Vol.2021-HPC-182 No.4
2021/12/6

A64FXに向けた NeK CFD solver における axhelmカーネルの最適化と評価

辻 美和子[1,a)]　佐藤 三久[1]

概要：Nek5000/RS は，スペクトル要素法に基づくオープンソースの計算流体力学ソルバである．本稿では，A64FX プロセッサに対する最適化技術の研究のために，Nek5000/RS におけるヘルムホルツ行列ベクトル積の計算を行うカーネル axheml の A64FX プロセッサ上での性能評価および最適化を行う．Axheml カーネルに対して，SIMD 化，ソフトウエアパイプライニング，連続メモリアクセス化，そしてプリテッチといった最適化を行い，性能解析ツールなどを用いて効果を検証する．

1. はじめに

近年，アームアーキテクチャは HPC システムに広く採用されている．特に，A64FX プロセッサは，スーパーコンピュータ富岳のために設計されたアームアーキテクチャのマイクロプロセッサであるが，ストーニーブルック大学の Ookami HPE Apollo 80 システムや名古屋大学のスーパーコンピュータ「不老」など複数のシステムで使用されている．

A64FX プロセッサは 158,976 ノードからなる超並列システム・富岳のために設計されたため，消費電力を抑えるためにコンパクトな設計となっており，コア内の計算リソースが比較的少ない一方で，演算のレイテンシは比較的長い．このような A64FX プロセッサの性質から，いくつかのアプリケーションは as-is のコードでは，A64FX 上で性能を発揮できないことがある．

本稿では，オープンソースの計算流体力学 (computational fluid dynamics, CFD) コードである Nek5000 および NekRS から重要な関数を集めたベンチマークセット nekBench に収録されている axheml カーネルを対象に，A64FX プロセッサ向けの性能チューニングのケーススタディを行う．axheml カーネルの最適化と性能評価を通して，広く用いられる最適化技術 — SIMD 化，ソフトウエアパイプライニング，メモリアクセスパターンの改良，ソフトウエアプリフェッチなど — が，A64FX プロセッサ上でのカーネルの実行性能にどのような影響を与えるかを示す．

[1] 理化学研究所計算科学研究センター
RIKEN Center for Computational Science
[a)] miwako.tsuji@riken.jp

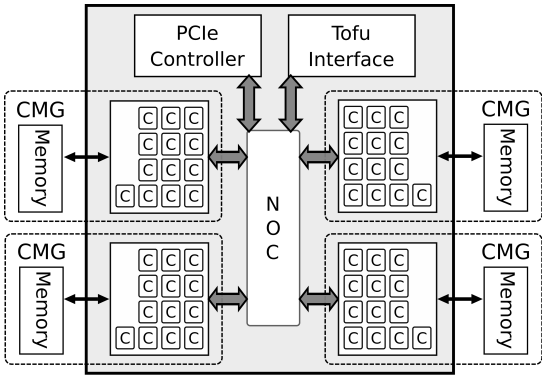


図 1　A64FX の概要.

本稿の構成は以下のようになっている：まず，2 章で A64FX プロセッサを，3 章でケーススタディの対象となる Nek5000/RS/Bench および axhelm カーネルについて述べる．4 章で性能評価および最適化を行い，最後に 5 章でまとめと議論を行う．

2. A64FX プロセッサ

A64FX プロセッサは，Arm8.2 アーキテクチャおよびその SVE (Scalable Vector Extensions) 拡張に基づき設計されたマイクロプロセッサである [6][7][3]．**図 1** に A64FX プロセッサの概要を示し，**図 2** に A64FX プロセッサのコアの詳細を示す．また，**表 1** に仕様のサマリを示す．図 1 に示すように，A64FX プロセッサは， Core Memory Groups (CMGs) と呼ばれリングバスで接続されるグループを 4 つ持つ NUMA アーキテクチャのプロセッサである．各 CMG は 12 の演算コア，1 つのアシスタントコアからなる[*1]．また，各 CMG はバンド幅 256GiB/s で 8GiB の

[*1] 富岳の場合，すべての CMG がアシスタントコアを持つノードと

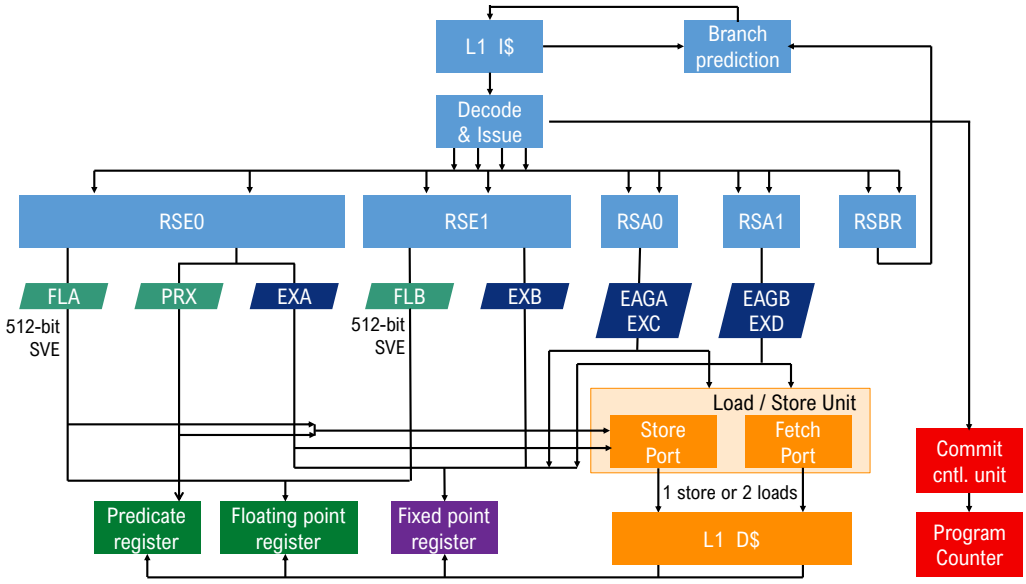


図 2　A64FX のコアの詳細

表 1　A64FX のノード仕様

CPU	A64FX
ISA	Armv8.2 with SVE
Number of cores	48 compute cores and 2 or 4 assistant cores
Base Frequency	1.8/2.0/2.2GHz
SIMD Width	512bit
L1I Cache Size	64KiB/core
L1D Cache Size	64KiB/core
L2 Cache Size	8MiB/CMG
Process Technology	7nm CMOS FinFET
Memory	HBM2
Memory Bandwidth	1024 GB/s
Memory Capacity	32 GiB

HBM2 メモリとメモリコントローラを持つ．演算コアはクロスバーを介して 8MiB の L2 キャッシュを共有し，コアごとに 64KiB の L1 データキャッシュを持つ．プロセッサ全体のキャッシュコヒーレンスは，L2 キャッシュパイプラインによってサポートされている．

図 2 に示すように，各コアは 4 つの ALUs (EXA, EAB, EAC, EAD) を持ち，うち 2 つはアドレス演算ユニット (EAGA, EAGB) を兼ねる．さらに，SVE 命令をサポートする 2 つの浮動小数点演算ユニット (FLA, FLB) と 1 つのプレディケートユニット (PRX) が存在する．浮動小数点演算ユニットは積和演算をサポートする．SVE の仕様は SIMD 幅を定義せず，128 ビットから 2048 ビットまでの 128 ビット刻みの選択肢からベンダが可能である．A64FX では SIMD 幅は 512 ビットとなっており，128 ビットと 256 ビットもサポートされている．動作周波数は 1.8/2.0/2.2GHz であるが，富岳では 2.0 および 2.2GHz のみがサポートされている．計算ノードのデフォルトの動作は 2.0GHz であり，2.2GHz はブーストモードと呼ばれ，ユーザがジョブスクリプトで陽に指定する必要がある．ロードおよびストア命令は L1 キャッシュユニットで行われ，SIMD ロードは 2 命令，SIMD ストアは 1 命令が同時に処理可能である．ロードにおける L1 キャッシュバンド幅は最大で 256GB/s (2.0GHz 実行時) であるが，最大のバンド幅を得るためには連続アクセスかつ 512 ビット SIMD ロード命令でなければならない．

2 つの CMG のみアシスタントコアを持つノードがある

3. Nek5000/RS, NekBench, および ax-helm カーネル

本章では A64FX 向け最適化のケーススタディで用いたカーネルについて紹介する．

3.1 nek5000/RS

Nek5000/RS [8][2] は，非圧縮性流体流，熱対流，燃焼，磁気流体力学などを計算するオープンソースの CFD コードであり，スペクトル要素法 (Spectral element method, SEM) と呼ばれるスペクトル法と有限要素法のハイブリッド手法を用いている．

Nek5000/RS は非圧縮性 Navier-Stokes (NS) 方程式

$$\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u}\cdot\nabla\boldsymbol{u} = -\nabla p + \frac{1}{Re}\nabla^2\boldsymbol{u}, \quad (1)$$

$$\nabla\cdot\boldsymbol{u} = 0, \quad (2)$$

およびエネルギー方程式

$$\frac{\partial T}{\partial t} + \boldsymbol{u}\cdot\nabla T = \frac{1}{Pe}\nabla^2 T, \quad (3)$$

により定義される熱輸送をシミュレートする．ここで $\boldsymbol{u}$ は速度，p は圧力，T は温度，Re はレイノルズ数，Pe はペクレ数である．

3.2 axhelm kernel

本稿で最適化の対象となるカーネル axhelm (BK mode) は，ヘルムホルツ行列ベクトル積を計算する Nek5000/RS におけるスペクトル要素離散化の一部である．

$\underline{u}_L = [\underline{u}^1 \cdots \underline{u}^E]$ をすべての局在基底ベクトルの集合とする．それぞれの $\underline{u}^i$ の大きさは，N を多項式次数として，$(N+1)^3$ である．E は SEM の要素数である．A_L は以下の A^e の block-diag(A^e) である：

$$A^e = \begin{pmatrix} D_1 \\ D_2 \\ D_3 \end{pmatrix}^T \begin{pmatrix} G^e_{11} & G^e_{12} & G^e_{13} \\ G^e_{12} & G^e_{22} & G^e_{23} \\ G^e_{13} & G^e_{23} & G^e_{33} \end{pmatrix} \begin{pmatrix} D_1 \\ D_2 \\ D_3 \end{pmatrix}. \qquad (4)$$

6 つの局所行列 $G^e_{mm'}$ は幾何学的係数であり，各グリッド点に対する 1 つの nontrivial entry を含む対角行列である．行列 D_1, D_2, D_3 は， $\hat{I}$ を $(N+1)^2$ の単位行列，$\hat{D}$ を Gauss-Lobatto-Legendre (GLL) の節点から求積点への 1 次元微分行列写像として，$(\hat{I} \otimes \hat{I} \otimes \hat{D})$, $(\hat{I} \otimes \hat{D} \otimes \hat{I})$, $(\hat{D} \otimes \hat{I} \otimes \hat{I})$ である．詳細は Fischer ら [2] を参照されたい．

axhelm カーネルは以下を計算する：

$$A_L \underline{u}_L \qquad (5)$$

3.3 NekBench

NekBench は，Nek5000/RS から抜粋されたベンチマークカーネルを集めたベンチマークスイートであり，bw, gs,axhelm, dot, nekBone, および adv の 6 つのカーネルが利用可能である [1].

NekBench は，性能ポータビリティのためのオープンソースのライブラリである OCCA[5] を用いている．OCCA のランタイムライブラリはランタイムにカーネルをコンパイルし，コンパイルしたライブラリを動的に呼び出す．ユーザは，OCCA 環境変数を設定することで，コンパイラや最適化オプションを指定することができる．NekBench は特定のカーネルを指定された回数だけ呼び出し，平均実行時間，メモリスループット，GFLOPS/s を出力する．多項式次数 N，SEM における要素数 E，カーネルの呼び出し回数は，プログラムの引数として指定する．

3.4 axhelm kernel のソースコード

オリジナルの axheml カーネルのソースコードを**図 3** に示す．axheml 関数は，3 つの配列 ggeo, D, q を入力として用い，配列 Aq を出力する．パラメータ p_Nq は多項式次数を N とすると $N+1$ である．p_Nq は，典型的には 4 から 14 の値をとる．この数値は，# define ディレクティブにより，事前に与えられる．Nelements は，SEM の要素数であり，3.2 章の E に同じである．本稿は主にもっとも単純かつ平易な p_Nq $= 8 (N = 7)$ の場合を扱う．配列 ggeo は幾何学的係数，D は 1 次元微分行列，q は局所基底ベクトルの集合である．それぞれの配列のサイズは，

表 2 実験リスト

Nelements	# of kernel calls in a run	# of runs	# of MPI processes	# of OMP threads
7680	100	3	48	1
			4	12
3840	200	3	48	1
			4	12
1920	400	3	48	1
			4	12
960	800	3	48	1
			4	12

$\text{p\_Nq}^3 \times \text{N}_{\text{elements}} \times 7$, p_Nq^2, および $\text{p\_Nq}^3 \times \text{N}_{\text{elements}}$ であり，出力される配列 Aq のサイズは $\text{p\_Nq}^3 \times \text{N}_{\text{elements}}$ である．

4. 性能評価と最適化

4.1 実験環境

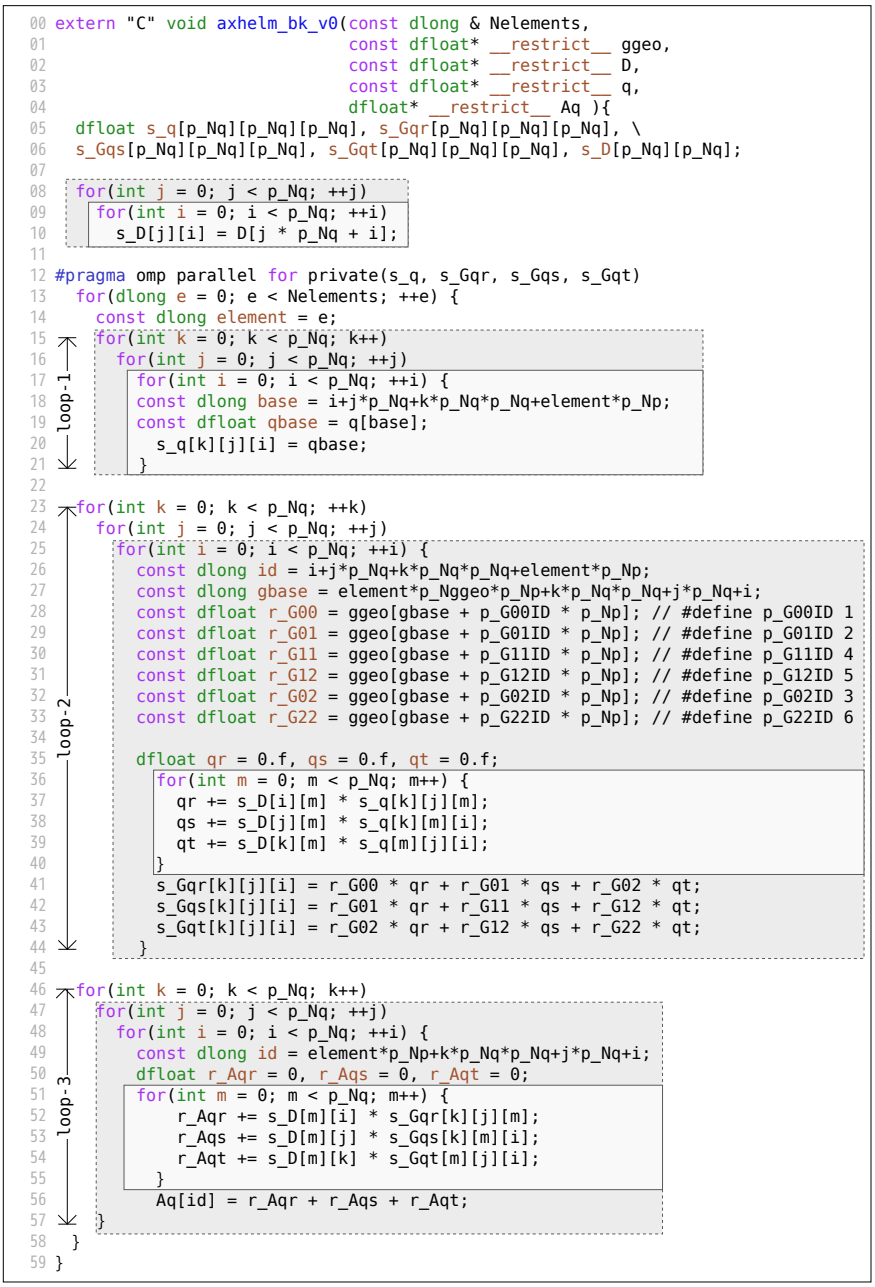
表 2 に評価対象とした問題サイズ，呼び出し回数，試行回数，および並列数を示す．Nelements は，7680, 3840, 1920, および 960　とし，問題サイズにあわせてカーネルの呼び出し回数を 100, 200, 400, および 800 回とした．それぞれに対して富岳の異なるノードを用いて 3 回の独立な試行を行った．富岳の A64FX プロセッサの 1 ノードのみを用い，フラット MPI (48 MPI プロセス) および OpenMP+MPI のハイブリッド並列 (4 MPI プロセス，12 OpenMP　スレッド) を実行した．

LLVM や gcc などのさまざまなオープンソースコンパイラが A64FX プロセッサをサポートしており富岳でも利用可能となっているが，本稿では富士通コンパイラ `FCC (FCC) 4.6.1 20210812/tcsds-1.2.33` を用いた．とくに言及のない場合，コンパイラオプションは `-Kfast,openmp` である．

2 章で述べたように，A64FX プロセッサは 1.8/2.0/2.2GHz で動作する．本実験では，2.2GHz を用いた．

4.2 as-is コードの性能評価

表 3 および**表 4** の左カラムが”as-is” のコードの GFLOPS/s である．GFLOPS/s は浮動小数点演算数の理論値と経過時間から算出された．それぞれの結果は，100, 200, 400, および 800 回のカーネル呼び出しを 3 試行した結果の平均である．評価から，フラット MPI のほうが OpenMP+MPI ハイブリッドよりもやや性能が高い傾向があり，より大きい問題サイズのほうが小さい問題サイズよりも性能が高かった．しかし，全体的に見て，GFLOPS/s は理論ピーク性能の 4.0 から 4.3%であり，“as-is” コードの A64FX プロセッサ上での実行性能は決して高いもので



```
00 extern "C" void axhelm_bk_v0(const dlong & Nelements,
01                              const dfloat* __restrict__ ggeo,
02                              const dfloat* __restrict__ D,
03                              const dfloat* __restrict__ q,
04                              dfloat* __restrict__ Aq ){
05   dfloat s_q[p_Nq][p_Nq][p_Nq], s_Gqr[p_Nq][p_Nq][p_Nq], \
06   s_Gqs[p_Nq][p_Nq][p_Nq], s_Gqt[p_Nq][p_Nq][p_Nq], s_D[p_Nq][p_Nq];
07
08   for(int j = 0; j < p_Nq; ++j)
09     for(int i = 0; i < p_Nq; ++i)
10       s_D[j][i] = D[j * p_Nq + i];
11
12 #pragma omp parallel for private(s_q, s_Gqr, s_Gqs, s_Gqt)
13   for(dlong e = 0; e < Nelements; ++e) {
14     const dlong element = e;
15     for(int k = 0; k < p_Nq; k++)
16       for(int j = 0; j < p_Nq; ++j)
17         for(int i = 0; i < p_Nq; ++i) {
18         const dlong base = i+j*p_Nq+k*p_Nq*p_Nq+element*p_Np;
19         const dfloat qbase = q[base];
20           s_q[k][j][i] = qbase;
21         }
22
23     for(int k = 0; k < p_Nq; ++k)
24       for(int j = 0; j < p_Nq; ++j)
25         for(int i = 0; i < p_Nq; ++i) {
26           const dlong id = i+j*p_Nq+k*p_Nq*p_Nq+element*p_Np;
27           const dlong gbase = element*p_Nggeo*p_Np+k*p_Nq*p_Nq+j*p_Nq+i;
28           const dfloat r_G00 = ggeo[gbase + p_G00ID * p_Np]; // #define p_G00ID 1
29           const dfloat r_G01 = ggeo[gbase + p_G01ID * p_Np]; // #define p_G01ID 2
30           const dfloat r_G11 = ggeo[gbase + p_G11ID * p_Np]; // #define p_G11ID 4
31           const dfloat r_G12 = ggeo[gbase + p_G12ID * p_Np]; // #define p_G12ID 5
32           const dfloat r_G02 = ggeo[gbase + p_G02ID * p_Np]; // #define p_G02ID 3
33           const dfloat r_G22 = ggeo[gbase + p_G22ID * p_Np]; // #define p_G22ID 6
34
35           dfloat qr = 0.f, qs = 0.f, qt = 0.f;
36             for(int m = 0; m < p_Nq; m++) {
37               qr += s_D[i][m] * s_q[k][j][m];
38               qs += s_D[j][m] * s_q[k][m][i];
39               qt += s_D[k][m] * s_q[m][j][i];
40             }
41           s_Gqr[k][j][i] = r_G00 * qr + r_G01 * qs + r_G02 * qt;
42           s_Gqs[k][j][i] = r_G01 * qr + r_G11 * qs + r_G12 * qt;
43           s_Gqt[k][j][i] = r_G02 * qr + r_G12 * qs + r_G22 * qt;
44         }
45
46     for(int k = 0; k < p_Nq; k++)
47       for(int j = 0; j < p_Nq; ++j)
48         for(int i = 0; i < p_Nq; ++i) {
49           const dlong id = element*p_Np+k*p_Nq*p_Nq+j*p_Nq+i;
50           dfloat r_Aqr = 0, r_Aqs = 0, r_Aqt = 0;
51           for(int m = 0; m < p_Nq; m++) {
52               r_Aqr += s_D[m][i] * s_Gqr[k][j][m];
53               r_Aqs += s_D[m][j] * s_Gqs[k][m][i];
54               r_Aqt += s_D[m][k] * s_Gqt[m][j][i];
55             }
56           Aq[id] = r_Aqr + r_Aqs + r_Aqt;
57       }
58   }
59 }
```

図 3 評価対象となるカーネルのソースコード．実線のループは”as-is” に対するコンパイル SIMD 化されており，点線のコードはフルアンロールされている．パラメータ p_Nq は#define ディレクティブで定義され，4 から 14 の値を取る．本稿では 8 のケースを扱う．p_Np = p_Nq^3 であり，p_G00ID, p_G01ID, $\cdots$, p_G22ID はそれぞれ $1, 2, \cdots 6$ と事前に定義される．

```
00 for(int k = 0; k < p_Nq; ++k){
01   for(int j = 0; j < p_Nq; ++j){
02     for(int i = 0; i < p_Nq; ++i){
03       qr[i]= 0.f, qs[i]= 0.f, qt[i] = 0.f;
04       for(int m = 0; m < p_Nq; m++) {
05         qr[i] += s_D[i][m] * s_q[k][j][m];
06         qs[i] += s_D[j][m] * s_q[k][m][i];
07         qt[i] += s_D[k][m] * s_q[m][j][i];
08       }
09     }
10     for(int i = 0; i < p_Nq; ++i){
11       const dlong id = i+j*p_Nq+k*p_Nq*p_Nq+element*p_Np;
12       const dlong gbase = element*p_Nggeo*p_Np+k*p_Nq*p_Nq+j*p_Nq+i;
13       const dfloat r_G00 = ggeo[gbase + p_G00ID * p_Np];
14       const dfloat r_G01 = ggeo[gbase + p_G01ID * p_Np];
15       const dfloat r_G11 = ggeo[gbase + p_G11ID * p_Np];
16       const dfloat r_G12 = ggeo[gbase + p_G12ID * p_Np];
17       const dfloat r_G02 = ggeo[gbase + p_G02ID * p_Np];
18       const dfloat r_G22 = ggeo[gbase + p_G22ID * p_Np];
19
20        s_Gqr[k][j][i] = r_G00 * qr[i] + r_G01 * qs[i] + r_G02 * qt[i];
21        s_Gqs[k][j][i] = r_G01 * qr[i] + r_G11 * qs[i] + r_G12 * qt[i];
22        s_Gqt[k][j][i] = r_G02 * qr[i] + r_G12 * qs[i] + r_G22 * qt[i];
23     }
24 }}
```

loop-2a: lines 02–09
loop-2b: lines 10–23

図 4 tune1 コード．“as-is” における loop-2 を分割し，loop-2b を新たに SIMD 化した．実線部分が SIMD 化され，点線部分はフルアンロールされている．

はなかった．

4.3 SIMD 化

図 3 では，”as-is” コードに対してコンパイラが適用した最適化も示した．実線で囲まれた部分は SIMD 化されたコードを示す．図からほとんどの浮動小数点演算がコンパイラにより SIMD 化されている．これは最内ループの回転数 p_Nq が 8 回転であり，A64FX プロセッサの SIMD 幅 (512-bit，8-倍精度浮動小数点) と等しく，コンパイラにとって SIMD 化が容易であるからであると考えられる．p_Nq の数は# define ディレクティブにより与えられるため，コンパイラは事前にこの数を知ることができる．

SIMD 化をさらにすすめるために，**図 4** に示すように，loop-2 における変数 i のループを 2 つのループに分割した．loop-2 では，前半の変数 m のループのみが SIMD 化されていたが，図 4 に示すように，この分割により後半の変数 i のループも SIMD 化することができた．以降ではこのコードは，tune1 コードと呼ぶ．

4.4 ソフトウエアパイプライニング

ソフトウエアパイプライニング (SWP) は，ループにおける複数の繰り返しをオーバラップする最適化手法であり，i 回目の繰り返しの終了をまたずに，$i+1$ 回目の繰り返しが開始される．このオーバラップにより，ある命令が依存関係のある前の命令の終了をまっている間に，次の繰り返しにある独立な命令を発行することができ，命令レイテンシを隠蔽することが可能になる．

図 5 に，A64FX プロセッサにおけるソフトウエアパイプライニングの例を示す．これらの例は，ベクトルの足し算 $\boldsymbol{z} = \boldsymbol{x} + \boldsymbol{y}$ を行う図の左上のソースコードを実際に富士通コンパイラで SWP なし・ありで変換し，得られたアセンブラリストからループ部分を抜き出し，可読性を高めるために浮動小数点演算とロード・ストア以外の命令を割愛したものである．右のアセンブリリストが SWP を適用しない場合，左のアセンブリリストは SWP を適用した場合である．左のリストにおいては，SIMD 幅と等しい 8 回の繰り返しに含まれる一連の命令 `ld1d`，`ld1d`, `fadd`, および `st1d` は同じ色でマークされている．SWP を適用しない場合，`fadd` 命令は $\boldsymbol{x}$ および $\boldsymbol{y}$ に対する 2 つのロード命令の完了を待つ必要がある．一方，SWP を適用した場合，これらの命令の間に他の繰り返しの命令を発行している．たとえば，図 5 の 15 行目の命令は，レジスタ `z3` および `z2` を使用しており，これらは図の 0 行および 1 行目でロードされている．これらの `fadd` および `ld1d` 命令の間には別の 13 の命令が発行されているため[*2]，15 行目の `fadd z18.d z2.d z3.d` が呼ばれたときには，必要なレジスタ `z2` および `z3` は準備ができている．

A64FX プロセッサの SVE 命令のレイテンシは比較的長く，一般的な浮動小数点演算，`fadd` (足し算)，`fmul` (掛け算)，`fmad` (積和演算) などは 9 サイクルを要する [3]．さらに，複数の μ オペレーションで構成された命令のレイテンシは 9 より大きくなる．コアにおけるアウトオブオーダ (out-of-order, OOO) もレイテンシを隠蔽する手段であるが，A64FX プロセッサの OOO 資源は省電力化のためにそれほど豊富ではない [3]．そのため，SWP によりレイテンシを隠蔽することが重要である．

SWP においては，オーバラップされるループの繰り返しの回数が大きくなるほど，レジスタプレッシャも大きくなる．幸いにも，本稿であつかうカーネルの最内ループはそれほど複雑ではなく，それほど大量のレジスタを必要としない．SWP を機能させるためのもうひとつの重要な要素は，ループの繰り返し回数である．ループの繰り返し回数が少ないとき，オーバラップさせる十分な命令が得られず，SWP の効果は低いものになる．また，効果が低いと判断された場合，コンパイラは SWP を適用しない．

図 3 に示すように，as-is コードにおいては loop-1 のすべてのループは SIMD 化あるいは完全にアンロールされており，SWP 化の余地はない．loop-2 では，変数 j のループは no schedule is obtained [*3] により SWP 化が適用されなかった．loop-3 では，変数 k のループが SWP 化の対象となるが，このループは完全にアンロールされた i および j ループを含み，結果的に命令数が多くなったことからレジスタプレッシャが大きく，SWP 化が適用されなかった．

SWP 化を促進するために，**図 6** に示すように，各 j と k のループを collapse して 1 つの jk ループにまとめた．その結果，loop-1 および loop3 の jk ループには SWP が適用された．loop-2 の jk ループは no schedule is obtained と

[*2] 2 つのロード命令，あるいはストアと演算命令は同時に発行可能なため，この間のサイクル数は 13 より短いことに注意されたい
[*3] これはコンパイラによる最適化メッセージである

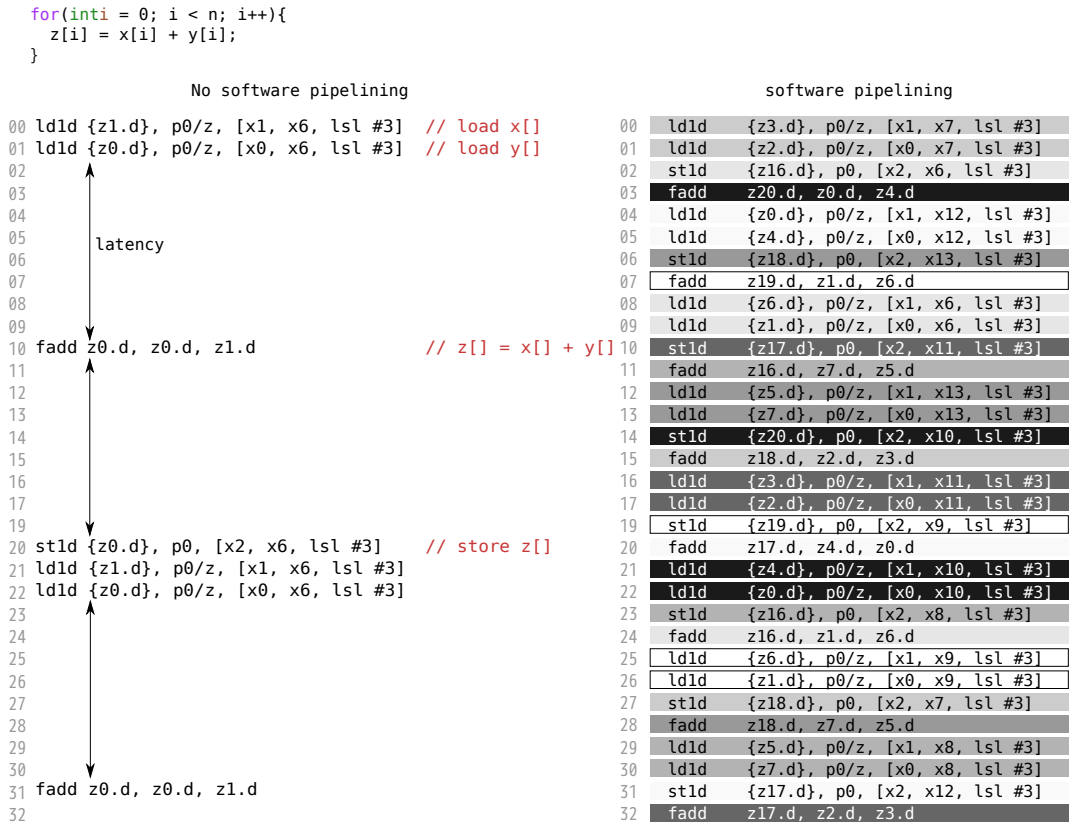


図 5 A64FX 向けアセンブラリストにおけるソフトウエアパイプライニングの例．右上のベクトル和に対して，左がソフトウエアパイプライニングなし，右がソフトウエアパイプライニングを適用した例．右のリストにおいては，同じ繰り返しにおける命令は同じ色でマークされている．

```
for(int k = 0; k < p_Nq; ++k){
  for(int j = 0; j < p_Nq; ++j){
    ⬇
for(int jk = 0; jk < p_Nq*p_Nq; ++jk){
int j = jk%p_Nq; int k = jk/p_Nq;
```

図 6 tune2．変数 j, k のループが collapse された．

され，SWP が適用されなかったが，SWP を短いループ向けに調整するコンパイラオプション`-Kswp_weak` を追加したところ，このループにも SWP を適用することができた．なお，“as-is” コードと，tune1 コードにも同じオプションを適用したが，これらのコードに対しては SWP は適用されなかった．

以降ではこのコードを tune2 と呼ぶ．

4.5 メモリアクセスパターンの改善

A64FX プロセッサはギャザーロードとスキャッターストアをサポートしている．ギャザーロードは，即値あるいはスカラ変数とベクトルにより定義されたメモリアドレスのリストを 1 つのベクトルレジスタにロードする．スキャッターストアは，即値あるいはスカラ変数とベクトルにより定義されたメモリアドレスのリストに 1 つのベクトルレジスタの各要素をストアする [4]．

対象のカーネルでギャザーロードが現れる例としては，図 3 の 38 行目の `s_q[k][m][i]` に対するロード命令が挙げられる．この命令を含むループのインデックスは変数 m であるため，配列 `s_q` に対するメモリアクセスは，ストライド幅 8 のストライドアクセスとなり，以下のように変換される：

```
index   z0.d, 0, 8
ld1d    {z3.d}, p0/z, [x10, z0.d, lsl #3]
```

ここで，`index` 命令は，第 2 オペランドの数値でスタートし，第 3 オペランドで指定された値づつ増加する数値のリストを生成し，第 1 オペランドで指定されるレジスタに代入する．この例では，`z0.d` に {0,8,16,..,56} が代入される．`ld1d` 命令は，第 3 オペランドの 3 つの要素を用いてロードされる要素のアドレスを計算し，それらのロードして，第 1 オペランドのレジスタに代入する．連続ロードもギャザーロードも同じように 1 命令で定義されるが，連続ロードは 1 つのアドレス計算パイプラインと 1 つのフェッチポートのみを用いるのに対して，ギャザーロードは 2 つのアドレス計算パイプラインと 4 つのフェッチポートを用いる．また，要素数のリストからアドレスを計算する際には，SVE の浮動小数点演算ユニットを使用する．ゆえに，連続アクセスの SIMD のロードのレイテンシが 11

```
00 for(int jk = 0; jk < p_Nq; ++jk){
01   int j = jk%p_Nq; int k = jk/p_Nq;
02   for(int i = 0; i < p_Nq; ++i){
03     qr[jk][i]= 0; qs[jk][i]= 0; qt[jk][i] = 0;
04   }
05   for(int m = 0; m < p_Nq; m++) {
06   dfloat dtmp0 = s_q[k][j][m];
07   dfloat dtmp1 = s_D[j][m];
08   dfloat dtmp2 = s_D[k][m];
09   for(inti = 0; i < p_Nq; i++){
10       qr[jk][i] +=    s_E[m][i] * dtmp0;  // s_E = s_D^T
11       qs[jk][i] += s_q[k][m][i] * dtmp1;
12       qt[jk][i] += s_q[m][j][i] * dtmp2;}
13 }}}
14 for(int jk = 0; jk < p_Nq; ++jk){
15   int j = nk%p_Nq; int k = jk/p_Nq;
16   for(int i = 0; i < p_Nq; ++i){
17     const dlong id = i+j*p_Nq+k*p_Nq*p_Nq+element*p_Np;
18     ...
19      s_Gqr[k][j][i] = r_G00 * qr[jk][i] + r_G01 * qs[jk][i] + r_G02 * qt[jk][i];
20      s_Gqs[k][i][j] = r_G01 * qr[jk][i] + r_G11 * qs[jk][i] + r_G12 * qt[jk][i];
21      s_Gqt[j][i][k] = r_G02 * qr[jk][i] + r_G12 * qs[jk][i] + r_G22 * qt[jk][i];
22   }
23 }
24 for(int jk = 0; jk < p_Nq; jk++){
25   int j = jk%p_Nq; intk = jk/p_Nq;
26   for(int i = 0; i < p_Nq; ++i) {
27     const dlong id = element*p_Np+k*p_Nq*p_Nq+j*p_Nq+i;
28     dfloat r_Aqr = 0;
29     for(int m = 0; m < p_Nq; m++) {
30        r_Aqr += s_E[i][m] * s_Gqr[k][j][m];
31        r_Aqr += s_E[j][m] * s_Gqs[k][i][m];
32        r_Aqr += s_E[k][m] * s_Gqt[j][i][m];
33     }
34     Aq[id] = r_Aqr;
35   }
36 }
```

loop-2a, loop-2b, loop-3

図 7 tune3 コード．行列 s_D の転置行列である s_E が導入された．連続メモリアクセスを促進するために，ループ m と i の順序を入れ替えた (L5 および L9)．もともとストライドであった loop-2a における s_E, s_q へのアクセスはすべて連続とした．中間行列 s_Gqs と s_Gqt のインデックスの順序も，後のアクセスが連続となるように入れ替えた．

であるのに対して，ギャザーロードでは 15 もしくはそれ以上となっている．

図 7 はメモリアクセスパターンを改善し連続アクセスを促進するように最適化したコードである．新たに二次元配列 s_E が導入された．これは s_D の転置行列であり，最外のループ e の前に生成される．s_D へのアクセスがストライドアクセスであるときは，かわりに s_E が使用される．さらに，loop-2a の m と i のループの順序を変更した．これにより，loop-2a の s_D, s_E, および s_q へのアクセスはすべて連続ロードとなった．加えて，途中の結果を保存するために使用される中間配列の s_Gqs, s_Gqt のインデックスの順序を図 7 に示すように変更し，続く loop-3 におけるこれらの配列へのアクセスを連続ロードとした．

以降ではこのコードを tune3 と呼ぶ．

4.6 プリフェッチ

プリフェッチは事前にデータをメモリからキャッシュに移動することでメモリアクセスのレイテンシを隠蔽する最適化手法である．A64FX プロセッサにおいては，ソフトウエアプリフェッチおよびハードウエアプリフェッチがサポートされている．前者はプリフェッチ命令やディレクティブをソースコードに記述することで陽にプリフェッチ命令を記述する．後者はハードウエアのアドレス予想メカニズムにより自動的に行われる．

ソフトウエアプリフェッチのために，C/C++では，__builtin_prefetch 関数が用意されている．FORTRAN では，!OCL PREFETCH_READ/WRITE ディレクティブが使用可能である．

__builtin_prefetch 関数は以下で定義される：

```
void __builtin_prefetch
  (void *addr, int rw, int locality);
```

`*addr` にプリフェッチされるデータの先頭アドレスを，`rw` に read (0) もしくは write (1) を，そして `locality` にはプリフェッチ先として L1 (3) もしくは L2 (2) を指定する．A64FX プロセッサにおいては，各プリフェッチ命令は 1 キャッシュラインを先読みする．なお，1 キャッシュラインは 256byte なので，倍精度浮動小数点の場合は 32 要素となる．

図 8 に，本稿で追加したプリフェッチ命令を示す．図 8 の 31 から 39 行目に示すように，loop-2b の内部で $jk+8$ 回転先の配列 ggeo が L1 に先読みされる．また，loop-2a が開始される前に，loop-2b で使用されるすべての ggeo 配列が L2 に先読みされる．前述のようにプリフェッチ命令は 1 回で 32 要素を読むため，1 回の繰り返しで 8 要素しか用いられない L1 へのプリフェッチを毎回行うことは過剰

```
00    dlong base0= element * p_Nggeo * p_Np + p_G00ID * p_Np; // p_G00ID=1
01    __builtin_prefetch(&(ggeo[base0 + p_Np*0 + 32*0]), 0, 2);
02    __builtin_prefetch(&(ggeo[base0 + p_Np*0 + 32*1]), 0, 2);   } 16 prefetch
03        ...                                                      } instructions
04    __builtin_prefetch(&(ggeo[base0 + p_Np*0 + 32*15]), 0, 2);  }
05
06    __builtin_prefetch(&(ggeo[base0 + p_Np*1 + 32*0]), 0, 2);    } 16x6 prefetch
07        ...                                                      } instructions
08
09    __builtin_prefetch(&(ggeo[base0 + p_Np*2 + 32*0]), 0, 2);
10        ...
11        ...
12    __builtin_prefetch(&(ggeo[base0 + p_Np*5 + 32*15]), 0, 2);
13
14    for(int jk = 0; jk < p_Nq; ++jk){
15      int j = jk%p_Nq; int k = jk/p_Nq;
16      for(int i = 0; i < p_Nq; ++i){
17        qr[jk][i]= 0; qs[jk][i]= 0; qt[jk][i] = 0;
18       }
19      for(int m = 0; m < p_Nq; m++) {                  loop-2a
20      dfloat dtmp0 = s_q[k][j][m];
21      dfloat dtmp1 = s_D[j][m];
22      dfloat dtmp2 = s_D[k][m];
23      for(inti = 0; i < p_Nq; i++){
24          qr[jk][i] +=    s_E[m][i] * dtmp0;  // s_E = s_D^T
25          qs[jk][i] += s_q[k][m][i] * dtmp1;
26          qt[jk][i] += s_q[m][j][i] * dtmp2;}
27    }}}
28    #pragma loop swp_policy large
29    for(int jk = 0; jk < p_Nq; ++jk){
30      int j = nk%p_Nq; int k = jk/p_Nq;
31      dlong base0 = e*p_Nggeo*p_Np+ k * p_Nq * p_Nq + j * p_Nq + p_G00ID * p_Np + 64;
32              // p_G00ID = 1
33              // 64 = 8 * p_Nq (fetch jk+8-th iteration)
34      __builtin_prefetch(&(ggeo[base0 + p_Np*0]), 0, 3);
35      __builtin_prefetch(&(ggeo[base0 + p_Np*1]), 0, 3);
36      __builtin_prefetch(&(ggeo[base0 + p_Np*2]), 0, 3);
37      __builtin_prefetch(&(ggeo[base0 + p_Np*3]), 0, 3);
38      __builtin_prefetch(&(ggeo[base0 + p_Np*4]), 0, 3);
39      __builtin_prefetch(&(ggeo[base0 + p_Np*5]), 0, 3);
40      for(int i = 0; i < p_Nq; ++i){                   loop-2b
41        const dlong gbase = e*p_Nggeo*p_Np + k * p_Nq * p_Nq + j * p_Nq + i;
42        const dfloat r_G00 = ggeo[gbase + p_G00ID * p_Np];  // #define p_G00ID 1
43        const dfloat r_G01 = ggeo[gbase + p_G01ID * p_Np];  // #define p_G01ID 2
44        ...
45        s_Gqr[k][j][i] = r_G00 * qr[jk][i] + r_G01 * qs[jk][i] + r_G02 * qt[jk][i];
46        s_Gqs[k][i][j] = r_G01 * qr[jk][i] + r_G11 * qs[jk][i] + r_G12 * qt[jk][i];
47        s_Gqt[j][i][k] = r_G02 * qr[jk][i] + r_G12 * qs[jk][i] + r_G22 * qt[jk][i];
48      }
49    }
50
```

図 8 tune4 コード．配列 ggeo に対するプリフェッチ命令を挿入した．

である．しかし，ループ内で分岐 (if) を用いてプリフェッチの発行を制御する場合，分岐を含むループでは SWP が阻害され，その場合のペナルティのほうが大きいため，本稿ではプリフェッチ命令を冗長に発行している．また，プリフェッチ命令を追加したことで，ループ内の命令の数が変化し，それにより SWP 適用時のコンパイラのコード生成戦略が変化したため，これを陽に指定するディレクティブである`#pragma loop swp_policy large`を追加した．

以降ではこのコードを tune4 と呼ぶ．

4.7 実験結果

表 3 および表 4 に，“as-is” と　4.3–4.6 章で述べた tune1 から tune4 のコードの GFLOPS/s を示す．いずれの実験も 1 ノードのみが用いられた．ここで tune n は tune $n-1$ を元にして開発されたため，たとえば tune4 は tune1 から tune3 で使用されたすべての最適化手法を含むことに注意されたい．表 3 は 48 プロセスを用いたフラット MPI の結果であり，表 4 は 4 プロセス 12 スレッドを用いた OpenMP+MPI ハイブリッド並列の結果である．ハイブリッド並列では各 CMG に 1 プロセス 12 スレッドが割り当てられている．

SIMD 化により得られた性能の改善は，0.45 ～ 7.2 %と低かった．これは，図 3 でも示したように，“as-is” のコードでもほとんどの浮動小数点演算は SIMD 化されており，さらなる SIMD 化による性能改善の余地が少なかったことが原因であると考えられる．**表 5** に，プロファイラにより得られた各コードにおける有効な命令数，浮動小数点演算数，SIMD 化率，SVE 化率を示す．SVE 化率は SIMD 命令のうちの何パーセントが SVE 命令であるかを示す．SVE ではない SIMD 命令としては NEON 命令がある．これらの数値は，Nelement が 7680 のカーネルを 100 回呼び出した場合であり，全プロセスの平均値である．表 5 に示すように，“as-is” からループを分割した tune1 では，SIMD 化率および SVE 化率の両方が上昇している．また，いくつかのスカラ命令がベクトル命令としてまとめ上げられた結果，有効命令数も減少している．

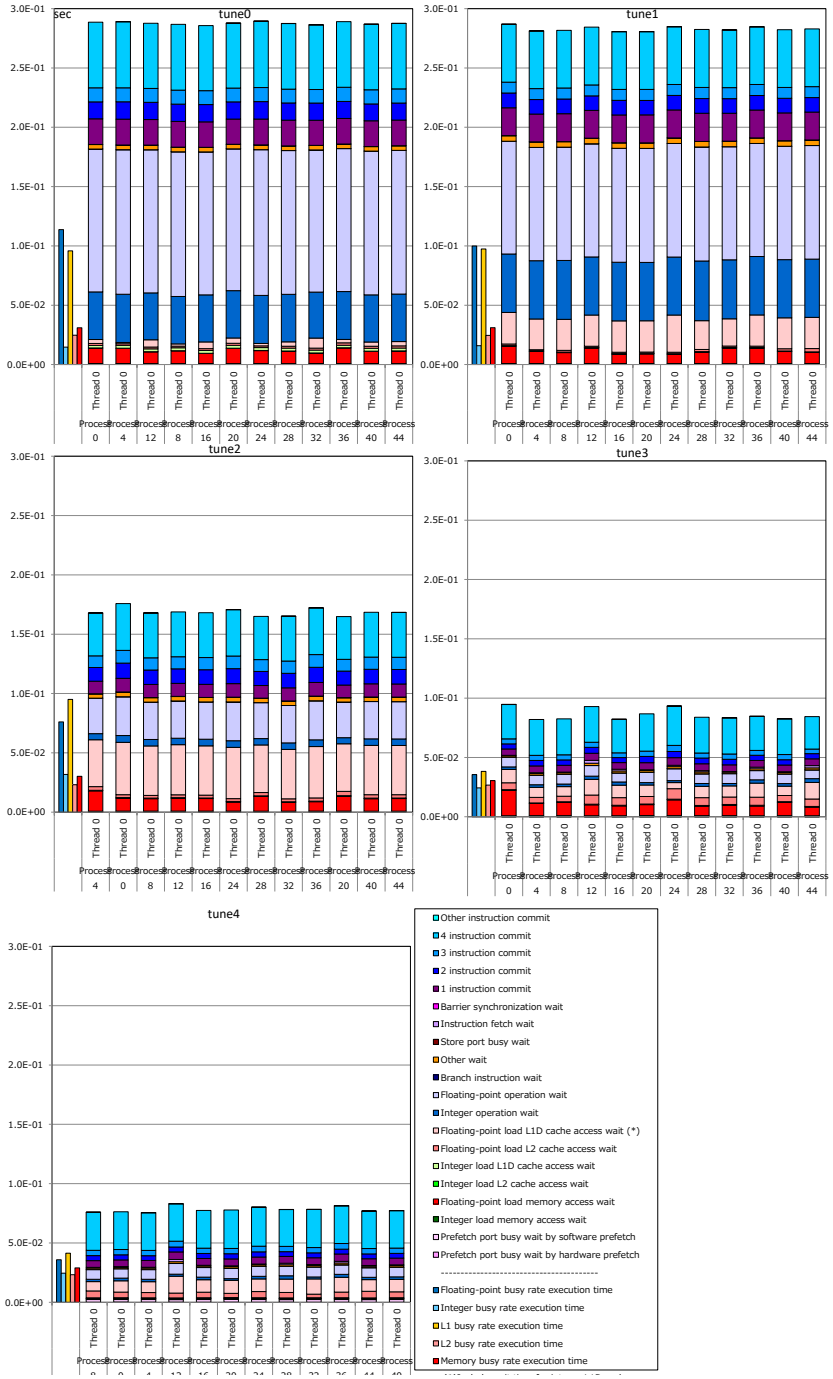


図 9 各コードに対するサイクルアカウント実行時間 (sec)．Nelements=7680，flatMPI のときの CMG-0 における各プロセスの結果である．

SWP の促進により，tune2 では，tune1 から 47–55%の性能の向上が得られた．**図 9** は，性能解析ツールにより得られた Nelements=7680, flat MPI に対する cycle accounting execution time である．図から tune2 では，SWP によってそれまで大きな割合を示してた `Floating-point operation wait` が大幅に減少しており，`Integer operation wait` も減少していることがわかる．

さらに表 5 から，tune2 では有効命令数や浮動小数点演算数も減少している．これは，ループの collapse によって 2 つあったループ変数 j,k が 1 つになり，これらのインデックスを用いた計算が削減されたためであると考えられるが，詳細な解析は今後の課題としたい．

メモリアクセスパターンを改善し，2 次元配列 3 次元配列に対するストライドアクセスを連続アクセスにした tune 3 でも，67–85 % という大きな性能の改善が見られた．図 9 によれば，tune2 でもっとも大きな時間を占めた `Floating-point L1D cache access wait` が大幅に減少している．表 5 によれば，有効命令数と浮動小数点演算数も減少している．これは，ロードされるリストの生成や SVE 演算ユニットを用いて行われるアドレスの計算といっ

表 3 48MPI プロセスの flatMPI のときのカーネルの性能 (GFLOPS/s)．結果は，100/200/400/800 回のカーネル呼び出しを 3 回独立に行った平均である．

Nele ments	tune0 as-is	tune1 SIMD	tune2 SWP	tune3 cont. access	tune4 prefetch
7680	143.449	146.648	228.673	397.603	535.046
3840	142.723	145.874	227.101	391.258	528.602
1920	141.351	144.691	223.982	383.852	508.101
960	139.183	141.791	218.960	378.087	470.633

表 4 4 MPI プロセス/12 OpenMPthread のハイブリッドのときのカーネルの性能 (GFLOPS/s)．結果は，100/200/400/800 回のカーネル呼び出しを 3 回独立に行った平均である．

Nele ments	tune0 as-is	tune1 SIMD	tune2 SWP	tune3 cont. access	tune4 prefetch
7680	141.073	144.209	219.052	385.953	547.749
3840	139.515	142.407	215.083	375.339	521.830
1920	136.454	139.259	208.541	348.598	476.651
960	131.399	133.595	191.120	318.500	429.264

表 5 Nelements=7680, flatMPI のときの 100 回のカーネル呼び出しに対する CPU 性能解析ツールの結果．48 プロセスの平均値を示した．SIMD 率は命令全体に対する SIMD 化%を，SVE 率は SIMD 命令に対する SVE 命令%を示す．

code	Effective instructions	Floating-point operations	SIMD rate (%)	SVE rate (%)
as-is	675,158,798	1,712,128,002	58.59	91.87
tune1	594,760,559	1,712,128,002	70.25	99.04
tune2	478,056,281	1,433,600,002	67.90	100.0
tune3	328,024,601	1,015,808,002	59.06	100.0
tune4	341,413,843	1,015,808,002	57.31	100.0

たギャザーロードにともなう演算が削減された結果であると考えられる．

tune3 ではメモリアクセスパターンを改善することで，キャッシュアクセス待ち時間を減らすことができたが，メモリアクセス待ち時間が残っていた．図 9 で示すように，プリフェッチ命令を追加した tune4 では，L1, L2, メモリアクセス待ち時間を減少することができた．tune3 から tune4 での性能改善は，19–27%だった．**表 6** は，Nelements=7680, flat MPI の場合の，各コードの L1D および L2 のキャッシュミス率と，そのうちのデマンドミス率を示す．キャッシュミスは，デマンド，ハードウエアプリフェッチ，ソフトウエアプリフェッチに対するキャッシュミスの合計であるが，プリフェッチ時のキャッシュミスは即座に性能低下につながらない一方でデマンドに対するミスは性能に対する影響が大きい．表 6 で示されるように，プリフェッチ命令を追加した tune4 コードでは L2 デマンドミス率を 16–20 %から 2.74%まで減少させている．L1 でのデマンドミス率も 32–39 %から 24.8 % に減少している．表 5 より，tune3 から tune4 への最適化では浮動小数点演算数は一定であるものの，プリフェッチ命令が追加されたため有効命令数は増加している．

5. おわりに

本稿では，計算流体力学コードである Nek5000/RS から抜粋されたカーネル axheml を用いて，SIMD 化，ソフトウエアパイプライニング，連続メモリアクセス，ソフトウエアプリフェッチを促進する最適化技術が，A64FX での axheml カーネルの実行性能に与える影響を調査した．“as-is” のコードの段階でほとんんどの浮動小数点演算が SIMD 化されたため，さらなる SIMD 化による性能向上は小さかった．しかし，512-bit の SIMD 演算器を持つ A64FX プロセッサにおいて，SIMD 化が重要であることは言うまでもない．ソフトウエアパイプライニングも比較的大きな命令レイテンシを持つ A64FX プロセッサ最適化に重要な技術である．“as-is” のコードをそのままコンパイルした場合，SWP は適用されなかったが，8 回転のショートループ 2 つを collapse して 64 回転のループとすることで，SWP の適用を促進することができた．性能解析ツールから得られた cycle account execution time によれば，SWP によって浮動小数点および整数の演算待ち時間が大きく減少した．また，ループの順序や多次元配列のインデックスの順序を入れ替えることで，ストライドアクセスの多かった”as-is” コードの連続アクセスを促進した．これにより，L1D キャッシュアクセス待ち時間および浮動小数点演算待ち時間が減少した．幾何学的係数の配列 ggeo に対するプリフェッチ命令を追加し，L2 デマンドミス率大幅に削減したほか，浮動小数点メモリアクセス待ち時間をほぼゼロとした．以上を総合し，全体としては”as-is” のコードに対して 3.3 から 3.9 倍の高速化を得た．

本稿では，もっとも簡単なケースである多項式次数が 7 の場合，すなわち多重ループのループ回転数が 8 の場合を扱った．今後の課題としては，ショートループの回転数が A64FX プロセッサの SIMD 幅に一致しないより複雑な場合の最適化が挙げられる．

謝辞

本研究成果（の一部）は，理化学研究所のスーパーコンピュータ 「富岳」を利用して得られたものです．

参考文献

[1] Nek5000/nekbench. https://github.com/Nek5000/nekBench.
[2] P. Fischer, S. Kerkemeier, M. Min, Y.-H. Lan, M. Phillips, T. Rathnayake, E. Merzari, A. Tomboulides, A. Karakus, N. Chalmers, and T. Warburton. NekRS, a GPU-accelerated spectral element navier-stokes solver, 2021.
[3] Fujitsu Limited. A64FX microarchtecture manual .

表 6 Nelements=7680，flatMPI のときのキャッシュミスに対する解析．100 回のカーネル呼び出しの結果であり，CMG-0 上の 12 プロセスの平均．

code	L1D miss rate / load store inst.(%)	L1D miss demand rate /L1D miss (%)	L2 miss rate rate/load store inst. (%)	L2 miss demand demand rate /L2 miss (%)
as-is	1.67	32.55	1.62	16.00
tune1	2.03	34.02	2.00	20.14
tune2	2.25	32.72	2.23	18.04
tune3	2.84	39.30	2.42	18.09
tune4	2.76	24.82	2.30	2.74

https://github.com/fujitsu/A64FX/blob/master/doc/A64FX_Microarchitecture_Manual_en_1.6.pdf.

[4] A. Limited. Arm a64 instruction set architecture (version 2021-06). https://developer.arm.com/documentation/ddi0596/2021-06.

[5] D. S. Medina, A. St-Cyr, and T. Warburton. OCCA: A unified approach to multi-threading languages, 2014.

[6] R. Okazaki, T. Tabata, S. Sakashita, K. Kitamura, N. Takagi, H. Sakata, T. Ishibashi, T. Nakamura, and Y. Ajima. Supercomputer fugaku CPU A64FX realizing high performance, high-density packaging, and low power consumption. Technical Report Fujitsu technical review, 2020.

[7] M. Sato, Y. Ishikawa, H. Tomita, Y. Kodama, T. Odajima, M. Tsuji, H. Yashiro, M. Aoki, N. Shida, I. Miyoshi, K. Hirai, A. Furuya, A. Asato, K. Morita, and T. Shimizu. Co-design for A64FX manycore processor and Fugaku. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC20)*, 2020.

[8] L. UCHICAGO ARGONNE. Nek5000. http://nek5000.github.io/.