

A64FX におけるテンポラルブロッキングの実装と性能評価

星野 哲也¹ 埴 敏博¹

概要：A64FX はスパコン富岳に採用されたプロセッサであり，Scalable Vector Extension (SVE) を初めて実装したプロセッサであることから，その性能特性を明らかにすることは喫緊の課題である．本稿では，流体計算等で頻出するステンシル計算の中でも最も基本的な，三次元の拡散方程式のカーネルをベースとして一連の最適化を施す．特にその性能がアーキテクチャの特性に大きく依存する最適化であるテンポラルブロッキングを実装し，A64FX にて評価を行うことで，性能特性を明らかにするための試金石とする．一連の最適化を施した三次元拡散方程式のカーネルを Intel Xeon CascadeLake, Intel Xeon Phi Knights Landing, A64FX において比較評価することで，A64FX がそれらとは異なる性能特性を持っていることが明らかとなった．特にテンポラルブロッキング実装においては，Intel Xeon CascadeLake で良好な性能が得られた一方で，A64FX では性能向上を確認することができなかったが，その原因調査を通じて今後のプログラム開発において有益な知見を得ることができた．

1. はじめに

2020 年 11 月の Top500[3] にて 1 位となったスパコン富岳に採用されたプロセッサである A64FX は，Arm v8.2-A 命令セットアーキテクチャを拡張した Scalable Vector Extension (SVE) を初めて実装したプロセッサである．東京大学情報基盤センターが 2021 年 5 月に運用を開始する Wisteria/BDEC-01 も A64FX を搭載する予定であるなど，A64FX を採用した計算環境の導入が国内外で広く進められているが，A64FX がどのようなアプリケーションを得意とし，どのような最適化が有効であるのかなどの評価事例は，X86 系のプロセッサなどと比較すると絶対的に少なく，その性能特性を明らかにすることは喫緊の課題である．

そこで本研究では A64FX の性能特性を明らかにすることを目的とする．性能特性を明らかにするため，流体シミュレーション等の科学技術計算において頻出する計算パターンである反復型ステンシル計算，その中でも最も基本的な三次元の拡散方程式のカーネル（7 点ステンシル）をベースとし，テンポラルブロッキング [4], [5], [6], [7] を始めとした各種最適化を施した実装を用意する．テンポラルブロッキングは，計算対象空間をキャッシュメモリに収まる程に細かく空間分割し，複数反復の時間ステップ分の計算をキャッシュメモリ内で完結することにより，メモリアクセスの局所性を高め，メインメモリへのアクセス回数を軽減することで，一般的にメモリ律速となるステンシル計

算を高速化する手法である．しかしテンポラルブロッキングは，局所性を高めて高速化を達成するという性質上，レジスタやキャッシュメモリの多寡など，プロセッサの特性が性能に大きく影響するため，プロセッサの性能特性を知る上で適した評価対象である．

本稿では，7 点ステンシルに対し一連の最適化を適用した実装を，Intel Xeon CascadeLake (CLX), Intel Xeon Phi Knights Landing (KNL), A64FX それぞれで実行し，比較評価を行った．その結果として，A64FX は CLX とともに異なる性能特性を示すことが明らかとなった．特に一連の最適化から，分岐のペナルティやハードウェアプリフェッチが外れた際のペナルティが相対的に大きいことが明らかとなり，また CMG 間の通信が性能劣化の原因となりやすいこと，パイプライン処理の効率化により性能向上が得られる可能性などが明らかとなった．また，本研究で実装したテンポラルブロッキング手法は，CLX では非常に効果的であることが示されたが，A64FX では他の手法を検討すべきであることが明らかとなった．

2. テンポラルブロッキング

テンポラルブロッキングの概要について示す．テンポラルブロッキングは，反復型ステンシル計算の空間ループと時間ループをブロック化することにより，データの局所性を高める手法である．反復型ステンシル計算では，ある格子点の時間 $t+1$ における値の計算に，隣接格子点の時間 t における値を利用するため，時間ループのブロック化を行う際にはこの計算順序の依存関係を考慮する必要がある．

¹ 東京大学情報基盤センター
Information Technology Center, The University of Tokyo

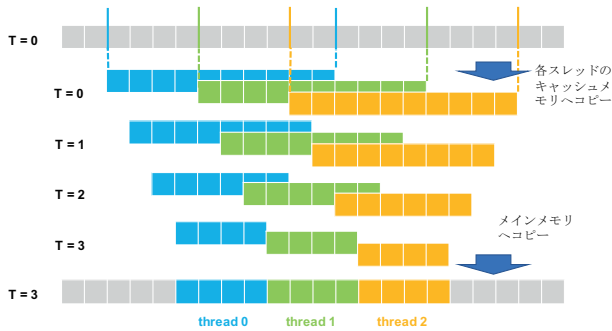


図 1 Overlapped 型のテンポラルブロッキングの実装例。

例えば最も単純な一次元の 3 点スタencilを例とすると，時間 $t+1$ ，座標 x の値 $f_{t+1,x}$ は次の式から計算される。

$$f_{t+1,x} = af_{t,x-1} + bf_{t,x} + cf_{t,x+1}$$

この時，計算に必要な隣接点までの距離をスタencil半径と言い，3 点スタencilでは一つ隣の点まで参照するためスタencil半径 = 1 である．3 点スタencilの計算順序の依存関係を保ったまま時間ブロッキングを行い，かつスレッド並列化を行う実装例を図 1 に示す．各スレッドは時間 $T=3$ で担当する幅 4 の領域を計算するために，時間 $T=0$ にて幅 $10(= 4+1(\text{スタencil半径}) \times 3(\text{時間ステップ数}) \times 2)$ の領域にアクセスする必要がある．しかしながら一度キャッシュメモリにコピーしてしまえば， $T=1$ ， $T=2$ ， $T=3$ の計算は各スレッドのキャッシュメモリ内で完結するため，結果としてメインメモリへのアクセスが削減され，高速化が期待される．図 1 の実装方式は，Overlapped 型のテンポラルブロッキングと呼ばれ，スレッド間で計算領域が重複し，冗長な計算を必要とする手法である．冗長計算を行わない実装方式も複数提案されており，図 2 に示した Overlapped 型以外の方式は冗長計算を必要としないテンポラルブロッキング手法である．冗長計算を必要としない手法は計算負荷の観点で有利だが，ブロック間の計算順序に依存関係があるが故に，並列化の際には高い並列度が得られず，スレッド間のデータ転送や同期を必要とし，メモリの連続的なアクセスを妨げるなどのデメリットがある．一方 Overlapped 型は各ブロックの計算が独立しているためにスレッド間のデータ転送や同期を必要とせず，メモリアクセスが比較的連続的であるなどのメリットがある．デメリットたり得るメインメモリへのアクセス領域の重複は，コア間で共有される L2 キャッシュメモリの効果などにより限定的であり，演算の重複はプロセッサの演算性能とメモリ性能の比によっては無視し得る．プロセッサのメニーコア化による演算性能の向上に対してのメモリ性能の向上の相対的鈍化が見られる昨今の計算環境を鑑み，本研究では Overlapped 型のテンポラルブロッキングを採用する．

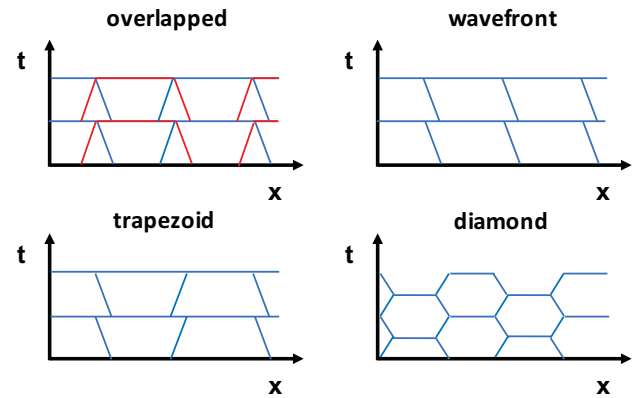


図 2 テンポラルブロッキングの形状の例．(研究 [4] より)．

3. 反復型スタencil計算の最適化

テンポラルブロッキングとは，データの局所性を高めることにより，メモリバンド幅の壁を超えるための最適化手法であると位置付けられる．テンポラルブロッキングを施した実装との比較にあたり，テンポラルブロッキングを行わない実装において，どれだけの性能が得られるかを確かめる必要がある．そこで本研究では，一般に知られるスタencil計算むけの一連の最適化を施した，以下に示す実装を用意した．

3.1 ベースライン実装 (BASE)

図 3 にベースラインの実装を示す．対象の反復スタencil計算は，時間 $t+1$ の値 $f_{t+1}(x, y, z)$ を時間 t の 7 点： $f_t(x, y, z)$ ， $f_t(x-1, y, z)$ ， $f_t(x+1, y, z)$ ， $f_t(x, y-1, z)$ ， $f_t(x, y+1, z)$ ， $f_t(x, y, z-1)$ ， $f_t(x, y, z+1)$ から求める，7 点スタencilと呼ばれる問題である．境界条件にはディリクレ境界条件を用いており，境界条件上の値を代わりに利用する．例えば x が 0 の境界においては， $f_t(x-1, y, z)$ の代わりに $f_t(x, y, z)$ が利用される．空間ループにおける最外ループである z ループを OpenMP スレッドにより並列化する．

3.2 ファーストタッチ (FT)

物理メモリの割当は，malloc によるメモリ確保時ではなく，最初にメモリアクセスが発生した時点で行われる．3.1 節の実装では，図 3 の $f1.t$ の初期化をマスタースレッドのみが行う実装としたため，マスタースレッドに近い物理メモリにデータが割り当てられる．この場合，マスタースレッドを実行するコアから遠いコアが担当するスレッドのメモリアクセスにペナルティが発生する．これを解決するため，各スレッドが担当する領域を初期化しよう 3.1 節の実装を変更する．

```

1  do {
2  #pragma omp parallel for
3      for (int z = 0; z < nz; z++) {
4          for (int y = 0; y < ny; y++) {
5              for (int x = 0; x < nx; x++) {
6                  int c = x + y * nx + z * nx * ny;
7                  int w = (x == 0) ? c : c - 1;
8                  int e = (x == nx-1) ? c : c + 1;
9                  int n = (y == 0) ? c : c - nx;
10                 int s = (y == ny-1) ? c : c + nx;
11                 int b = (z == 0) ? c : c - nx * ny;
12                 int t = (z == nz-1) ? c : c + nx * ny;
13                 f2_t[c] = cc * f1_t[c]
14                     + cw * f1_t[w] + ce * f1_t[e]
15                     + cs * f1_t[s] + cn * f1_t[n]
16                     + cb * f1_t[b] + ct * f1_t[t];
17             }
18         }
19     }
20     double *tmp = f1_t;
21     f1_t = f2_t;
22     f2_t = tmp;
23     time += dt;
24 } while (time + 0.5*dt < 0.1);

```

図 3 7点ステンシルのナイーブ実装

3.3 ループピーリング (PEEL)

3.2節の実装では、最内の x ループ内で境界条件を判定する分岐が三項演算子によって実装されている (図 3 の 7-12 行目)。分岐によって生じるペナルティを最小化するため、ループピーリング (branch hoisting と呼ばれる) を行う。最内ループを $x = 0, 0 < x < nx - 1, x = nx - 1$ の区間に区分することで、最内ループとなる $0 < x < nx - 1$ の範囲において分岐を行わない実装が可能である。これを 3.2 節の実装に加えたものが本実装である。

3.4 OpenMP スレッド割当手法の変更

3.3 節の実装にて最外の z ループに施されていた OpenMP のスレッド割当を変更する。ここでは、以下の 2 つの割当方法を用いる。なお、7 点ステンシルのループ分割手法の変更に伴い、ファーストタッチ時のループ分割手法も変更している。

3.4.1 y 軸ループ並列 (Ydim)

z 軸ループの代わりに、 y 軸ループを並列化する。この際、`#pragma omp parallel` は引き続き z ループの直上に配置し、 y ループの直上に `#pragma omp for nowait` 配置する。`nowait` は、`#pragma omp for` 終了時の暗黙の同期を行わないためのキーワードである。この変更により、全てのスレッドが同一の x - y 平面から計算を開始するため、L2 キャッシュなどのコア間で共有されるキャッシュの効率的な利用が期待される。

3.4.2 y - z 軸ループ並列 (Y-Zdim)

z 軸ループ単体ではなく、 y - z のループ双方を並列化す

```

1 #pragma omp parallel
2 #pragma omp for \ \ CMG 分割
3     for (int z = 0; z < nz; z++) {
4 #pragma omp for \ \ Thread 分割
5         for (int y = 0; y < ny; y++) {
6             for (int x = 0; x < nx; x++) {

```

図 4 A64FX における y - z ループ並列の概念。実際には上記のような指示文の指定は不可。スレッド番号を参照して手動により分割を行う。

る。図 4 に並列化の y - z ループ並列の概念を示す。例えば A64FX の場合、12 コア (12 スレッド) が一つの CMG と呼ばれる単位に配置され、この CMG が 4 つで 48 コア/1 ノードを構成している。ここでは、 z 軸ループは CMG 単位で分割し、 y 軸ループは CMG 内の 12 スレッドで分割している。

3.5 Intrinsics 利用 (INT)

3.4.1, 3.4.2 節の実装の内より高速であったものを、`intrinsics` を使って書き換える。AVX512, SVE 向けの実装をそれぞれ用意する。

3.6 FMA 不使用によるパイプライン実行効率化 (w/oFMA)

3.5 節の実装をベースにパイプライン実行の効率化を狙った最適化を行う。図 5 は、SVE を用いた素朴な実装例である。 $d = \text{svmad}_x(a, b, c)$ は SVE における積和演算 (Fused Multiply-Add; FMA) 命令で、 $d = ab + c$ を計算する。本来 2 命令で行われる積と和の演算を 1 命令で実行でき、プロセッサの理論演算性能を達成するためには必須の命令である。しかし図 5 の素朴な実装の 14-19 行の FMA 命令は、引数として直前で計算された結果である `tmp0` を受け取っているため、この依存関係によりパイプライン的に実行することが出来ない。従って FMA 命令のレイテンシを隠すことが出来ず、性能低下を引き起こす可能性がある。

これを改善するために、図 5 の 13-19 行を図 6 のように書き換える。この実装では敢えて FMA 命令を使わず、積・和それぞれの命令を用いて実装することで、命令間の依存関係を緩和する。足し算部分の依存関係を完全に解消することはできないが、ツリー状に足し込みを行うことで最大限の並列性を確保する。

3.7 ループアンローリング (UNR)

3.6 節の実装をベースとして、最内の x 軸ループをアンローリングする。ループ終了の条件判定回数を減らす効果よりも、ここでの狙いは 3.6 節と同様であり、1 ループ内で実行される独立な命令数を増やすことで、パイプライン実行の効率化を図る。

```

1  /*      f2_t[c] = cc * f1_t[c]          */
2  /*      + cw * f1_t[w] + ce * f1_t[e]  */
3  /*      + cs * f1_t[s] + cn * f1_t[n]  */
4  /*      + cb * f1_t[b] + ct * f1_t[t]; */
5  cc_vec = svdup_f64(cc);
6  ...
7  ct_vec = svdup_f64(ct);
8  svbool_t pg = svptrtrue_b64();
9  ...
10 fc_vec = svld1(pg, (float64_t*)&f1_t[c]);
11 ...
12 fct_vec = svld1(pg, (float64_t*)&f1_t[c+t]);
13 tmp0 = svmul_x(pg, cc_vec, fc_vec);
14 tmp0 = svmad_x(pg, cw_vec, fct_vec, tmp0);
15 tmp0 = svmad_x(pg, ce_vec, fct_vec, tmp0);
16 tmp0 = svmad_x(pg, cs_vec, fct_vec, tmp0);
17 tmp0 = svmad_x(pg, cn_vec, fct_vec, tmp0);
18 tmp0 = svmad_x(pg, cb_vec, fct_vec, tmp0);
19 tmp0 = svmad_x(pg, ct_vec, fct_vec, tmp0);
20 svst1(pg, (float64_t*)&f2_t[c], tmp0);

```

図 5 7 点ステンシル計算部のナイーブな SVE 実装例．変数名末尾に_vec と付与された変数は svfloat64_t 型の変数である．

```

1  fc_vec = svmul_x(pg, cc_vec, fc_vec);
2  fce_vec = svmul_x(pg, ce_vec, fce_vec);
3  fcw_vec = svmul_x(pg, cw_vec, fcw_vec);
4  fcn_vec = svmul_x(pg, cn_vec, fcn_vec);
5  fcs_vec = svmul_x(pg, cs_vec, fcs_vec);
6  fcb_vec = svmul_x(pg, cb_vec, fcb_vec);
7  fct_vec = svmul_x(pg, ct_vec, fct_vec);
8  tmp0 = svadd_x(pg, fce_vec, fcw_vec);
9  tmp1 = svadd_x(pg, fcn_vec, fcs_vec);
10 tmp2 = svadd_x(pg, fct_vec, fcb_vec);
11 tmp0 = svadd_x(pg, fc_vec, tmp0);
12 tmp1 = svadd_x(pg, tmp1, tmp2);
13 tmp0 = svadd_x(pg, tmp0, tmp1);

```

図 6 図 5 の 13-19 行目を，FMA を使わず積・和それぞれの命令で実装した例．

3.8 レジスタブロッキング及びアラインドアクセス (REG)

3.6, 3.7 節の実装の内高速だったものをベースとして，キャッシュメモリへのアクセス負荷を軽減するために， x 軸方向のレジスタブロッキングを適用する．ベクトル化済みのコードの x 軸方向のメモリアクセスを考えた時，ベクトル長が 512bit であるなら， $f2_t(x : x + 7, y, z)$ の計算に $f1_t(x - 1 : x + 6, y, z)$ ， $f1_t(x : x + 7, y, z)$ ， $f1_t(x + 1 : x + 8, y, z)$ へのアクセスが必要となる．この際，キャッシュラインサイズを踏まえると， $x = x + 8$ となる次ステップで利用することとなる $f1_t(x + 8 : x + 15, y, z)$ を同時にレジスタに読み込んでおくことで効率化が期待できる．加えて AVX512 の命令セットを利用できる環境では，_mm512_alignr_epi64() 命令を利用することで， $f1_t(x - 8 : x - 1, y, z)$ ， $f1_t(x : x + 7, y, z)$ ， $f1_t(x + 8 : x + 15, y, z)$ が保存された 3 つのレジスタ

から， $f1_t(x - 1 : x + 6, y, z)$ や $f1_t(x + 1 : x + 8, y, z)$ ，さらに境界部で必要となるデータがセットされたレジスタを用意することができる．これによって，非アラインドなメモリアクセスとなってしまう $f1_t(x - 1 : x + 6, y, z)$ や $f1_t(x + 1 : x + 8, y, z)$ のメモリアクセスを回避することができる．

3.9 タイリングによるキャッシュ利用効率化 (TILE)

3.8 の実装をベースに，タイリングによるキャッシュメモリの利用効率化を図る． x - y 平面をキャッシュに乗るサイズに区分し，サイクリックにスレッドへ割り当てることで， $f1_t(x, y, z + 1)$ のデータがキャッシュから追い出されないようにする．この最適化の狙いは 3.4.1 と同様であり， x - y 平面がキャッシュに乗るほど十分に小さければ必要がなく，タイル切り替え時にメモリアクセスの連続性が失われることから，むしろ性能劣化を引き起こす可能性がある．

3.10 テンポラルブロッキング (TB)

3.9 の実装をベースとし，テンポラルブロッキングを適用することで，メモリ性能の壁を超えることを目標に実装する．節 2 で書いたとおり，実装には Overlapped 型を用いる．空間方向のブロッキングは x - y 平面に対して行い，従って x , y 軸の正負双方向に計算領域の重複が発生する．図 7 に x - y 平面の分割方法と，メモリアクセスの重複を図示する．時間ブロックの終了する時間 $T = 4$ の時に，各スレッドが計算結果の書き込みを行う担当領域はブロックサイズ $BX \times BY$ で区分された領域であり，重複はない．時間 $T = 0$ で発生するメモリアクセスの重複は同一の x - y 平面上で発生するため，L2 キャッシュなどで共有されることが期待される．一方 z 軸方向は基本的に分割せず，1 スレッドが担当する． z 軸方向への計算の進行は，図 8 のように行う．時間ブロック幅 -1 個分の一時領域を用意する必要があり，そのサイズは z 軸方向に (ステンシル半径 $\times 2 + 1$)， x , y 軸方向にそれぞれ $(BX + \text{ステンシル半径} \times 2)$ ， $(BY + \text{ステンシル半径} \times 2)$ の幅を持てば，本アルゴリズムのテンポラルブロッキングを行うに十分である．即ち必要な一時領域のメモリサイズ S は，対象問題を倍精度，ステンシル半径を R ，時間ブロック幅を TB とした時， $S = 8 \times (TB - 1) \times (2R + 1) \times (2R + BX) \times (2R + BY)$ となる．

4. 計算機環境

本研究において使用した 3 種類の計算環境を表 1 に示す．FX は A64FX を指し，東京大学情報基盤センターに導入されている Fujitsu PRIMEHPC FX700 の内 1 ソケットを用いる．コンパイラには Fujitsu C/C++ Compiler 4.2.0 を用いる．コンパイルは trad モードと LLVM ベー

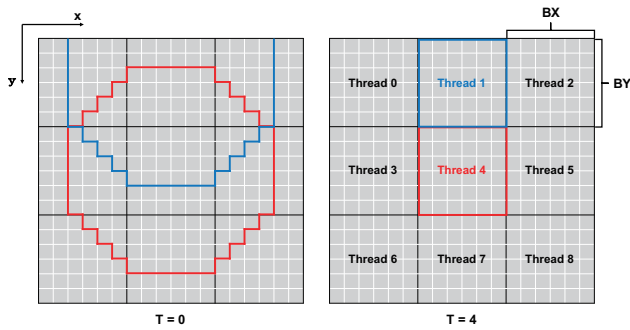


図 7 時間ブロック幅 = 4 の時の x - y 平面の時刻 $T = 4$ における空間ブロッキングとスレッド割当 (右), 及び $T = 0$ におけるスレッド 1, 4 のアクセス範囲 (左)。

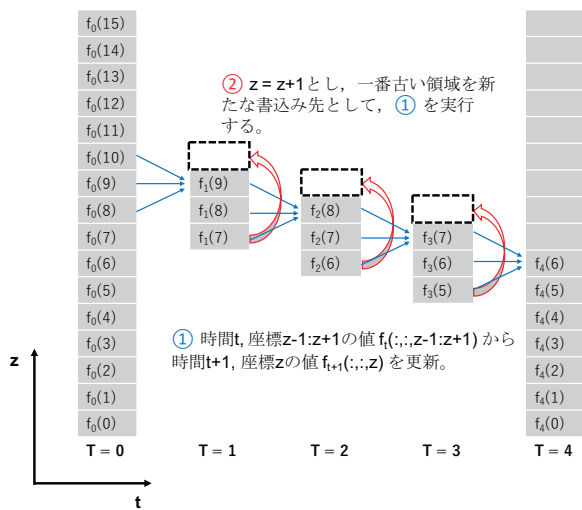


図 8 テンポラルブロッキングの z 軸方向への計算進行アルゴリズム。

スの clang モードを選択でき、本研究では両者を用いる。コンパイル時オプションとしては、trad モードの際には `-Kfast,openmp -O3 -Khptag -Nfjompilib -Nfjprof` を用い、clang モードの際には `-Kfast,openmp -O3 -Nclang` を用いた。実行時の環境変数としては、どちらのモードの場合にも `FLIB.FASTOMP=TRUE; FLIB.HPCFUNC=TRUE; XOS.MMM.L.PAGING.POLICY=demand:demand:demand` を指定した。

CLX は同じく東京大学情報基盤センターに導入されている Oakbridge-CX に搭載された CPU である。Oakbridge-CX は 1 ノードあたり CLX を 2 ソケット搭載しているが、計算に用いたのはそのうち 1 ソケットのみである。コンパイラには `icc 19.0.5.281` を使用した。コンパイル時オプションとしては `-xCORE-AVX512 -qopenmp -O3 -std:c99` を指定した。実行時の環境変数としては `KMP.AFFINITY=compact` を利用した。1 ソケットのみを使用するために、`numactl -cpubind=0 -membind=0` を実行コマンドの直前に指定している。

KNL は最先端共同 HPC 基盤施設 (JCAHPC) の運用す

表 1 計算環境概要

略称	FX	CLX	KNL
名称	Fujitsu A64FX (1.8GHz)	Intel Xeon Platinum 8280 (CascadeLake)	Intel Xeon Phi 7250 (Knights Landing)
コア数	48	28	68
理論演算性能	2,765 GFlops	2,419 GFlops	3,046.4 GFlops
主記憶容量	32GB	96GB	16GB
メモリバンド幅性能	809 GB/sec	101 GB/sec	490 GB/sec

る Oakforest-PACS の 1 ノードを用いている。KNL は通常の DDR4 メモリの他、高速な三次元積層メモリ MCDRAM を搭載しており、表 1 の主記憶要領・メモリバンド幅性能は MCDRAM のものである。また KNL のメモリモード・サブ NUMA クラスタリングモードは、それぞれ Flat・Quadrant モードを使用しており、本稿における KNL での実行は全て同モードで行われたものである。コンパイラには `icc 19.0.5.281` を使用した。コンパイル時オプションとしては `-xMIC-AVX512 -qopenmp -O3 -std:c99` を指定した。実行時の環境変数としては `KMP.AFFINITY=compact` を利用した。MCDRAM のみを使用するために、`numactl -membind=1` を実行コマンドの直前に指定している。環境変数には、`LMPI.PIN.DOMAIN=256; LMPI.PIN.PROCESSOR.EXCLUDE.LIST=0,1,68,69,136,137,204,205` を指定している。

なお、表 1 中のメモリバンド幅性能は、[2] より取得したベンチマークプログラム (OpenMP 並列) の Stream Triad の実測値を示している。

5. 性能評価

7 点ステンシルを用いた性能評価を 4 節で示した計算環境で行った。特に言及のない限り、演算精度は倍精度であり、問題サイズ $NX = NY = NZ = 256$ である。性能指標には、メモリバンド幅性能と FLOPS 値を用いる。ここで、メモリバンド幅性能 (Throughput) と FLOPS 値は、以下の式により求める。

$$Throughput = 8 * NX * NY * NZ * NT * 2 / ETIME$$

ここで $NX \times NY \times NZ$ は対象とする問題サイズであり、 NT は反復回数、 $ETIME$ は実行時間を表す。7 点ステンシルの最内ループでは、格子点 7 点を読み込み 1 点への書き込みを行うが、ここではメインメモリからのデータ読み込みは各格子点につき高々一度しか行われず、全てキャッシュに乗るものと仮定し、従ってメインメモリへのアクセスは読み書き一度ずつの計 2 回として Throughput を計算する。この指標においては、テンポラルブロッキング実装の throughput が表 1 に示したメモリバンド幅性能を上回り得る。

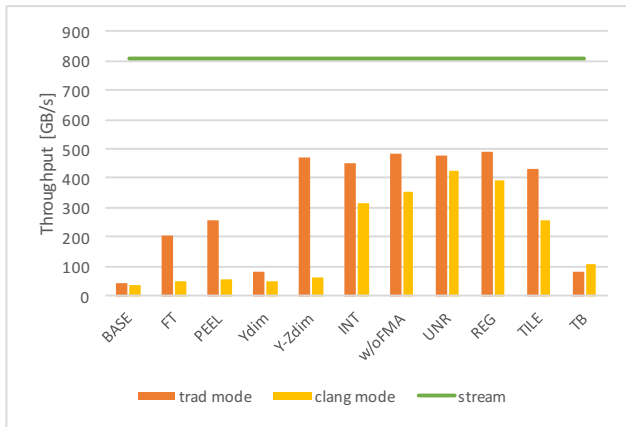


図 9 FX における 7 点ステンシルの性能。

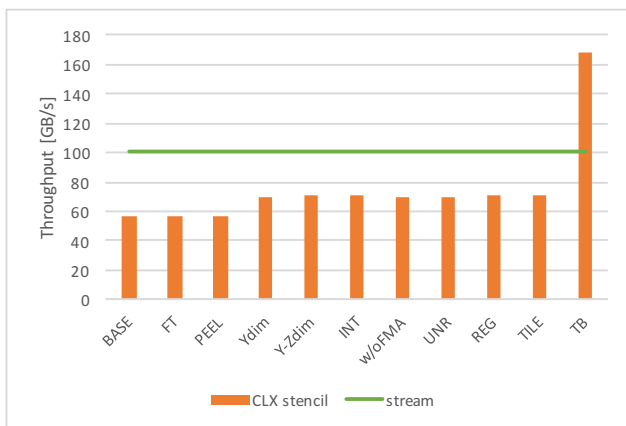


図 10 CLX における 7 点ステンシルの性能。

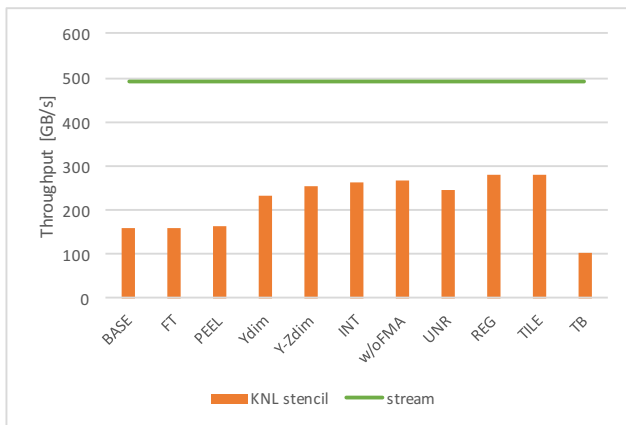


図 11 KNL における 7 点ステンシルの性能。

図 9, 図 10, 図 11 にそれぞれ, FX, CLX, KNL で評価を行った結果を示す。グラフは 11 回の計測を行った中央値を示している。また, 各グラフにおける緑の直線は, 表 1 に示したメモリバンド幅性能である。グラフ中の BASE, FT などの略称は, 3 節の実装と対応している。

BASE と FT の比較から, CLX と KNL においては殆ど性能上の変化がない (1%以下) に対し, FX の特に trad モードでは 41.7GB/sec から 207.8GB/sec へとおよそ 5 倍の性能向上が見られることから, FX においてはファース

トタッチが必須であることがわかる。FX は 4 つの CMG (Core Memory Group) から構成されており, CMG 間はリングバスによって接続されているものの, その性能は >115GB/sec 程度であり, メインメモリの性能 (CMG 当たり 256GB/sec, 4CMG で 1,024GB/sec の理論性能) に劣ることを踏まえると, 妥当な結果である。

FT と PEEL を比較すると, FX は相対的に分岐に弱いこともわかる。CLX では性能差が 1%以下で, KNL の性能向上が 3%程度であるところ, FX の trad モードでは 24%程度の性能向上が見受けられる。この差がアーキテクチャ的な特性から生じたのか, コンパイラなどソフトウェア面の習熟度によるものなのか, 現時点では分かっていない。

Ydim の性能を比べると, FX は他のプロセッサと正反対の反応を示している。一般的なマルチコアの CPU では, y 軸ループを分割することで, L2 キャッシュによるコア間でのデータ再利用性が高まり, CLX と KNL がそれぞれ PEEL と比較して 23%, 40%の性能向上を達成しているように, 性能向上が期待できるが, FX では性能が大きく落ち込んでいる。これは FT と同様, CMG 間のデータ転送が影響している。コア間のデータ共有性が高まったことで, CMG 間のデータ転送が増加してしまったことが性能低下の主要因であり, 実際に 1CMG のみの実行の場合には性能は向上する。

この CMG 間のデータ転送を削減しつつ, CMG 内のデータ共有性を高める手法が, Y-Zdim である。 z 軸ループを CMG 単位で分割することで, CMG を跨いだキャッシュヒットが起こることを防いでいる。これによって, FX の trad モードの性能が劇的に改善していることがわかる。PEEL と比較して 1.8 倍程度の性能向上が見られる。この実装は図 4 にあるとおり, 指示文のみでは実装できないが, 今回実装した intrinsics を使わない内では最速であり, OpenMP のみで FX の 48 スレッド全てを利用する場合には適用すべき最適化であると言える。

次の INT からは intrinsics を使った実装となるが, CLX と KNL では殆ど性能に寄与していないことがわかる。FX においては, trad モードにて Y-Zdim と比較して 4%程度性能が低下した一方, clang モードでは 5 倍以上の性能向上を確認した。clang モードでは付属の詳細プロファイラが使えない都合上, 性能向上の正確な理由を探るのは難しいが, 単純にベクトル化が促進されたことと, 直接 intrinsics を書くことで効率の悪いコード変換が抑制されたためだと考えられる。

w/oFMA は, CLX で若干の性能減, KNL において 3%弱の性能向上に留まった一方, FX では INT と比較して trad モードで 8%程度, clang モードで 13%程度の性能向上が得られた。A64FX の仕様書 [1] を見るに, 浮動小数点の加算, 乗算, FMA のいずれもレイテンシは 9 cycle であり, 直感的には加算・乗算をそれぞれ別に行った場合の方が効

率は悪そうであるが、FX ではパイプラインを埋めるためにこのような変更を行うことが有効な場合もあるようだ。この変更により、詳細プロファイラが示す”Floating-point operation wait”がおよそ 40%減少したことを確認した。

UNR は、CLX、KNL、FX の trad モードで性能の低下が見受けられたが、clang モードでは性能向上が得られている。3.7 節で書いた通り、本最適化はループ内の独立に実行可能な命令を増やすことで、パイプラインの効率化を狙ったものであり、2 ループ分を手動で展開することで実装した。clang モードでは詳細プロファイラを利用できないため、trad モードを調査した限り、狙い通り”Floating-point operation wait”はさらに減少していた一方で、”Floating-point load L1D cache access wait”が増加してしまった。2 ループ分を同時実行した場合、係数の 7 変数に 2 ループ分の 14 変数を加え、少なくとも 21 のベクトルレジスタが同時に利用されることとなる。A64FX のベクトルレジスタ数は 32 であるため、次ステップ分のベクトル要素を読み込むためのベクトルレジスタが不足した結果、L1D cache へのアクセス待ちが増加した可能性が考えられる。4 ループ分手動展開した際に劇的な速度低下を起こしたことからも、レジスタ数に気を配る必要があるようである。

REG では、FX の trad モードで 1%程度の性能向上、KNL で w/oFMA と比較して 4%程度の性能向上が確認されたが、clang モードでは 8%程度の性能低下が見受けられた。KNL では全てのメモリアクセスが aligned される効果があるが、SVE に aligner 命令に相当する命令は無いため、最適化の効果は限定的であったようだ。

TILE では、CLX、KNL で性能の微増が見られた一方、FX では trad モード、clang モード双方で大きな性能低下が見られた。詳細プロファイラを参照したところ、原因は L1D/L2D miss demand rate の大幅な増加（5%前後から 25%程度まで）、ハードウェアプリフェッチの効きが悪くなったことである。タilingを行い計算領域を小領域に分割した場合、キャッシュの利用効率が良くなる一方で、次のタイルへ処理を移す瞬間にメモリアクセスの連続性が失われるために、ハードウェアプリフェッチが効かなくなる。FX ではこのハードウェアプリフェッチが効かない場合のペナルティが相対的に大きいと言える。これを解決するために、コンパイラのオプションや intrinsics の利用によるソフトウェアプリフェッチを試みたが、効果的なソフトウェアプリフェッチ手法はまだ見つからない。

最後に TB であるが、CLX では stream の性能を大幅に上回る理想的な結果を得られた一方で、KNL と FX では大幅な性能低下を観測した。CLX で調査した範囲内では BX=64、BY=19、TB=4 のブロックサイズの際に最高性能となり、TILE と比較して 2.36 倍を記録した。このとき必要な一時領域のサイズを計算すると 140KB であり、コア当たり 32KB の L1 キャッシュには収まらないものの、L2

キャッシュメモリには十分収まる。KNL・FX でも L1 または L2 キャッシュに収まる範囲でブロックサイズを調整したが、性能低下にはもっと他の根本的な要因が存在すると思われる。まず、メモリアクセスパターンが複雑化したことで、ハードウェアプリフェッチは全く効かなくなり、L1D/L2D miss demand rate は 90%以上となった。また、複雑なインデックスを計算するために整数を多数使っているが、”Integer load L1D cache access wait”が増大した。Integer の配列は読み込んでいないため、変数を整数レジスタに置ききれず、レジスタスビルを起こしてしまったことも要因と考えられる。またそもそも論として、w/oFMA の最適化が効果的だったという事実が示す通り、演算レイテンシが隠れておらず、完全なメモリ律速となっていない。この状態からさらに冗長な浮動小数点演算を大幅に増やしてしまう Overlapped 型のテンポラルブロッキングを適用しても、期待する効果は得られないのでは無いかと考えられる。

6. おわりに

本研究では、A64FX の性能特性を明らかにするために、最も基礎的な 3 次元ステンシル計算である 7 点ステンシルをベースとして、テンポラルブロッキング始めとした一連の最適化を適用し、Intel Xeon CascadeLake や Intel Xeon Phi Knights Landing との比較評価を行った。評価結果が示す通り、A64FX は Xeon CPU とも KNL とも違う性能特性を示した。特に、CMG 間通信のペナルティ、分岐処理のペナルティ、パイプライン実行の効率化の重要性、ハードウェアプリフェッチが外れた際のペナルティなどを考慮する重要性が示された。

また、本研究で用いたテンポラルブロッキングの実装手法は、CLX においてテンポラルブロッキングなしでは超える事のできないメモリ性能の壁を大幅に上回る、非常に良好な結果が得られたが、KNL と FX では効果が得られなかった。今後、Overlapped 型以外の手法を適用する事で、KNL や FX のようなメニーコアプロセッサにおいて有効なテンポラルブロッキング手法についても検討する。

謝辞 本研究の一部は、JSPS 科研費 20H00580 の助成を受けたものです。

参考文献

- [1] A64FX: <https://github.com/fujitsu/A64FX>.
- [2] STREAM: Sustainable Memory Bandwidth in High Performance Computers: <http://www.cs.virginia.edu/stream/>.
- [3] The Top 500 List: <https://top500.org/>.
- [4] 勝汰黒田, 敏夫遠藤, 聡 松岡: ディレクティブによる時間ブロッキングの自動適用, 技術報告 18, 東京工業大学, 東京工業大学 (2016).
- [5] 真平佐藤, 幸紀佐藤, 敏夫遠藤: テンポラルブロッキングを適用したステンシル計算コードの SIMD 化とループライ

ンモデルを用いた性能解析, 技術報告 17, 東京工業大学学術国際情報センター / JST CREST, 東京工業大学学術国際情報センター / JST CREST, 東京工業大学学術国際情報センター / JST CREST (2015).

- [6] 猛 深谷, 武史岩下: タイルレベルの並列処理を可能とする時空間タイリング手法を用いた 3 次元 FDTD カーネルの実装と性能評価, 技術報告 35, 北海道大学情報基盤センター, 北海道大学情報基盤センター (2017).
- [7] 猛 深谷, 武史岩下: Knights Landing における Tiled 3D FDTD カーネルの性能評価, 技術報告 6, 北海道大学情報基盤センター, 北海道大学情報基盤センター (2018).