インテル® Xeon Phi™ コプロセッサー上での等方性倍精度 3 次元有限差分ステンシル・アルゴリズムの実行パフォーマンスの最適化

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「Optimizing Execution Performance for an Isotropic Double Precision 3DE Finite Difference Stencil Algorithm on the Intel® Xeon Phi™ Coprocessor」の日本語参考訳です。

概要

3 次元ステンシル・アプリケーションを再構成することで、アプリケーションがインテル® Xeon Phi™ コプロセッサー・ベースの Linux* システムで倍精度計算を高速に処理できるようにします。ループタイリング (ループ・ブロッキング)、OpenMP* プラグマ、SIMD ベクトル化プラグマ、ループ交換を利用してアプリケーションを再構成します。再構成に適した手法を特定するため、インテル® VTune™ Amplifier XE のサンプリングを使用します。

1. インテル® Xeon Phi™ コプロセッサー向け等方性倍精度 3 次元有限差分ステンシル・アルゴリズム

1.1. はじめに

この記事では、ループタイリング [4] (ループ・ブロッキング)、OpenMP* によるスレッド化、ベクトル化、ループ交換、およびインテル® VTune™ Amplifier XE の解析機能を使用して、3 次元ステンシル・アプリケーションのパフォーマンスをインテル® Xeon Phi™ コプロセッサー向けにチューニングします。ここでは倍精度浮動小数点を使用します。

偏微分方程式 (PDE) ソルバーは、熱拡散、電磁気学、流体力学などのさまざまな分野の科学アプリケーションで利用されます [3]。通常、これらのアプリケーションは、反復有限差分法を利用して、空間グリッドを走査し、ステンシルと呼ばれる隣接する計算を実行します。ヤコビカーネルやガウス・ザイデル・カーネルなどのステンシルコードは、多くの科学および工学アプリケーションで利用されています [2]。以下は、7 ポイントのヤコビステンシル [2] の例です。

```
for t=1..nt
// nt 回実行
    for i3=1..n3-1
// 第 3 次元
        for i2=1..n2-1
// 第 2 次元
            for i1=1..n1-1
// 第 1 次元
                b[i1,i2,i3] += a[i1−1,i2,i3] + a[i1+1,i2,i3]
// 第 1 次元の空間
                            +a[i1,i2−1,i3] + a[i1,i2+1,i3]
// 第 2 次元の空間
                            +a[i1,i2,i3-1] + a[i1,i2,i3+1]
// 第 3 次元の空間
            done
        done
    done
    swap b ⇔ a
done
```

上記の行列代入は、ガウス･ザイデル法と異なります。ガウス･ザイデル法では、右辺のオペランド a[i1−1,i2,i3]、a[i1,i2−1,i3]、a[i1,i2,i3-1]) と左辺の a[i1,i2,i3] への代入の間にデータ依存性があります [2]。

```
for t=1..nt
// nt 回実行
    for i3=1..n3-1
// 第 3 次元
        for i2=1..n2-1
// 第 2 次元
            for i1=1..n1-1
// 第 1 次元
                a[i1,i2,i3] += a[i1−1,i2,i3] + a[i1+1,i2,i3]
// 第 1 次元の空間
                            +a[i1,i2−1,i3] + a[i1,i2+1,i3]
// 第 2 次元の空間
                            +a[i1,i2,i3-1] + a[i1,i2,i3+1]
// 第 3 次元の空間
            done
        done
    done
done
```

1.2. インテル® Xeon Phi™ コプロセッサーのアーキテクチャー

インテル® Xeon Phi™ コプロセッサーには 61 のコアがあり、各コアには 4 つのハードウェア･スレッドがあります。各ハードウェア･スレッドは、512 ビットの SIMD (Single Instruction Multiple Data) ベクトル命令を実行できます。ハードウェア･スレッドごとに 32 のベクトルレジスターがあり、各レジスターは 512 ビットで (図 1)、16 の単精度オペランドまたは 8 つの倍精度オペランドを保持できます。L1 キャッシュは 64K で、32K の L1 命令キャッシュ (I キャッシュ) と 32K の L1 データキャッシュ (D キャッシュ) から成ります。L2 キャッシュは 512K で、256K の L2 命令キャッシュと 256K の L2 データキャッシュから成ります。

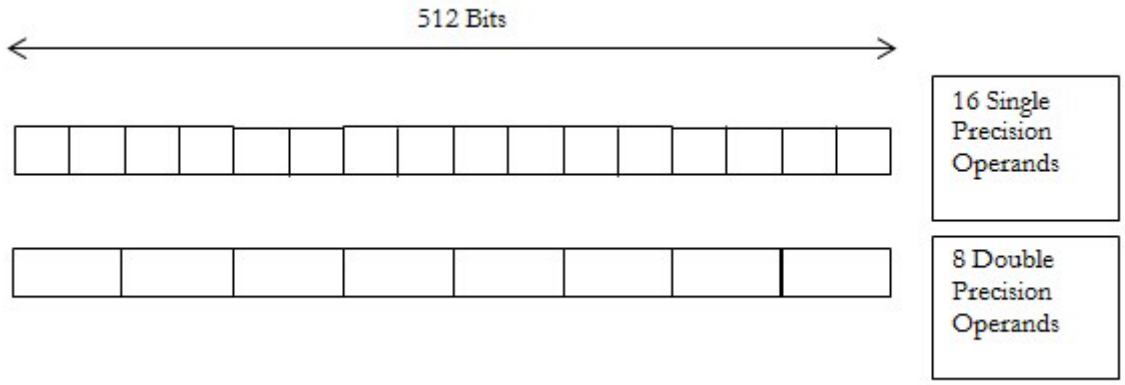

図 1 – インテル® Xeon Phi™ コプロセッサーのベクトルレジスターの構成

1.3. インテル® Xeon Phi™ コプロセッサーで利用可能な並列性レベル

インテル® Xeon Phi™ コプロセッサーは 3 つの並列性レベルをサポートします。

1. **61 コアを利用してスレッドの並列性を表現することができます。**
2. **各コアでは 4 つのハードウェア・スレッドを利用できます。**
3. **各ハードウェア・スレッドは SIMD (Single Instruction Multiple Data) ベクトル命令を実行できます。**

61 コアあり、コアごとに 4 つのハードウェア・スレッドがあるため、244 スレッドをアプリケーションに利用することができます。さらに、各スレッドで SIMD ベクトル命令を実行できます。

2. 3 次元ステンシルデータ構造に X、Y、Z 次元のタイリングを適用する

ここまでは、高いレベルからインテル® Xeon Phi™ コプロセッサーのキャッシュ・アーキテクチャーと並列性レベルについて説明しました。ここからは、アプリケーションがインテル® Xeon Phi™ コプロセッサーで倍精度計算を高速に処理できるようにする、プログラミング手法について見てみましょう。タイリング (ブロッキング) は、ストリップマイニングとループ交換を組み合わせて、ループの反復を小さなタイルに形成し、同時に実行することでデータ局所性を利用できるようにします [1]**。**

1. **この手法は、アルゴリズム中の異なる計算でデータを再利用できる場合、データキャッシュの時間と空間の局所性を高めます。**
2. **そして、各 "ベクトル" の長さだけ、または SIMD 命令で実行される操作の数だけ、ループの反復回数を減らします。インテル® Xeon Phi™ コプロセッサー上で倍精度ベクトル演算を行う場合、このベクトルまたはストリップの長さは 1/8 になります。つまり、1 つの倍精度浮動小数点 SIMD ベクトル命令ごとに 8 つの浮動小数点データが処理されます。**

Borges のオリジナルのヤコビ法 3 次元ステンシル・アルゴリズムは、次のようになります [4]。

```
for t=1..nt
// nt 回実行
    for i3=4..n3-4
// 第 3 次元
        for i2=4..n2-4
// 第 2 次元
            for i1=4..n1-4
// 第 1 次元
                div = coeff[0]*prev[i1,i2,i3]
                for r=1..4
// 次数 8 のステンシル
                    div += coeff[r]*(prev[i1+r,i2,i3] + prev[i1-r,i2,i3]
// 第 1 次元の空間
                        +prev[i1,i2+r,i3] + prev[i1,i2-r,i3]
// 第 2 次元の空間
                        +prev[i1,i2,i3+r] + prev[i1,i2,i3-r])
// 第 3 次元の空間
                done
                next[i1,i2,i3] = 2*prev[i1,i2,i3] - next[i1,i2,i3] + div*vel[i1,i2,i3]
// 時間を更新
            done
        done
    done
    swap prev ↔ next
done
```

これは、25 ポイントのヤコビ法 3 次元ステンシル・アルゴリズムです [4]。

前述のとおり、インテル® Xeon Phi™ コプロセッサーの各コアには、512KB の L2 キャッシュと 64KB の L1 キャッシュがあり、L2 キャッシュと L1 キャッシュの半分は命令に、残り半分はデータに使用されます。上記のステンシル・アルゴリズムでは、行列/格子が効率良く実行されないことがあります。これは、大きな問題サイズでは、再利用される第 3 次元のデータ要素がキャッシュに収まらないためです。

タイリングは、コンパイラーが 3 次元ステンシルの再利用を把握できるようにするプログラミング手法です。タイリングの仕組みを理解するため、簡単な格子データ構造について考えてみます。格子 "a" (次元 a[1][4][2]) の場合、インデックスの参照パターンは次のようになります。

	Subscript Indices for Lattice “a”		
	k	j	i
Subscript values	0	0	0
	0	0	1
	0	1	0
	0	1	1
	0	2	0
	0	2	1
	0	3	0
	0	3	1

図 2 – **格子** "a" (**次元** a[1][4][2]) **のインデックス**

格子 "a" (**次元** a[1][4][2]) **は、図** 3 **のように表すことができます。**

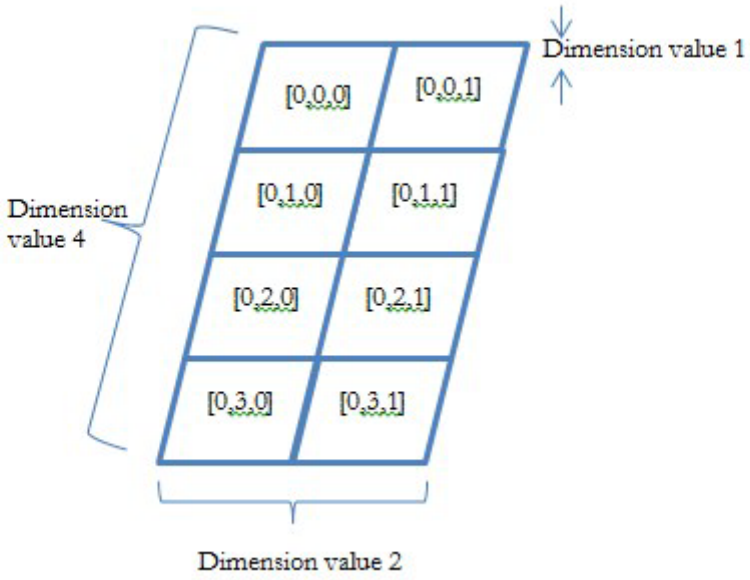

図 3 - **格子** "a" (**次元** a[1][4][2])

図 3 **の格子** "a" (**次元** a[1][4][2]) **を行優先順でコンピューター・メモリーに線形に格納した場合、図** 4 **のように表すことができます。**

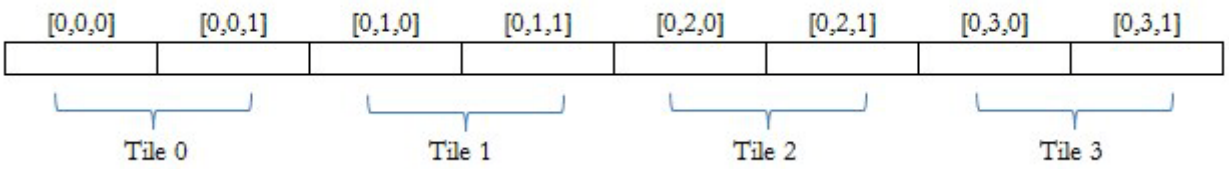

図 4 - **格子** "a" (**次元** a[1][4][2]) **を線形に格納した場合**

図 4 の単純な例では、ストレージ全体をサイズ 2 のタイルに分割することができます。そして、各タイルをインテル® Xeon Phi™ コプロセッサーの別々のコアで処理することができます。コアに割り当てられたタイルには 2 つの要素が含まれており、各要素はコア内のハードウェア・スレッドによって処理されます。

2.1. OpenMP* スレッドによる並列処理

Borges[4] の並列処理実装では、タイリング手法を使用します。サイズ n1_Tblock x n2_Tblock x n3_Tblock のスレッド化されたタイルブロックにデータを分割し、インテル® Xeon Phi™ コプロセッサーのコア、そして各コアで利用可能なハードウェア・スレッドに分配します。オリジナルの n1 × n2 × n3 範囲は、n1_Tblock × n2_Tblock × n3_Tblock ブロックを表すインデックスのリストに分解されます。このリストの長さは、num_blocks = num_n1_blocks × num_n2_blocks × num_n3_blocks = ceiling((n1-2×R)/n1_Tblock) × ceiling((n2-2×R)/n2_Tblock) × ceiling((n3-2×R)/n3_Tblock) で、各ブロックがキャッシュライン境界でアライメントされるようにパディングが追加されています。スレッド化は、OpenMP* 構文により表現されます [4]。

```
int index=0;
for (int i3b=4; i3b<n3-4; i3b+=n3_Tblock)
    for(int i2b=4; i2b<n2-4; i2b+=n2_Tblock)
        for(int i1b=4; i1b<n1-4; i1b+=n1_Tblock) {
            blocking[index].i1_idx = i1b;
            blocking[index].i2_idx = i2b;
            blocking[index].i3_idx = i3b;
            index++;
        }

#pragma omp parallel for num_threads(num_threads)
schedule(dynamic) \
    firstprivate (n1, n2, n3, num_blocks, n1_Tblock, n2_Tblock, n3_Tblock) \
    shared( coeff, prev, next, vel, blocking)
    for (int i=0; i<num_blocks; i++) {
        int i1b = blocking[i].i1_idx;
        int i2b = blocking[i].i2_idx;
        int i3b = blocking[i].i3_idx;
        apply_stencil(next, prev, vel, coeff, i1b, i2b, i3b, n1, n2, n3,
                        n1_Tblock, n2_Tblock, n3_Tblock);
    }
```

2.2. SIMD ベクトル化

ステンシル・アルゴリズムの最内ループは、次のようにベクトル化 [4] されます。

```
// 次数 8 の ISO ステンシルをブロック
// [i1b..i1b+n1_Tb]X[i2b..i2b+n2_Tb]X[i3b..i3b+n3_Tb] に適用
void apply_stencil(float *ptr_next, float *ptr_prev, float *ptr_vel, float *coeff, // 配列
const int i1b, const int i2b, const int i3b, // ブロックのインデックス
const int n1, const int n2, const int n3, // 範囲

const int n1_Tb, const int n2_Tb, const int n3_Tb) { // ブロックのサイズ
    const int n1_end = n1-4, n2_end = n2-4, n3_end = n3-4;
    const float c0 = coeff[0], c1=coeff[1], c2=coeff[2], c3=coeff[3], c4=coeff[4];
    const int n1n2 = n1*n2; const int n1_2 = 2*n1, n1n2_2 = 2*n1n2;
```

```
    const int n1_3 = 3*n1, n1n2_3 = 3*n1n2;
    const int n1_4 = 4*n1, n1n2_4 = 4*n1n2;

    const int n3b_end = MIN(i3b+n3_Tb, n3_end);
    const int n2b_end = MIN(i2b+n2_Tb, n2_end);
    const int n1b_end = MIN(i1b+n1_Tb, n1_end);

    for (int i3=i3b; i3< n3b_end; i3++) {
        double *prev = &ptr_prev[i3*n1n2+i2b*n1];
        double *next = &ptr_next[i3*n1n2+i2b*n1];
        double *vel = &ptr_vel [i3*n1n2+i2b*n1];
        for(int i2=i2b; i2< n2b_end; i2++, prev+=n1, next+=n1, vel+=n1) {

#pragma vector always
#pragma simd
            for(int i1=i1b; i1<n1b_end; i1++) {
                double tmp = c0* prev[i1]
                           + c1*( prev[i1+1] + prev[i1-1] +
                                  prev[i1+n1] + prev[i1-n1] +
                                  prev[i1+n1n2] + prev[i1-n1n2])
                           + c2*( prev[i1+2] + prev[i1-2] +
                                  prev[i1+n1_2] + prev[i1-n1_2] +
                                  prev[i1+n1n2_2] + prev[i1-n1n2_2])
                           + c3*( prev[i1+3] + prev[i1-3] +
                                  prev[i1+n1_3] + prev[i1-n1_3] +
                                  prev[i1+n1n2_3] + prev[i1-n1n2_3])
                           + c4*( prev[i1+4] + prev[i1-4] +
                                  prev[i1+n1_4] + prev[i1-n1_4] +
                                  prev[i1+n1n2_4] + prev[i1-n1n2_4]);
                next[i1] = 2.0f*prev[i1] -next[i1] +tmp*vel[i1];
            }
        }
    }
}
```

unroll プラグマでループアンロールを試みましたが、パフォーマンスは向上しませんでした。

3. インテル® Xeon Phi™ コプロセッサー上での 3 次元ステンシルの実行結果

Borges[4] のステンシルの例では、ステンシル計算の範囲を分解するために 3 次元タイルを使用しています。ここで問題となるのは、タイルのインデックス [2] i、j、k に対して 3 次元要素 s_i、s_j、s_k の値をどうすべきかです。Leopold[2] は、形状 (N-2) × s_j × (s_k × L/2) の矩形タイルと順序 N×N×N を推奨しています。ここで、L はキャッシュラインのサイズ、s_j と s_k は 3 次元タイルの j 次元と k 次元のブロック係数です。

3 次元ステンシル計算の倍精度 (DP) 浮動小数点オブジェクトを考慮しつつ、Borges[4] の式を使用してインテル® Xeon Phi™ コプロセッサー上の 512KB の L2 キャッシュに適したタイルの次元を予測します。

$$s_i \times s \times s \times ((1 \text{ キャッシュラインあたり } 8 \text{ DP オブジェクト}) / 2) \times (1 \text{ オブジェクトあたり } 8 \text{ DP バイト}) \le 512 \text{ KB (L2 キャッシュサイズ)}$$

ここで、s_i は i 次元のユニットストライドで、タイルの次元は約 1KB です。s_j と s_k は変数 s に置き換えています。

$$s^2 \leq 512KB / ((8 \times 8 / 2) \times 1KB)$$

$$s^2 \leq 16$$

$$s \leq 4$$

$s_j = s \leq 4$ になるので、次の式でタイルの次元 s_k の最小次元サイズ [2] を計算します。

$$s_k = (L/2) \times s_j$$

ここで、L はキャッシュラインのサイズ (倍精度ワード・オブジェクトの数) です。

$$s_k = (8/2) \times 4$$

$$s_k = 16$$

3.1. 最初のタイリング・アルゴリズムのパフォーマンス

前述のように、Borges[4] は、1 秒あたりの浮動小数点演算のパフォーマンスを測定する、パラメーター化したステンシル・アプリケーションを開発しました。 サンプルコード (iso-3dfd-v2.tar.gz) は、この記事の最後にあるリンクからダウンロードできます。ここで示す実行結果は、インテル® Xeon Phi™ コプロセッサー上でネイティブ実行したものです。実際の実行結果は、使用するインテル® Xeon Phi™ コプロセッサーのシステム構成やインテル® C/C++ コンパイラーのバージョンにより異なります。

ここでは、ヒューリスティックに基づいて、インテル® Xeon Phi™ コプロセッサー上でのネイティブ実行に次のタイル・パラメーターを使用します。

n1_Tblock=928 n2_Tblock=5 n3_Tblock=124

n3_Tblock の値 (タイルサイズの値 s_k と同じ) は 124 に初期化されます。61 コアで、1 コアあたり 3 つのハードウェア・スレッドを使用して上記のタイル・パラメーターを実行した場合、次のような結果になります。

時間: 4.93 秒
スループット: 1365.05 MPoints/秒
FLOPS: 45.05 GFlops

パフォーマンスをさらに向上するにはどうしたら良いでしょうか? インテル® VTune™ Amplifier XE を利用して調査します。

Cepeda[5] のインテル® Xeon Phi™ コプロセッサー向けのアルゴリズムのチューニングには、並列化、I/O のチューニング、効率良いデータ構造やライブラリー・ルーチンの選択などのコードの再構成が含まれます。一般に、アルゴリズムの最適化は、アプリケーションの hotspot やソースコードに関する知識を必要とし、アプリケーションのパフォーマンスを向上することを目的としています。

インテル® VTune™ Amplifier XE によるパフォーマンス解析は、一般に以下の手順で行います [5]。

1. hotspot (アプリケーションの合計 CPU サイクルの大半を占める関数) を特定します。

2. 次の記事の「4 効率性の評価基準」に従って、hotspot の効率を評価します: http://www.isus.jp/article/mic-article/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-2-understanding。
3. 効率が悪い場合は、同じ記事の「5 潜在的なパフォーマンス問題」にある該当する項目を確認します。評価基準の値が推奨されるしきい値に満たない場合、またはその他の理由により許容できない場合は、同じ記事の追加情報を参考に問題を特定し、修正します。
4. 上位の hotspot をすべて評価するまで上記のステップを繰り返します。

例えば、インテル® VTune™ Amplifier XE は、VPU_ELEMENTS_ACTIVE や DATA_READ_OR_WRITE などのハードウェア・イベントを収集できるため、データアクセスに対する L1 計算の比率 (VPU_ELEMENTS_ACTIVE / DATA_READ_OR_WRITE) などのパフォーマンス評価基準を計算することが可能です。

"–g" オプションを指定してコンパイルしていない場合は、このオプションを指定してアプリケーションを再コンパイルします。"-g" オプションを指定すると、インテル® VTune™ Amplifier XE はサンプリングの結果をソースにマップします。最適化オプションと "-g" オプションを同時に指定しても、通常は実行パフォーマンスに影響しません。ここでは、Linux* 上で次のインテル® VTune™ Amplifier XE の収集オプションを使用して、17 のハードウェア・カウンター (図 5) でイベントデータを収集します。

```
-collect-with runsa-knc -allow-multiple-runs -knob event-config=${vtevents}
```

シェル変数 "vtevents" には、次のハードウェア・イベントを割り当てます。

```
vtevents=CPU_CLK_UNHALTED,INSTRUCTIONS_EXECUTED,VPU_ELEMENTS_ACTIVE,DATA_READ_MISS_OR_WRITE_MISS,L2_DATA_READ_MISS_MEM_FILL,L2_DATA_WRITE_MISS_MEM_FILL,L1_DATA_HIT_INFLIGHT_PF1,DATA_READ_OR_WRITE,EXEC_STAGE_CYCLES,L2_DATA_READ_MISS_CACHE_FILL,L2_DATA_WRITE_MISS_CACHE_FILL,DATA_PAGE_WALK,LONG_DATA_PAGE_WALK,VPU_INSTRUCTIONS_EXECUTED,L2_VICTIM_REQ_WITH_DATA,HWP_L2MISS,SNP_HITM_L2
```

Linux* 上でインテル® VTune™ Amplifier XE GUI の収集画面を使用して、上記のコマンドライン情報をエクスポートすることもできます。

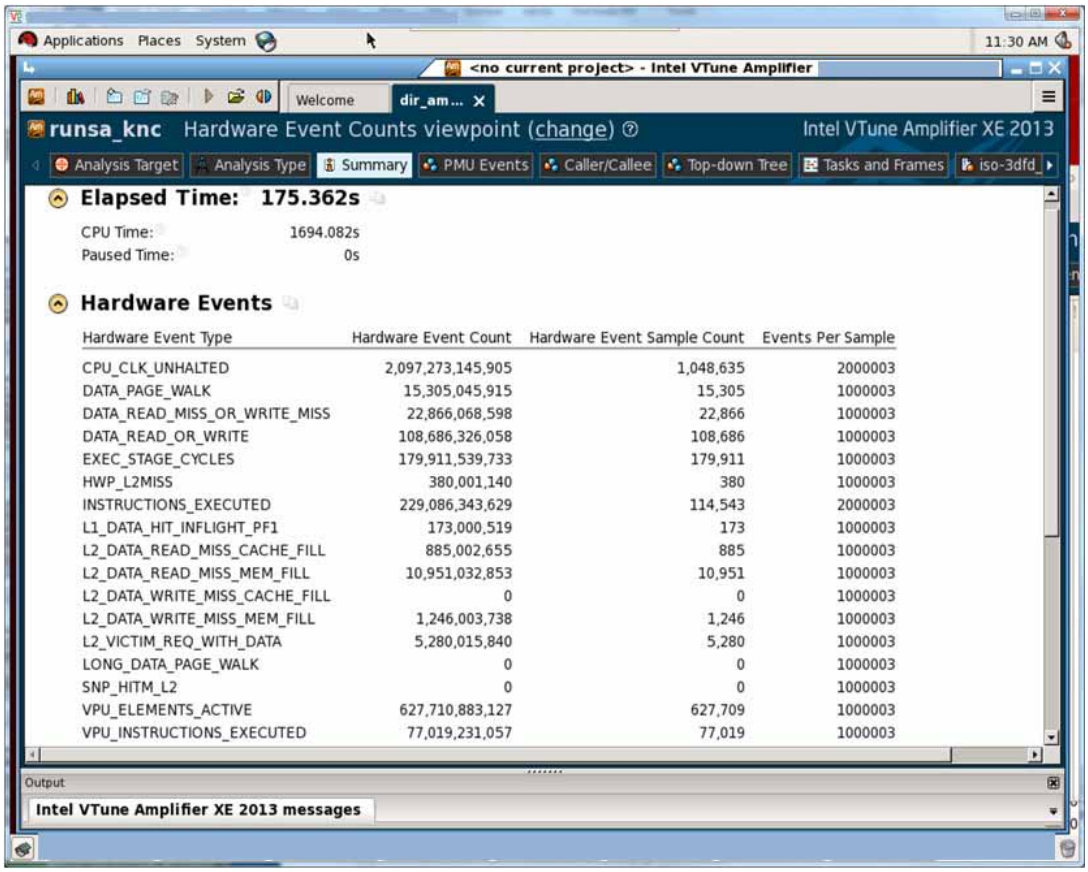

図 5 – **インテル**® VTune™ Amplifier XE **の** [Summary] **パネル**: **タイル・パラメーター** n1_Tblock=928 n2_Tblock=5 n3_Tblock=124 **を使用した場合の** 17 **のハードウェア・イベントの収集結果**

[Top-Down Tree] **パネル上で項目をクリックすると、インテル**® VTune™ Amplifier XE **の** [Source] **パネルが開き、ハードウェア・イベントをソースに関連付けることができます** (**図** 6)。

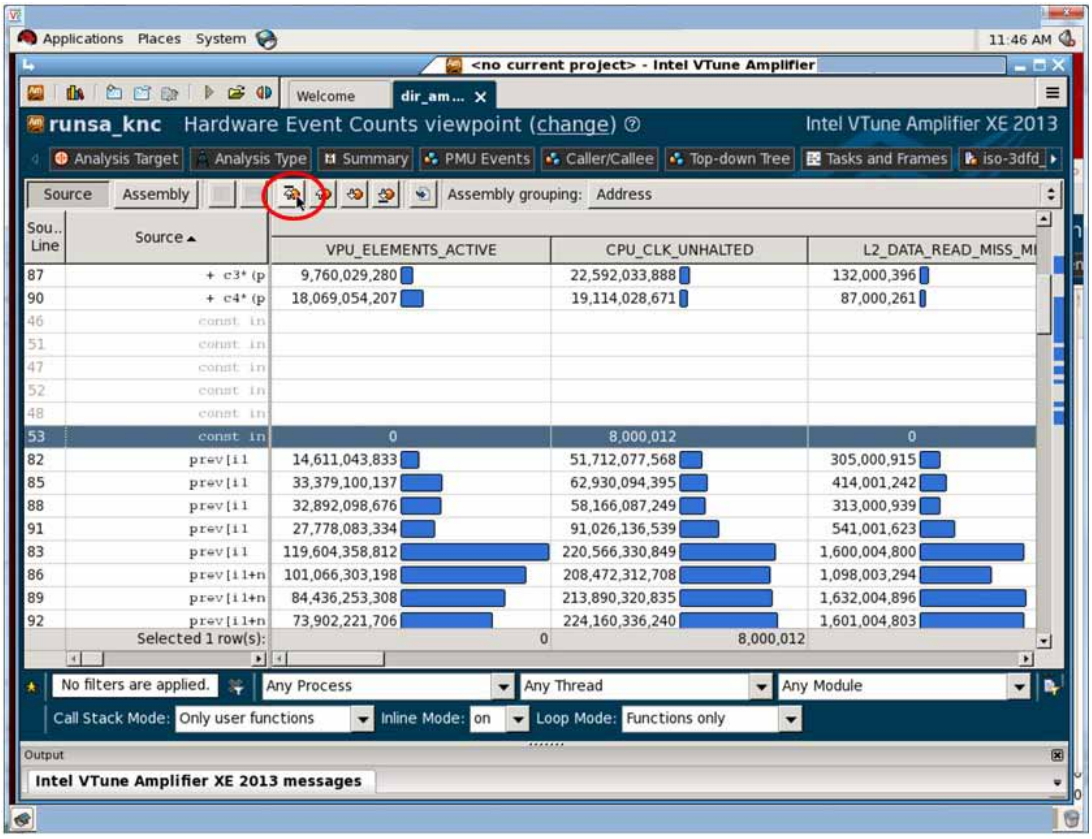

図 6 – [Source] **パネル**: [Go to Biggest Function Hotspot] **ボタンにマウスのカーソル (赤色の丸で囲んだ黒色の矢印) を合わせた状態**

[Go to Biggest Function Hotspot] **ボタンは、アプリケーションの最も重要なボトルネックに対応するソースコードを特定し、タイルの次元パラメーターを使用して** 3 **次元ステンシル・アルゴリズムの実行パフォーマンスを向上するのに役立ちます。**

このアプリケーションでは、図 5 **の次のハードウェア・イベントに注目します。**

イベント	**意味**
DATA_READ_OR_WRITE	**スレッドの** L1 **データキャッシュへのロードとストアの数**
DATA_READ_MISS_OR_WRITE_MISS	**スレッドの** L1 **キャッシュをミスした要求されたロードまたはストアの数**
VPU_INSTRUCTIONS_EXECUTED	**スレッドで実行された** VPU **命令数**
VPU_ELEMENTS_ACTIVE	VPU **命令でアクティブなベクトル要素の数、またはベクトル演算の数 (各命令は複数のベクトル演算を実行するため)。**

インテル® VTune™ Amplifier XE **で収集したイベントカウンターの値から、**Cepeda[5] **の評価基準を計算します。この評価基準は、タイリングによりアプリケーションのベクトル演算のパフォーマンスを向上するためのガイダンスを提供します。**

評価基準	式	以下の場合は調査:	タイル値 n1_Tblock=928 n2_Tblock=5 n3_Tblock=124 の倍精度評価基準
データアクセスに対する L1 計算の比率	VPU_ELEMENTS_ACTIVE / DATA_READ_OR_WRITE	< ベクトル強度	6
データアクセスに対する L2 計算の比率	VPU_ELEMENTS_ACTIVE / DATA_READ_MISS_OR_WRITE_MISS	< 100 × データアクセスに対する L1 計算の比率	27
ベクトル強度	VPU_ELEMENTS_ACTIVE / VPU_INSTRUCTIONS_EXECUTED	<8 (倍精度)、<16 (単精度)	8

上記の表で、"SP" は単精度、"DP" は倍精度を表します。

「データアクセスに対する L2 計算の比率」は、VPU_ELEMENTS_ACTIVE を DATA_READ_MISS_OR_WRITE_MISS で割った結果です [5]。同様に、「ベクトル強度」は、VPU_ELEMENTS_ACTIVE を VPU_INSTRUCTIONS_EXECUTED で割った商です。表の「以下の場合は調査」列は、良好な実行パフォーマンスを達成するために必要なしきい値を示します。しきい値に満たない場合、ボトルネックの原因を調査したほうが良いでしょう。

「L2 のデータアクセスに対する計算の比率」は、各 L2 アクセスで発生するベクトル演算の平均数を示します [5]。一般に、L1 キャッシュにデータをブロック化できるアプリケーション、またはデータアクセスを減らせるアプリケーションは、この比率が高くなります。ベースラインとして使われている 100 x L1 比率のしきい値は、100 回の L1 データアクセスごとにほぼ 1 回の L2 データアクセスが行われることを意味します。L1 評価基準と同様に、(データの移動を含む) ベクトル演算はすべて分子に含まれます。

3.2. タイルサイズ調整後のパフォーマンス

インテル® Xeon Phi™ コプロセッサーでネイティブ実行する場合は、タイル・パラメーターを次のような値に調整します。

n1_Tblock=928 n2_Tblock=5 n3_Tblock=16

タイルの次元の値 n3_Tblock=16 は、前述の倍精度 3 次元ステンシル計算の理論値と一致します。

インテル® VTune™ Ampilifier XE で実行すると、次のような結果が得られます。

メトリック	式	以下の場合は調査:	タイル値 n1_Tblock=928 n2_Tblock=5 n3_Tblock=16 の倍精度評価基準
データアクセスに対する L1 計算の比率	VPU_ELEMENTS_ACTIVE / DATA_READ_OR_WRITE	< ベクトル強度	7
データアクセスに対する L2 計算の比率	VPU_ELEMENTS_ACTIVE / DATA_READ_MISS_OR_WRITE_MISS	< 100 × データアクセスに対する L1 計算の比率	27
ベクトル強度	VPU_ELEMENTS_ACTIVE / VPU_INSTRUCTIONS_EXECUTED	<8 (倍精度)、<16 (単精度)	8

61 コアで、1 コアあたり 3 つのハードウェア・スレッドを使用して実行した場合、次のような結果になります。

時間: 4.49 秒
スループット: 1501.28 MPoints/秒
FLOPS: 49.54 GFlops

「データアクセスに対する L1 計算の比率」は 6 から 7 になり、オリジナルのベクトル強度よりも向上しています。1 秒あたりの GFlops も向上していますが、「データアクセスに対する L2 計算の比率」は改善されていません。

次のようにタイル・パラメーターを調整してみましょう。

n1_Tblock=928 n2_Tblock=5 n3_Tblock=5

インテル® VTune™ Ampilifier XE で実行すると、次のような結果が得られます。「データアクセスに対する L2 計算の比率」が向上しているのが分かります。

評価基準	式	以下の場合は調査:	タイル値 n1_Tblock=928 n2_Tblock=5 n3_Tblock=5 の倍精度評価基準
データアクセスに対する L1 計算の比率	VPU_ELEMENTS_ACTIVE / DATA_READ_OR_WRITE	< ベクトル強度	7
データアクセスに対する L2 計算の比率	VPU_ELEMENTS_ACTIVE / DATA_READ_MISS_OR_WRITE_MISS	< 100 × データアクセスに対する L1 計算の比率	28
ベクトル強度	VPU_ELEMENTS_ACTIVE / VPU_INSTRUCTIONS_EXECUTED	<8 (倍精度)、<16 (単精度)	8

61 コアで、1 コアあたり 3 つのハードウェア・スレッドを使用して実行した場合、次のような結果になります。

時間: 3.83 秒
スループット: 1758.12 MPoints/秒
FLOPS: 58.02 GFlops

上記の実行結果から、3 次元ステンシル・アルゴリズムの 1 秒あたりの GFlops が向上したことが分かります。

3.3. ループ交換とタイルサイズ調整後のパフォーマンス

Borges[4] のオリジナルのヤコビ法 3 次元ステンシル・アルゴリズムの OpenMP* ループ構造を再度見てみましょう。

```
for(int it=1; it<nreps; it++) { // タイミングループの開始

…

#pragma omp parallel for num_threads(num_threads) schedule(dynamic) \
    firstprivate (n1, n2, n3, num_blocks, n1_Tblock, n2_Tblock, n3_Tblock) \
    shared( coeff, prev, next, vel, blocking)
    for (int i=0; i<num_blocks; i++) {
        int i1b = blocking[i].i1_idx;
        int i2b = blocking[i].i2_idx;
        int i3b = blocking[i].i3_idx;
        apply_stencil(next, prev, vel, coeff, i1b, i2b, i3b, n1, n2, n3,
                      n1_Tblock, n2_Tblock, n3_Tblock);
    }

…

} // タイミングループの終了
```

タイミングループに OpenMP* 並列領域があるのが分かります。ほとんどの並列領域では [6]、フォーク・ジョインの数をできるだけ小さくすると (つまり、OpenMP* 並列領域を最外入れ子レベルに配置すると)、最良の並列パフォーマンスが得られます。上記のコードの場合、ループ交換 [6] がループの最適化プロセスの一部であり、"it" ループが各フォーク操作に含まれると仮定すると、タイミングループを "apply_stencil" 関数内に配置できます。理論的には、タイミングループが時間の次元でデータの再利用を向上する*可能性がある*計算の一部になるように、ループの入れ子を再構成します [7]。オリジナルの 3 次元ステンシル・アルゴリズムでそのような再構成を行い、次のタイル値を使用して 3 次元ステンシル計算を実行します。

n1_Tblock=928 n2_Tblock=7 n3_Tblock=1

すると、次のような実行結果になります。

時間: 3.60 秒
スループット: 1871.31 MPoints/秒
FLOPS: 61.75 GFlops

これは、61 コアで、1 コアあたり 3 つのハードウェア・スレッドを使用して実行した場合の結果です。ステンシルの格子データ構造は、時間要素に対応するため 1 次元を追加した配列です [8]。次元の範囲は 2 です。

ループ交換適用後のステンシル・アプリケーションをインテル® VTune™ Amplifier XE で実行し、イベントを収集してセクション 3.1 と 3.2 で述べた評価基準を確認してみてください。

4. サンプルのダウンロード

単精度および倍精度 3 次元有限差分ステンシル・アルゴリズムのソースコード (iso-3dfd-v2.tar.gz) は、この記事の最後にあるリンクからダウンロードできます。

ダウンロードしたファイルを展開すると、"iso-3dfd_V2" ディレクトリーに、オリジナルの 3 次元ステンシル・アルゴリズム実装のソースとループ交換バージョンの関連ソースが含まれています。ソースファイルのビルドには、-mmic コマンドライン・オプションをサポートするインテル® C++ コンパイラーが必要です。

4.1. オリジナルの 3 次元有限差分ステンシルのサンプルコード

オリジナルの 3 次元ステンシル・アプリケーションのビルドには "Makefile_orig" という名前の makefile を使用します。表 1 は、単精度および倍精度 3 次元ステンシルのビルドに関する情報です。スクリプト ./float_orig_mic_native.sh と ./double_orig_mic_native.sh は、KMP_AFFINITY="granularity=thread,balanced" を指定して、インテル® Xeon Phi™ コプロセッサー上の 61 コアで、1 コアあたり 4 スレッドを使用して、単精度モードおよび倍精度モードでサンプルコードをネイティブ実行する方法を示します。シェルスクリプト ./float_orig_offload.sh と ./double_orig_offload.sh は、インテル® Xeon Phi™ コプロセッサー上で、単精度モードおよび倍精度モードでサンプルコードをオフロード実行する方法を示します。

モード	ディレクトリーのクリーン	実行ファイルの生成	実行ファイルのシェルスクリプト・コマンド
単精度ネイティブモード	make –f Makefile_orig clean arch=mic precision=float	make –f Makefile_orig arch=mic precision=float	float_orig_mic_native.sh
単精度オフロードモード	make –f Makefile_orig clean arch=offload precision=float	make –f Makefile_orig arch=offload precision=float	float_orig_offload.sh
倍精度ネイティブモード	make –f Makefile_orig clean arch=mic precision=double	make –f Makefile_orig arch=mic precision=double	double_orig_mic_native.sh
倍精度オフロードモード	make –f Makefile_orig clean arch=offload precision=double	make –f Makefile_orig arch=offload precision=double	double_orig_offload.sh

表 1 – オリジナルの 3 次元ステンシル・アルゴリズムの makefile コマンドとシェルスクリプト

4.2. ループ交換後の 3 次元有限差分ステンシルのサンプルコード

ループ交換バージョンの 3 次元ステンシル・アプリケーションのビルドには、"Makefile_loop_interchange" という名前の makefile を使用します。表 2 は、単精度および倍精度 3 次元ステンシルのビルドに関する情報です。スクリプト ./float_loop_interchange_mic_native.sh と ./double_loop_interchange_mic_native.sh は、インテル® Xeon Phi™ コプロセッサー上で、単精度モードおよび倍精度モードでサンプルコードをネイティブ実行する方法を示します。シェルスクリプト ./float_loop_interchange_offload.sh と ./double_loop_interchange_offload.sh は、インテル® Xeon Phi™ コプロセッサー上で、単精度モードおよび倍精度モードでサンプルコードをオフロード実行する方法を示します。

モード	ディレクトリーのクリーン	実行ファイルの生成	実行ファイルのシェルスクリプト・コマンド
単精度ネイティブモード	make –f Makefile_loop_interchange clean arch=mic precision=float	make –f Makefile_loop_interchange arch=mic precision=float	float_loop_interchange_mic_native.sh
単精度オフロードモード	make –f Makefile_loop_interchange clean arch=offload precision=float	make –f Makefile_loop_interchange arch=offload precision=float	float_loop_interchange_offload.sh
倍精度ネイティブモード	make –f Makefile_loop_interchange clean arch=mic precision=double	make –f Makefile_loop_interchange arch=mic precision=double	double_loop_interchange_mic_native.sh
倍精度オフロードモード	make –f Makefile_loop_interchange clean arch=offload precision=double	make –f Makefile_loop_interchange arch=offload precision=double	double_loop_interchange_offload.sh

表 2 – ループ交換バージョンの 3 次元ステンシル・アルゴリズムの makefile コマンドとシェルスクリプト

5. 参考文献

1. G. Rivera, C. W. Tseng, “Tiling Optimizations for 3D Scientific Computations”, *Proceedings of 2000 IEEE/ACM Conference on Supercomputing*, November 2000, pp. 1-23.
2. C Leopold, “Tight Bounds on Capacity Misses for 3D Stencil Codes”, *Proceedings of the International Conference on Computational Science*, Amsterdam, April 2002, pp. 843-852.
3. K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, K. Yelick, “Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors”, *SIAM Review*, Vol. 51, No. 1, 2009, pp. 129-159.
4. http://software.intel.com/en-us/blogs/2012/10/26/experiences-in-developing-seismic-imaging-code-for-intel-xeon-phi-coprocessor
5. http://www.isus.jp/article/mic-article/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-2-understanding
6. M. Wolfe, “More Iteration Space Tiling”, *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, November 1989, pp. 655-664.
7. S. M. Faisur Rhaman, Q. Yi, A. Qasem, “Understanding Stencil Code Performance on Multicore Architectures”, *Proceedings of the 8th ACM International Conference on Computing Frontiers*, May 2011, p. 30.
8. S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, K. Yelick, “The Potential of the Cell Process for Scientific Computing”, *Proceedings of the 3rd ACM International Conference on Computing Frontiers*, May 2006, pp. 9-20.

iso-3dfd-v2.tar.gz (9.86 KB)

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