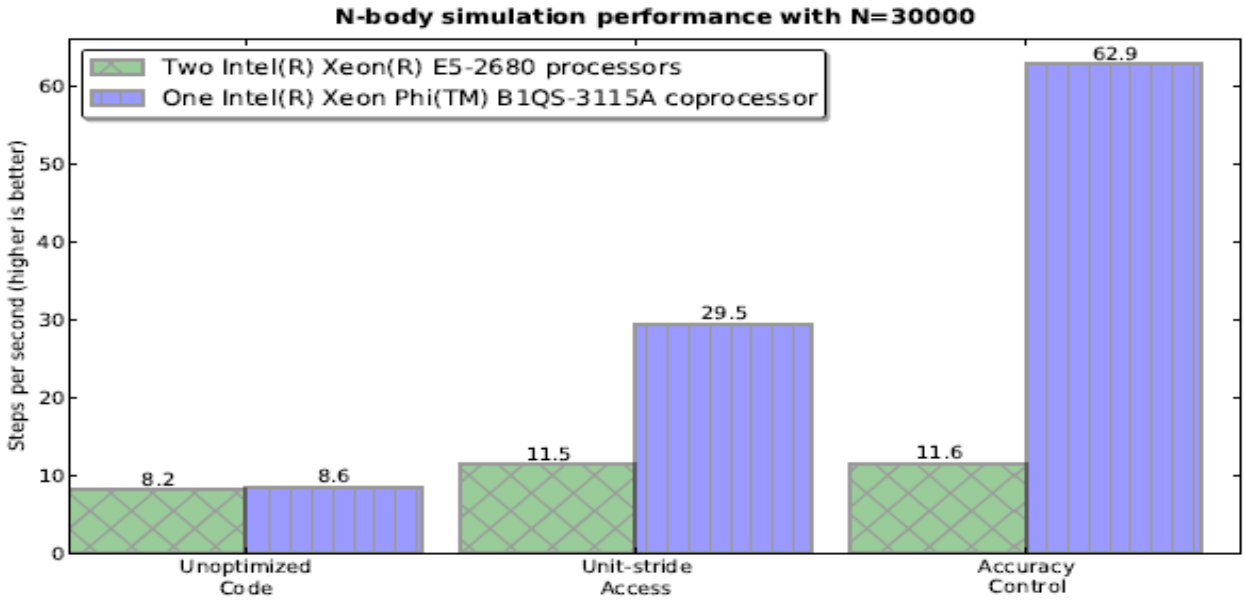


図 4: セクション 4、5、6 で説明した N 体シミュレーションのパフォーマンス結果の要約

インテル® Xeon Phi™ コプロセッサーで優れたパフォーマンスを得るには並列処理が不可欠です。タスク並列処理はベクトル並列処理と併用できます。データ構造とメモリー・アクセス・パターンが適切であれば、コンパイラーはベクトル命令を効率的に実装できます。ベクトル化されたループでは、データの局所性とユニットストライド形式のデータアクセスが利用できます。

インテル® C++ コンパイラーは、自動的にベクトル化されるコードでさまざまなデータ・アライメントに対応するため、複数の実行パスを実装することができます。ベクトル化されるループでは、リダクションや粗粒度の分岐にも対応します。ここで紹介したケースでは、自動ベクトル化を効率的に行うために追加のコードやコンパイラー・オプションは必要ありません。コンパイラーの機能を利用することで、低水準プログラミングを行うことなく、非常に効率的なアプリケーションを開発することができます。

ここで取り上げた N 体シミュレーションのネイティブ・アプリケーションは、インテル® Xeon Phi™ コプロセッサー向けのプログラミング・モデルや最適化の優れた例です。ネイティブ実行に加えて、オフロードモデルを使用することもできます。オフロードモデルでは、この計算を複数のコプロセッサーで簡単にスケーリングできます。「百聞は一見にしかず」です。Colfax の YouTube チャネルで、複数のコプロセッサーでの N 体シミュレーションのデモを一度ご覧になってください (図 1 にデモのスナップショットがあります)。

謝辞

この記事について多くの建設的なコメントを寄せてくださった、スタンフォード大学 シニア・リサーチ・サイエンティストの Troy A. Porter 氏に感謝します。

追補あり

ホワイトペーパー「基本的な N 体シミュレーションによるインテル® Xeon Phi™ コプロセッサーのテスト」の追補

Andrey Vladimirov (スタンフォード大学)
Vadim Karpusenko (Colfax International)

2013 年 5 月 8 日

概要

この追補では、オリジナルのホワイトペーパーに含まれていなかった最適化について説明します。-xAVX や -xhost コンパイラー・オプションを使用することで、ホストのシミュレーションのパフォーマンスが 1.7 倍になります。

追補

オリジナルのホワイトペーパー[1] で説明した N 体シミュレーションでは、インテル® AVX 命令セットをサポートする、インテル® Xeon® プロセッサー E5-2680 を使用しました。この命令セットは、128 ビットのレジスターの代わりに 256 ビットのベクトルレジスターを利用して、いくつかのインテル® SSE 命令のパフォーマンスを向上します。しかし、オリジナルのホワイトペーパーでは `-xhost` や `-xAVX` コンパイラー・オプションを使用しなかったため、ホストコードがインテル® AVX 命令ではなくインテル® SSE 命令でコンパイルされていました。

この追補では、`-xhost` と `-O3` コンパイラー・オプションを使用した場合の結果を説明します。`-O3` を指定してもパフォーマンスへの影響はほとんどありませんが、`-xhost` を指定するとホスト・アプリケーションの速度は 1.7 倍になり、インテル® Xeon Phi™ コプロセッサーのパフォーマンスは、2 つのインテル® Xeon® プロセッサー E5 のパフォーマンスの 3.7 倍になります。

図 1 は、ホストシステムとコプロセッサーのシミュレーションのパフォーマンス結果を示したものです。これらのベンチマークでは、オリジナルのホワイトペーパーよりも新しい SKU のコプロセッサー (B1QS-5110P) と新しいバージョンのコンパイラー (インテル® C++ Composer XE バージョン 13.1.1.163) を使用しています。右端の 2 つのバーが、`-xhost` オプションを指定した場合のパフォーマンスです。

図 2 は、`-xhost` オプションがコンパイルの結果に与える影響を示したものです。上の画面は、`-xhost` を指定しないでコンパイルしたコードをインテル® VTune™ Parallel Amplifier XE で解析した結果です。アセンブリー・リスト (図の右側に表示) によれば、インテル® SSE 命令セットの SIMD 命令 `rsqrtps` で逆数平方根が計算されています。対称的に、下の画面は `-xhost` を指定してコンパイルしたコードの解析結果です。`rsqrtps` の代わりに、インテル® AVX 命令 `vrsqrtps` が使用されています。

`-x`*code* オプションの詳細は、『インテル® C++ コンパイラー XE 13.1 ユーザー・リファレンス・ガイド』[2] および Colfax Research の記事、『Auto-Vectorization with the Intel Compilers: is Your Code Ready for Sandy Bridge and Knights Corner?』[3] を参照してください。

`-xAVX`[4] オプションが不足していることをご指摘いただいた Georg Hager 氏[5] に感謝します。

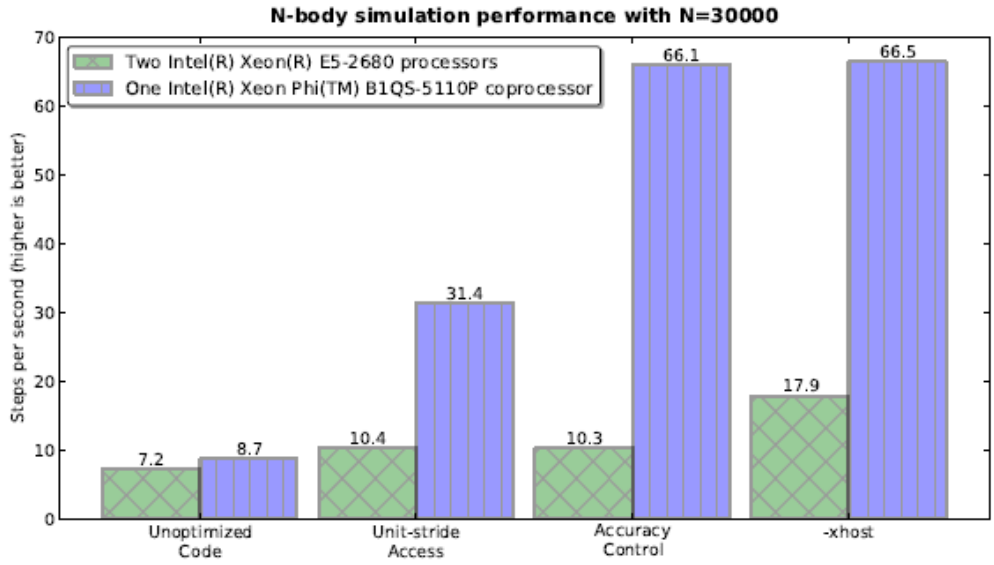
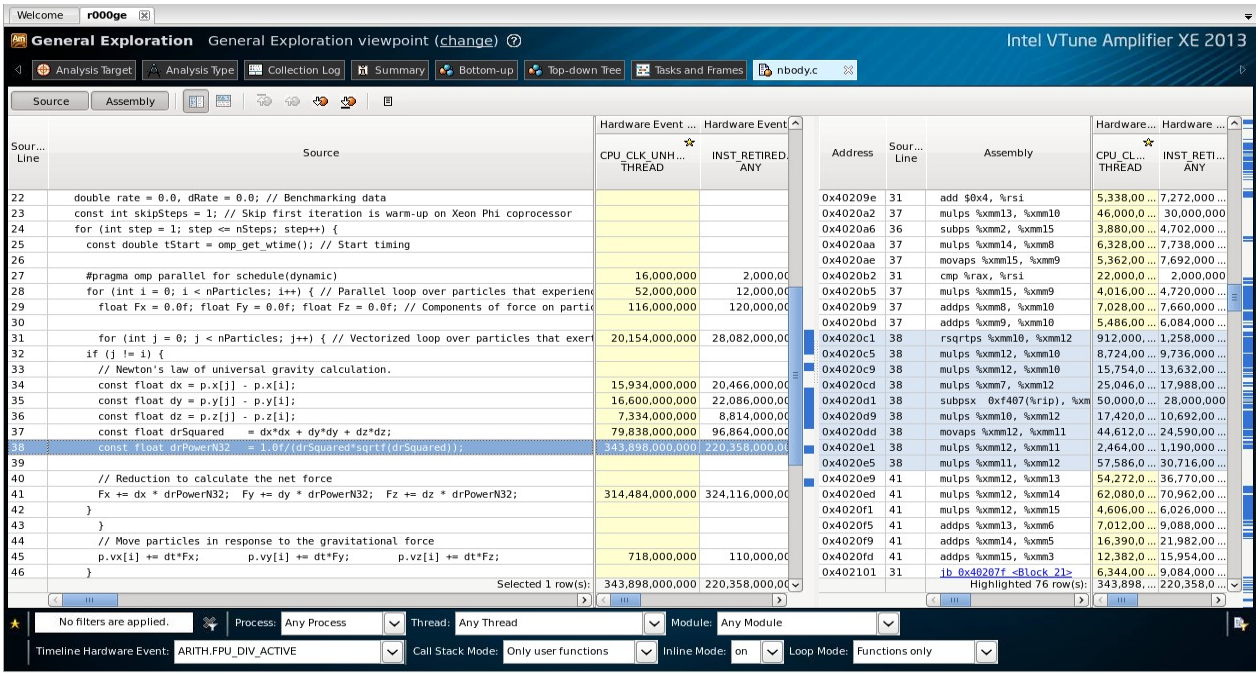


図 1: N 体シミュレーションのパフォーマンス結果の要約

[1] http://research.colfaxinternational.com/post/2013/01/07/Nbody-Xeon-Phi.aspx
[2] http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-lin/index.htm
[3] http://research.colfaxinternational.com/post/2012/03/12/AVX.aspx
[4] http://goparallel.sourceforge.net/test-driving-intel-r-xeon-phitm-coprocessors-with-a-basic-n-body-simulation/
[5] http://blogs.fau.de/hager/

-xhost を指定しない場合、インテル® SSE 命令 rsqrtps が使用されます。



-xhost を指定した場合、インテル® AVX 命令 vrsqrtps が使用されます。

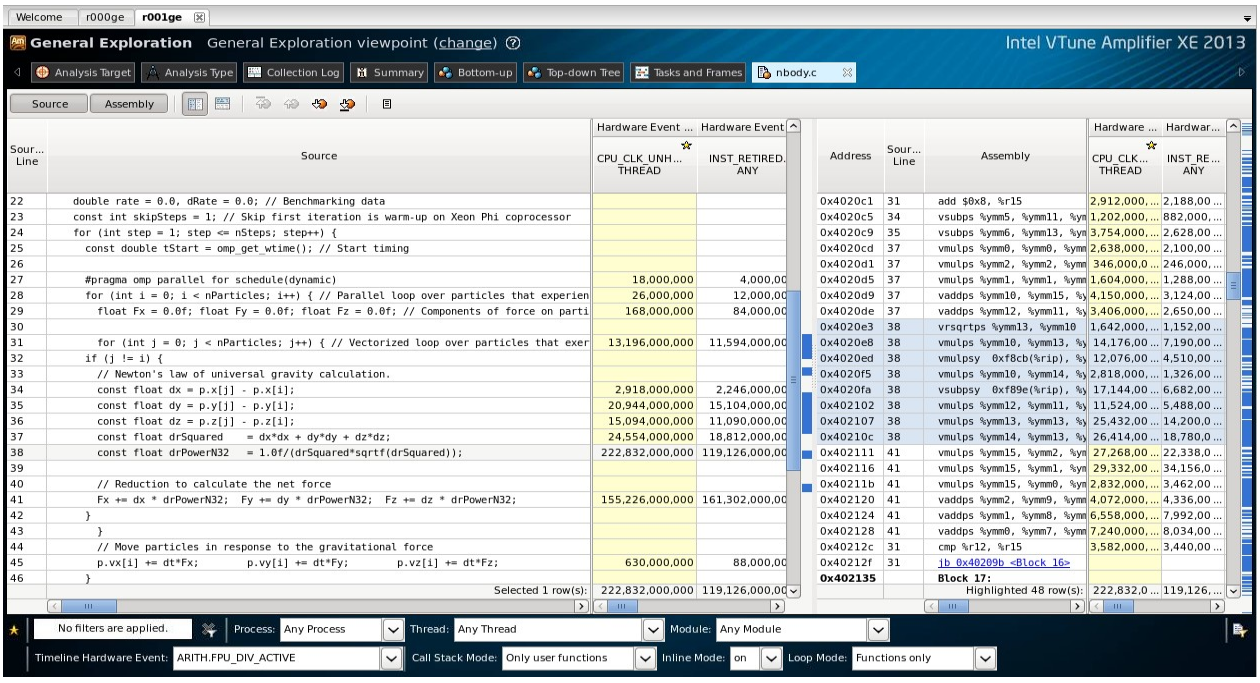


図 2: インテル® VTune™ Amplifier XE のパフォーマンス解析で、-xhost オプションを指定しない場合/指定した場合にインテル® C++ コンパイラーによって生成される命令を確認。