Vol.2014-HPC-143 No.31
2014/3/4

Xeon Phi（Knights Corner）の性能特性とステンシル計算の評価

松田 元彦[1,2]　丸山 直也[1,2]　滝澤 真一朗[1]

概要：高レベルなアルゴリズム記述を行う DSL のターゲット・アーキテクチャとして，メニーコアを考える必要がある．そこで，Xeon Phi（Knights Corner）上でステンシル計算のベンチマーク評価行った．タイリングと組込み関数による記述で 186 GF/s（単精度）の性能が得られた．しかし，この値はピーク性能の約 2 TF/s からはほど遠い．メモリバウンドの緩和を狙ってテンポラル・ブロッキングを行う最適化を行ったが，性能は逆に 127 GF/s に低下した．オーバーヘッドが大きい最適化であるので，比較的能力の低い整数演算部への負荷の増加が原因と考えられる．ベンチマークでは，コードを最適化すると SMT スレッド数により性能が低下する現象が現れた．確認のため単純な L2 キャッシュ・ベンチマークを行い，同様に SMT スレッド数により性能が低下することを確認した．

1. はじめに

Physis [1] などの高レベルなアルゴリズム記述を行う DSL のターゲット・アーキテクチャとして，メニーコアを考える必要がある．Physis の適用ドメインはステンシル計算カーネルであり，CPU や GPU 等で高効率なコードを生成することができる．メニーコア上でも高効率なコードを生成する必要がある．そこで，Xeon Phi (Knights Corner)（以下 KNC）でステンシル計算の最適化を試みる [2], [3], [4], [8]．KNC は比較的能力の低い整数演算部と，強力な SIMD ベクタ演算器を持つ．この構成は，整数部 : ベクタ部の性能比率は変っても，HPC ワークロードをターゲットにするメニーコアでは普遍的だと考えられる．

ステンシル計算は演算当りのメモリ参照が多いので，キャッシュを活用するため各種のタイリング（ブロッキング）による最適化が行われる．ステンシル計算のタイリング手法は複雑化しており [9]，整数演算部への負担が大きくなっていることが考えられる．

KNC のメモリコントローラは特徴的なリング・バスによる構成で，詳細な性能調査研究がある [5], [6], [7]．一方，コアの性能については詳細な性能調査はあまりない．そこで，メモリ参照の多いプログラムでのコアの挙動調査を含めたベンチマークを行うことにした．以下では，KNC 上でのステンシル計算ベンチマーク評価，および，そこで現れたスレッド数による性能低下の現象について報告する．

[1] 理研 計算科学研究機構
RIKEN AICS
[2] JST CREST

2. KNC の構成

2.1 KNC の仕様

使用した環境の主要要素を以下に示す．ソフトウェアは 2014 年 1 月時点で最新のものを使用している．ハードウェアは 60 コア，クロック 1.053 GHz の KNC で，一世代古い B1-stepping のチップである [25].

KNC ボード	P5110P (冷却ファンなし)
KNC チップ	B1-stepping
KNC OS 環境	MPSS 3.1.2
コンパイラ	ICC 14.0.1
プロファイラ	VTune Amplifier XE 2013 update15

KNC の OS の最新版である MPSS 3.1.2 では，ページサイズはデフォールトで 2 MB になっている．MPSS の導入時点で Linux カーネルのパラメータに `transparent_hugepage always` が設定されている．これは `/sys/kernel/mm/transparent_hugepage/enabled` の値で確認できる．実験ではパワーマネージメントを切って行った．そのため，MPSS の設定を変えている．

```
PowerManagement
  "cpufreq_off;corec6_off;pc3_off;pc6_off;"
```

ベンチマークでは，性能を出すためメモリの ECC を無効にすることもあるが，今回の実験ではメモリの ECC は有効なままである．

KNC の主な仕様を以下に挙げる．KNC は Intel Pentium コアの改良品に SIMD ベクタ演算器が付加された構成をし

ている．主な仕様は文献 [18] に記載がある．

整数演算器	
命令発行	同時 2 命令
SMT	4 スレッド
ベクタ演算器	
SIMD ベクタ長	倍精度 8，単精度 16
スループット	1 サイクル
レイテンシ	4 サイクル

次にキャッシュ構成を挙げる．キャッシュについても主な数値は文献 [18] に記載がある．

キャッシュ	
L1（データ）サイズ	32 KB (64 B line，8-way)
L2（unified） サイズ	512 KB (64 B line，8-way)
L1 レイテンシ	1 サイクル (文献 [22])
L2 (raw) レイテンシ	11 サイクル (文献 [18])
L2 (demand) レイテンシ	21 サイクル (文献 [18])

L1，L2 キャッシュのポート数（同時読み書き数），スループット（サイクル数），バンク数等については，文献 [18], [19], [21], [22] の間で表や記述に互いに食い違いがある．これらの値はチューニングの助けになるが，信頼できる値がない．

L2 (demand) レイテンシの値は文献 [21] には B-stepping チップについての値として記載されている．他に 30+クロックとの報告もあるが，これについては実測で確認した．実測で 21 前後の値が観測され，文献 [18] の値と一致した．実測に使ったコードから生成されたアセンブリコードを以下に示す．このコードでメモリ参照命令 `movq` のありなしの時間差を計測した．

```
rdtsc
movl %edx, %ecx; movl %eax, %r8d
movq array(%rbx), %rbx
rdtsc
```

`rdtsc` 命令はクロックカウンタ読み出しである．`rdtsc` 直後の `movl` 命令は読み出した値を他のレジスタに待避する．L2 レイテンシの計測なので，キャッシュ・エントリを L1 から排除する `clevict0` ヒント命令を使用した．

2.2 KNC でのチューニング方針

ステンシル計算のチューニングでは L2 キャッシュでのタイリングを行う．ドキュメント類に記載のある KNC の特徴のうち関係しそうな点を挙げる．

- 命令発行が SMT スレッド毎に最低 2 クロック毎
- ハードウェア・プリフェッチは L2 が対象
- L1 キャッシュ・ミスはパイプラインの命令フラッシュを伴う

L1 ミスはレイテンシのほか，付随するペナルティに注意する必要がある．L1 ミスした命令はパイプラインの実行（E）ステージでフラッシュされ，L1 にデータがロードされた時点でフェッチから再スタートする．（そのとき，同じ SMT からの以降の命令もフラッシュされる）．無駄になる命令がパイプラインを流れることになるので，演算器利用率の高いコードではこのペナルティは避ける必要がある．

```
#define REAL float

void
diffusion(REAL *f0, REAL *f1,
          int NX, int NY, int NZ,
          REAL ce, REAL cw,
          REAL cn, REAL cs,
          REAL ct, REAL cb,
          REAL cc, int count)
{
   for (int i = 0; i < count; i++) {
#pragma omp parallel for collapse(2)
      for (int z = 0; z < NZ; z++) {
         for (int y = 0; y < NY; y++) {
#pragma simd
            for (int x = 0; x < NX; x++) {
               int c, w, e, n, s, b, t;
               c = x + y * NX + z * NX * NY;
               w = (x == 0) ? c : c - 1;
               e = (x == NX - 1) ? c : c + 1;
               n = (y == 0) ? c : c - NX;
               s = (y == NY - 1) ? c : c + NX;
               b = (z == 0) ? c : c - NX * NY;
               t = (z == NZ - 1) ? c : c + NX * NY;
               f1[c] = (cc * f0[c]
                        + cw * f0[w] + ce * f0[e]
                        + cn * f0[n] + cs * f0[s]
                        + cb * f0[b] + ct * f0[t]);
            }
         }
      }
      REAL *t = f0;
      f0 = f1;
      f1 = t;
   }
}
```

図 1　ステンシル計算コード (文献 [17])

3. ステンシル計算

3.1 ステンシル計算コード

ステンシル計算は科学技術計算に現れる典型パターンのひとつで，格子点の近傍に対する一定の計算を全格子点に行うものである．古くからコンパイラ最適化の対象となっており [14]，2014 年 1 月には第一回ワークショップも行われた [9]．

本稿で用いるステンシル計算のカーネルは，青木ら [10] によるベンチマーク・コードを使用する．Xeon Phi の解説書 [17] でコード変換の解説に使用されているものと同一である．また，同一のコードに対する GPU 上の評価もある [11]．**図 1** にステンシル計算のコードを示す．これは 7 点ステンシルで，各点で 3 次元各軸の近傍 ±1 点の 7 点について計算する．各点での演算は 7 近傍に対する積和で，演算数は 13（7 FMA，Fused Multipy-Add），メモリ参照数は読み出し 7，書き込み 1 である．

問題サイズ等も解説書に従い，単精度データ，格子点サイズは $256 \times 256 \times 256$，最外ループ回数 (count) は 6553 で計測を行う．

文献 [17] のコードをベースにベンチマークを行う．ただし，文献ではコードを一つのファイルに記述しているが，カーネル部分はファイルを分けてコンパイルした．意図か

ら外れた最適化を抑制するためである．文献のままでは，呼出し側で計算している係数（cw，ce，cs，cn，cb，ct）が同じ値であるのでコンパイラが最適化してしまう．

3.2 ステンシル計算のループ変換

解説書では SIMD ベクトル化後，基本的なピーリング，タイリングのループ変換を行っている．

ピーリングでは，X 軸の 0 および $NX-1$ 時の境界条件処理を最内ループから外に移動する．

タイリングは，1 次元のタイリングで Y 軸をベクタ長と同じ 16 でタイリングしている．ベンチマークでは 2 次元のタイリングも行うので，区別するため，こちらはタイリング（1 軸）と書くことにする．タイリング（1 軸）のコード断片を以下に示す．

```
#pragma omp parallel for collapse(2)
for (int yy = 0; yy < NY; yy += 16) {
   for (int z = 0; z < NZ; z++) {
      int yub = MIN(NY, yy + 16);
      for (int y = yy; y < yub; y++) {
         ......
```

3.3 組込み関数による最内ループの記述

次に，最内ループをコンパイラ組込み関数（intrinsics）を使って書き直した．**図 2** にコードを示す．このチューニングはマイクロコード記述に近い [14]．X 軸上でインデックスが ±1 ずれた格子データ要素は，ベクタレジスタ上のシフトで生成できる．また，データ参照パターンが単純なので手作業でプリフェッチ命令を挿入する．この時，コンパイラによるプリフェッチ命令の挿入は`#pragma noprefetch` によって抑制している．

Intel のコンパイラは組込み関数の記述に対してもレジスタ割当てと命令スケジューリングを行う．面倒な命令同時発行のペアルールや 4 サイクルあるベクタ命令のレイテンシはコンパイラが対処してくれる．このコード列では，ベクタ演算の命令列に対してその他のプリフェッチやオフセット計算がそれぞれペアになり，ほぼ無駄のない命令列が生成される．

`_MM_HINT_T0` の付いた`_mm_prefetch` 命令は，L2 から L1 へのプリフェッチである．先に述べたように，KNC では L1 ミスを回避する必要があるのでこのプリフェッチが必要である．

使用している命令のいくつかを補足説明する．`__m512` はベクタ（単精度）データタイプである．`_mm512_extload_ps` は，定数データをベクタレジスタにロードする．末尾の `_ps` は（パックされた）単精度データであることを表す．`_mm512_alignr_epi32` はデータのアドレス・アラインメントを合わせる命令である．ベクタデータに対して，データ単位でのシフト，マスク，マージを行う．この `alignr` 命令を使って X 軸上でインデックスがずれた格子データをベクタレジスタ上に生成する．`_mm512_mask_alignr_epi32`

```
#define uc _MM_UPCONV_PS_NONE
#define bc _MM_BROADCAST_1X16
#define ht _MM_HINT_NONE
__m512 vcc = _mm512_extload_ps(&cc, uc,bc,ht);
__m512 vcw = _mm512_extload_ps(&cw, uc,bc,ht);
__m512 vce = _mm512_extload_ps(&ce, uc,bc,ht);
__m512 vcn = _mm512_extload_ps(&cn, uc,bc,ht);
__m512 vcs = _mm512_extload_ps(&cs, uc,bc,ht);
__m512 vcb = _mm512_extload_ps(&cb, uc,bc,ht);
__m512 vct = _mm512_extload_ps(&ct, uc,bc,ht);
int c, n, s, b, t;
c = 0 + y * NX + z * NX * NY;
n = (y == 0) ? c : c - NX;
s = (y == NY - 1) ? c : c + NX;
b = (z == 0) ? c : c - NX * NY;
t = (z == NZ - 1) ? c : c + NX * NY;
__m512 vprv;
__m512 v0;
v0 = _mm512_load_ps(&f0[c]);
vprv = _mm512_alignr_epi32(v0, v0, 1);
#pragma noprefetch
#pragma novector
for (int x = 0; x < NX; x += 16) {
    __m512 vnxt;
    vnxt = _mm512_load_ps(&f0[c + 16]);
    int mask = (0 - ((x + 16) >> 8));
    __mmask16 kk = _mm512_int2mask(mask);
    vnxt = _mm512_mask_alignr_epi32(vnxt, kk, v0, v0, 15);

    _mm_prefetch(&f0[b + 256], _MM_HINT_T0);
    _mm_prefetch(&f0[s + 256], _MM_HINT_T0);
    _mm_prefetch(&f0[t + 256], _MM_HINT_T0);

    __m512 v1 = _mm512_alignr_epi32(v0, vprv, 15);
    __m512 v2 = _mm512_alignr_epi32(vnxt, v0, 1);

    __m512 v3 = _mm512_load_ps(&f0[n]);
    __m512 v4 = _mm512_load_ps(&f0[s]);
    __m512 v5 = _mm512_load_ps(&f0[b]);
    __m512 v6 = _mm512_load_ps(&f0[t]);

    __m512 v7, v8;
    v8 = _mm512_setzero_ps();
    v8 = _mm512_fmadd_ps(vcs, v4, v8);
    v8 = _mm512_fmadd_ps(vct, v6, v8);
    v8 = _mm512_fmadd_ps(vce, v2, v8);
    v7 = _mm512_setzero_ps();
    v7 = _mm512_fmadd_ps(vcc, v0, v7);
    v7 = _mm512_fmadd_ps(vcw, v1, v7);
    v7 = _mm512_fmadd_ps(vcn, v3, v7);
    v7 = _mm512_fmadd_ps(vcb, v5, v7);
    v7 = _mm512_add_ps(v7, v8);

    _mm_prefetch(&f0[t + 2*256], _MM_HINT_T1);
    vprv = v0;
    v0 = vnxt;
    _mm512_storenrngo_ps(&f1[c], v7);
    c += 16;
    n += 16;
    s += 16;
    b += 16;
    t += 16;
}
```

図 2 組込み関数を使った最内ループ

はプリディケート付きの `alignr` 命令である．ここでは，$x = NX-1$ 時の境界条件処理をプリディケート付き命令で行っている．

ロード命令`_mm512_load_ps` と積和命令`_mm512_fmadd_ps` に対してはメモリオペランドを持つ積和 1 命令が生成される．

`_mm512_storenrngo_ps` はストリーミング・ストア命令のひとつでストア順を保持しないものである（non-globally ordered）．ステンシル計算では，次の繰り返しまでデータを参照しないのでストリーミング・ストア命令が有効に使

える．

_mm512_mask_alignr_epi32 と _mm512_alignr_epi32 やその他の場所で C 言語の型キャストは省略した．また，このコードでは，配列領域外を 1 ベクタ長分余分にアクセスしている．

3.4 タイリング

次は，組込み関数で書いた最内ループを使ってタイリングとテンポラル・ブロッキングの変換を行う．

まず，タイリングであるが，前節のタイリング（1 軸）とは異なり，L2 キャッシュ・サイズに合わせた分割を行う．そのため，Y-Z 軸両方を分割する．

```
#pragma omp parallel for collapse(2)
for (int y0 = 0; y0 < NY; y0 += BY) {
   for (int z0 = 0; z0 < NZ; z0 += BZ) {
      int zub = MIN((z0 + BZ), NZ);
      int yub = MIN((y0 + BY), NY);
      for (int z = z0; z < zub; z++) {
         for (int y = y0; y < yub; y++) {
            ......
```

SMT スレッド数毎にブロックサイズを変えている．タイリングの場合，X-Y 面が最低 2 面キャッシュに載る必要がある．次節で述べるテンポラル・ブロッキングの場合，X-Y 面が 5 面キャッシュに載る必要がある．それ以外に特別な方針はなく，X 軸は分割しないので，Y 軸のサイズを選択した．以下に計測に使ったブロックサイズを示す．

スレッド数	$BY \times BZ$ (カッコ内はスレッド分割)
1	43×26 $(60 = 6 \times 10)$
2	22×26 $(120 = 12 \times 10)$
3	22×18 $(180 = 12 \times 15)$
4	26×11 $(240 = 10 \times 24)$

3.5 テンポラル・ブロッキング

テンポラル・ブロッキングは，ステンシル計算の最外ループ（コード上は i）をタイリングの対象にする [13]．意味的に時間発展の軸なのでテンポラル・ブロッキングという．コンパイラのループ変換としては，テンポラル・ブロッキングはスキューイングとタイリングの組合わせの適用である．古くからコンパイラ最適化の対象となっている [15]．

テンポラル・ブロッキングは最外ループの複数段をタイリングして一度に計算する．ステンシル計算を続けて複数段行うなうので，テンポラル・ブロッキングはデータ参照関係上はステンシルの点数を増やすことになる．点数が多い方が同じデータを再利用する回数が増えるので，タイリングの効果が高くなる．単純化したコスト計算を行うと，テンポラル・ブロッキングを N 段行う場合，タイリングで保持しているデータに対してステンシル計算を N 回行うことができる．よって，メモリ要求を $1/N$ にできる．ただし，実際にはループ運搬依存がありスキューイングに相当する変換が必要になるので複雑である．

ここではループ運搬依存の解決に，ループ変換ではなく halo に当る部分を余分に計算する overlapped tiling [12] を用いた．余分に行う計算の量はタイリングの分割によるが，KNC の 60–240 スレッドの場合 1 割から 2 割になる．KNC の場合スレッド間の同期が遅いことが知れているので，細粒度の同期が不要な overlapped tiling を選択した．

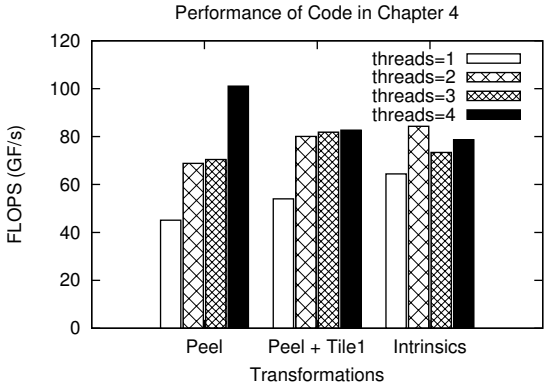

図 3 ステンシル計算 文献 [17] のループ変換

単純タイリングの場合，単精度 7 点ステンシルの FMA 演算数は 7 ops，read と write を 1 回ずつ行うのでデータ量は 4+4 byte である．演算要求の byte:ops 比は約 1:1 になる．一方，KNC の理論ピーク性能は，コア当り単精度 FMA 演算 16 G ops．理論メモリスループットはチップ全体で 320 GB/s，コア当り 5.3 GB/s．KNC の演算性能の byte:ops 比は約 1:3 である．byte:ops 比を比較すると，演算性能の ops 値が演算要求を上回るので単純タイリングはメモリバウンドである．

テンポラル・ブロッキングを一段行うと，キャッシュ上のデータにステンシル計算を 2 回行う．つまり，7 点ステンシルの FMA 演算数は 14 ops，データ量は 4+4 byte となる．演算数が 2 倍に増えるので，演算要求の byte:ops 比は KNC の性能に近づく．テンポラル・ブロッキングはコードが複雑になるので一段だけとした．

4. ステンシル計算の性能

4.1 評価環境設定

コンパイル・オプションは-restrict -opt-assume-safe-padding -O3 である．ポインタ変数の宣言には全て __restrict__ と __attribute__ ((align(64))) を付加している．配列は 2 MB 境界のアドレスに置いている．計測は 5 回行い，その最高値を使った．SMT スレッド数によるスレッドの配置（KMP_AFFINITY）は，以下の組合わせで指定した：スレッド数 1 の場合 scatter，2 の場合 balanced，3 の場合 balanced，4 の場合 compact．

4.2 基本ループ変換の性能

図 3 に解説書のコードによるステンシル計算の性能を示す．図示していないが OpenMP と SIMD ディレクティブ

によるベースラインの性能は約 60 GF/s であった．

ラベル **Peel** のグラフはピーリング変換の結果である．ベースラインの約 60 GF/s から 101 GF/s（4 スレッド時）に速くなった．

しかし，これは手動のピーリング変換による効果ではなかった．ピーリングは X 軸の境界条件処理をループ外に移動するが，それにともない，配列のインデックス変数（w，e，n，s，b，t）を個々にインクリメントするように書き直している．ピーリング変換を行わず，インデックス変数をインクリメントするように書き直しても同じ性能が得られた．コンパイラは自動でピーリング変換を行うので，インデックス変数の扱い方で最適化に違いが出たことになる．ただし，インデックス変数を実際にインクリメントしている訳ではなく，コンパイラの出力コードではメモリ参照はオフセット計算に戻っている．

ラベル **Peel + Tile1** のグラフはタイリング（1 軸）変換の結果で，性能は 82.7 GF/s（4 スレッド時）であった．タイリング（1 軸）変換はピーリング変換に劣る結果となった．

ラベル **Intrinsics** のグラフは最内ループを組込み関数で書き直した結果である．比較のためにグラフに加えた．これは元のコード（図 1）の最内ループを組込み関数で記述したもので置換えている．性能は 84.3 GF/s（2 スレッド時）であった．このコードには，L2 キャシュへのタイリングが最適な場合を想定したプリフェッチだけが挿入されている．

ピーリング後の 4 スレッド時が，際立って性能が良い．コア当り 4 スレッドの場合，全体では 240 スレッドであり，Y-Z 軸の 256×256 点を 240 スレッドで分けている．Y 軸が内ループであり 256 と 240 が近い値なので，ほぼ Z 軸をスレッドで分けていることになる．これは，同一コア上の 4 スレッドが Z 軸上 4 列で Y 軸上の近い位置をスキャンするような実行になる．コア上の 4 スレッドが Z 軸上で X-Y 軸の近い点を計算するので，L1 キャシュ上で幅 4 のタイリングと同等の効果があったと考えられる [4]．

4.3 チューニング後の性能

図 **4** にチューニング後の性能を示す．

ラベル **Intrinsics** のグラフは図 3 と同一である．

ラベル **Tile2** のグラフはタイリング結果で，性能は 186 GF/s（1 スレッド時）であった．

ラベル **Temporal Blocking** のグラフはテンポラル・ブロッキングの結果で，性能は 127 GF/s（1 スレッド時）であった．

結果として，テンポラル・ブロッキングが遅くなった．テンポラル・ブロッキングが有効である見込みは皮算用であった．そこで，単純タイリングがメモリバウンドであることを確認するため，プログラムはそのままで 1 コアだけ

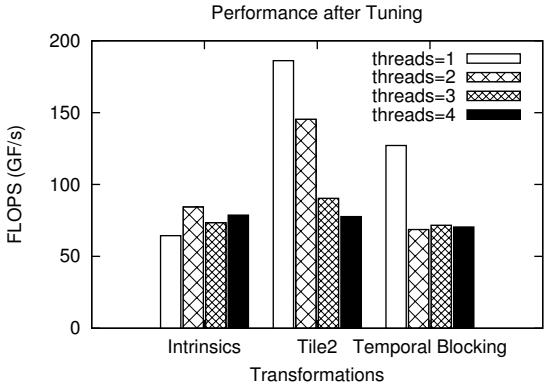

図 **4** ステンシル計算 チューニング後

で動作させてみた．1 コアならば使えるメモリバンド幅が十分高いと仮定している．1 コアで動作させた場合の実行時間は，227 GF/s に相当する値であった．一応，メモリバウンドの可能性が高いことは確認できた．これにより性能低下の原因は，テンポラル・ブロッキングによる処理オーバヘッドの増加と overlapped tiling による 1 割以上の計算の増加だと考えられる．

注意すべきは，SMT スレッド数により性能が低下する点である．Intel のドキュメントの説明に従うと，スレッドによる性能低下の原因はなさそうである．次節で，もっと単純なベンチマークを使って調べることにする．

5. KNC の性能特性

5.1 L2 キャッシュの性能測定コード

ステンシル計算のベンチマークを行うと，SMT スレッドを増やすと一貫して性能が悪くなる現象が見られた．L2 キャッシュの読み出しが，スレッドの増加により停滞している可能性が考えられる．そこで，L2 キャッシュの性能を計測するベンチマークを行った．

スレッド数を変えて，L2 キャッシュの読み出し性能を計測する．L2 キャッシュの性能を計測するには，ベンチマーク・コードを十分最適化する必要がある．L1 キャッシュミスはパイプラインをフラッシュするので，L1 のデマンドミスを起すと L2 キャッシュ性能が測定できない．一方，ハードウェア・プリフェッチはメモリ–L2 間で機能するので，連続アクセスであってもソフトウェア・プリフェッチが必要である．L2–L1 プリフェッチはコンパイラが生成してくれるが，適切でないものも生成される．そのため，コンパイラによる自動生成は抑制する．

L2 の性能測定に使ったコードの断片を以下に示す．整数配列の総和を求めるコードを使用する．C 言語で最適なコードを記述するのはトリッキーなのでここでも組込み関数を使って記述した．

```
v0 = _mm512_add_epi32(v0, _mm512_load_epi32(p0));
p0 += 16;
_mm_prefetch(q0, _MM_HINT_T0);
q0 += 16
```

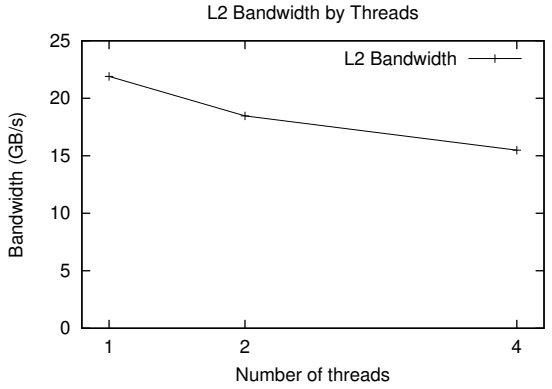

図 5 L2 キャッシュの性能

実際のコードは，ループをアンロールしてこのコード列を 16 回繰り返したものを使用する．このコード列は，プリフェッチとメモリ・オペランドを持つ加算の繰り返しである．総和は中間結果を 4 つの変数に分割して加算する（変数として v0 のほか v1–v3 を使用する）．演算レイテンシは 4 であり 2 クロック毎に命令発行するので 2 変数で十分であるが，コンパイラが最短のコード列を生成しないので 4 変数を使っている．ポインタ q0 には p0 からプリフェッチ距離だけ先行するアドレスが入っている．プリフェッチ距離は 14 キャッシュ・ラインにしている．試行錯誤で，プリフェッチ距離が 12–20 キャッシュ・ラインで性能が高かった．

このコードから以下のアセンブリが生成される．ポインタのインクリメントはコンパイラの最適化により定数オフセットになっている．

```
vpaddd n(%rdi), %zmm3, %zmm4
vprefetch0 n(%r13)
```

n は定数オフセット値であり，0，64，128，... が入る．この命令の組合わせは同時発行できる点が重要である．この組合わせは 2 クロック毎に 1 キャッシュ・ラインを読み出すので，1 スレッドで L2 の性能を引き出せるものである．ただし，実際にはキャッシュ・アクセスに 1 クロックの追加があるので約 3 クロック毎の読み出し性能になる．Intel の MIC フォーラムの情報でも，3 クロック毎に 1 キャッシュ・ラインが読み出しの限界である．

5.2 L2 キャッシュの読み出し性能

図 5 にコアにローカルな L2 アクセスの性能測定を示す．実験は，L2 サイズ 512 KB に対して十分小さい 256 KB のメモリ領域をスレッド数で分割し，そこの読み出しを繰り返した．値は，1 スレッド時 22.0 GB/s，2 スレッド時 18.5 GB/s，4 スレッド時 15.5 GB/s である．

Intel のドキュメントには，スレッド数を増やして遅くなるアーキテクチャ上の要因は書かれていない．一方，他文献のベンチマークでも，コア当りのスレッド数を 4 に増やすと性能低下する現象が良く見られる．しかし，その根拠の説明はない．一般のアプケーションではスレッド数を増やすと，問題の分割方法やキャッシュ・サイズに問題が起り得る．この L2 バンド幅測定ではそれらの問題なさそうである．

結果を説明できるような競合を調べるため，PMU (Performance Monitoring Unit) のカウンタ値を VTune を用いてプロファイルを取った．

命令実行数 `INSTRUCTIONS_EXECUTED` はスレッド数によらずほぼ同じであった．キャッシュミス率は 0.05–1.7%で 1 スレッドの場合にもっとも多く観測された．

L1 および L2 アクセスの性能低下に関係ありそうなカウンタとして，`PIPELINE_FLUSHES`，`FE_STALLED`，`EXEC_STAGE_CYCLES`，`BANK_CONFLICTS`，`VPU_STALL_REG` が挙げられる．このうち，`BANK_CONFLICTS` と `VPU_STALL_REG` ほぼ 0 であった．`PIPELINE_FLUSHES` は命令実行数の 2–6%，`FE_STALLED` は命令実行数の 10–55%と非常に多くが観測された．しかし，スレッド数による性能低下に関係しているようには見えない．`EXEC_STAGE_CYCLES` が実行時間に追従する量で増えているようであった．

`EXEC_STAGE_CYCLES` の増加はパイプラインの E ステージでの待ちである．ソフトウェアからは SMT スレッド間でのベクタ演算の依存はありえない．想像でしかないが，ハードウェア的に依存があるのかも知れない．ベクタレジスタは 128 本と多いので，あり得ないとは言えない．

6. 議論

ベクタレジスタ間での演算が主な計算では，ピーク性能が望めるかもしれない．しかし，L2 キャッシュのベンチマークから言えることは，KNC ではキャッシュにヒットしていても，メモリ参照を含む演算では 3 サイクル毎の演算が限界のようである．

命令発行が 2 サイクル毎であることや命令ペアルールの制限は，ループのオーバヘッドが大きくなる複雑な最適化を困難にしている．これは KNC の場合，アーキテクチャというより x86 命令セットに由来する制限のように思われる．HPC ワークロードでプリフェッチに効果があれば，最近でも IBM の Power 6 [16] が in-order 命令発行であったように，パイプラインは単純でキャッシュにデータがある限り演算を続けられるという構造は問題なさそうである．それを命令セットによって性能を出しにくくしているのは残念である．

KNC のチューニングで秘訣のようなものがあるとすれば，L2 キャッシュから L1 へのプリフェッチを手で挿入することである．その場合，命令発行に余裕がないので，命令数を増加させないためコンパイラによるプリフェッチ生成を抑制することも試す必要がある．

7. おわりに

KNC に対するステンシル計算の最適化を試みた．タイリングと組込み関数による記述で 186 GF/s（単精度）の性能が得られた．しかし，この値はピーク性能の約 2 TF/s からはほど遠い．さらにメモリバウンドの緩和を狙ってテンポラル・ブロッキングを行う最適化を行った．しかし，性能は逆に 127 GF/s に低下した．

ステンシル計算では，最適化したコードで SMT スレッド数により性能が低下する現象がみられた．Intel のドキュメント類には性能低下の理由が見当たらない．ただし，ステンシル計算で確認されても，ブロック分割の違い等が性能低下を引き起こしている可能性が残る．そこでブロック分割等の影響がない単純な L2 キャッシュ・ベンチマークによる測定を行った．L2 キャッシュ・ベンチマークでもスレッド数により性能が低下することを確認した．

謝辞

本研究の一部は JST CREST「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」領域，「高性能・高生産性アプリケーションフレームワークによるポストペタスケール高性能計算の実現」課題による支援を受けた．

本研究の一部は公益財団法人計算科学振興財団　研究教育拠点（COE）形成推進事業の助成を受けた．

参考文献

[1] Maruyama, N., Nomura, T., Sato, K., and Matsuoka, S.: Physis: an Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers, Proc. Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC), 2011.

[2] 高橋, 塙, 朴, 児玉, 扇谷, 佐藤: 各種アプリケーションにおける GPGPU 対 Many Core Processor の性能比較，情報処理学会研究報告, Vol.2013-HPC-139, No.21, 2013.

[3] 大島, 金子: メニーコアプロセッサ Xeon Phi の性能評価，情報処理学会研究報告, Vol.2013-HPC-139, No.20, 2013.

[4] 小林, 中田: スレッド間空間的ブロッキングを利用した Xeon Phi 上の姫野ベンチマークの最適化，情報処理学会研究報告, Vol.2013-ARC-207, No.4, 2013.

[5] Fang J., Varbanescu A. L., Sips H., Zhang L., Che Y., and Xu C.: Benchmarking Intel Xeon Phi to Guide Kernel Design, Delft Univ. of Tech., PDS, Tech. Rep. PDS-2013-005, 2013.

[6] Fang J., Varbanescu A. L., and Sips H.: Identifying the Key Features of Intel Xeon Phi: A Comparative Approach, Delft Univ. of Tech., PDS, Tech. Rep. PDS-2013-006, 2013.

[7] Fang, J., Sips, H., Zhang, L., Xu, C., and Varbanescu, A.L.: Test-Driving Intel Xeon Phi, Intl. Conf. on Performance Engineering (ICPE), 2104.

[8] Peraza, J., Tiwari, A., Laurenzano, M., Carrington, L., Ward, W. A., and Campbell, R.: Understanding the performance of stencil computations on Intel's Xeon Phi, IEEE Cluster Computing (CLUSTER), 2013.

[9] HiStencils 2014: 1st Intl. Workshop on High-Performance Stencil Computations. http://www.exastencils.org/histencils/

[10] 東京工業大学 青木 研究室 http://www.sim.gsic.titech.ac.jp/index-j.html

[11] Maruyama, N., and Aoki T.: Optimizing Stencil Computations for NVIDIA Kepler GPUs, Intl. Workshop on High-Performance Stencil Computations, 2014.

[12] Holewinski, J., Pouchet, L.-N., and Sadayappan, P.: High-performance Code Generation for Stencil Computations on GPU Architectures, Proc. of Intl. Conf. on Supercomputing (ICS), 2012.

[13] Tang, Y., and Chowdhury, R. A., Kuszmaul, B. C., Luk, C.-K., and Leiserson, C. E.: The Pochoir Stencil Compiler, Proc. Symp. on Parallelism in Algorithms and Architectures (SPAA), 2011.

[14] Bromley, M., Heller, S., McNerney, T., and Steele,Jr., G. L.: Fortran at Ten Gigaflops: The Connection Machine Convolution Compiler, Proc. of Conf. on Programming Language Design and Implementation (PLDI), 1991.

[15] Wolf, M. E. and Lam, M. S.: A Data Locality Optimizing Algorithm, Conf. on Programming Language Design and Implementation (PLDI), pp.30–44, 1991.

[16] Le, H. Q., Starke, W. J., Fields, J. S., O'Connell, F. P., Nguyen, D. Q., Ronchetti, B. J., Sauer, W. M., Schwarz, E. M., and Vaden, M. T.: IBM POWER6 microarchitecture, IBM J. Res. Dev., Nov., 2007.

[17] Jeffers, J. and Reinders, J.: *Intel Xeon Phi Coprocessor High Performance Programming*, Morgan Kaufmann, 2013.

[18] Rahman, R.: *Intel Xeon Phi Coprocessor Architecture and Tools*, Apress, 2013.

[19] Rahman R.: Intel Xeon Phi Core Micro-architecture, Apress, 2013.

[20] Rahman R.: Intel Xeon Phi Coprocessor Vector Microarchitecture, Apress, 2013.

[21] Cepeda S.: Optimization and Performance Tuning for Intel Xeon Phi Coprocessors, Part 2, http://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-2-understanding, 2012.

[22] Intel Corp.: *Intel Xeon Phi Coprocessor System Software Developers Guide*, Jun., 2013.

[23] Intel Corp.: *Intel Xeon Phi Coprocessor Instruction Set Architecture Reference Manual*, Sep. 7, 2012.

[24] Intel Corp.: *Intel Xeon Phi Coprocessor (codename: Knights Corner) Performance Monitoring Units 1.01*, Jul. 10, 2012.

[25] Intel Corp.: *Intel Xeon Phi Coprocessor x100 Product Family, Specification Update*, Dec., 2013.