情報処理学会論文誌　コンピューティングシステム　Vol.6　No.3　43–57 (Sep. 2013)

スーパーコンピュータ「京」における格子QCDの単体性能チューニング

寺井 優晃[1,a)]　石川 健一[1,2]　杉崎 由典[3]　南 一生[1]
庄司 文由[1]　中村 宜文[1]　藏増 嘉伸[1,4]　横川 三津夫[1,5]

受付日 2012年12月21日, 採録日 2013年4月10日

概要：格子量子色力学（格子 QCD）は，時空間を 4 次元の立方格子として離散化し，格子点にクォークを，格子点間を結ぶリンクにグルオンを配置し，そのダイナミクスを求めることでクォークとグルオン間に働く強い力の相互作用を数値的に解く計算手法である．ダイナミクスを求める過程で，Wilson-Dirac 演算子の逆行列の計算が行われる．この演算子は，複素要素を持つ大規模疎行列となるため，逆行列計算は格子 QCD で最も計算時間を要する．今回チューニングを行った格子 QCD コードである LDDHMC は，領域分割された HMC アルゴリズムに基づく手法（DD-HMC）を採用している．特徴としては，倍精度 BiCGStab 法の前処理として，単精度の領域分割シュワルツ交代法（SAP）を適用した BiCGStab 法を使うことでほとんどの計算を単精度で行いつつ倍精度の解を求めることにある．さらに SAP の小領域に制限された行列の逆を求めるところに SSOR 法を用い SAP の収束を改善している．「京」の単体性能向上のため，SSOR 法の部分から 3 つのカーネルを抽出し，詳細プロファイラ機能を用いたボトルネック解析を実施した．その結果，オリジナルコードでは，a) SIMD 命令率，b) 整数ロードキャッシュアクセス待ち，c) 浮動小数点ロードキャッシュアクセス待ち，d) 命令スケジューリング，e) バリア同期待ちに問題点があることが明らかになった．これらの問題点についてチューニングを実施した結果，カーネル 1 で 1 コアあたり 23.2%から 38.1%，カーネル 2 で 24.3%から 38.0%，カーネル 3 で 23.6%から 44.9%に実効効率が改善された．1 チップあたりでは，カーネル 1 で 29.5%，カーネル 2 で 30.9%，カーネル 3 で 37.8%の改善が得られた．コンパイラの改良において，カーネルを用いたプロファイル情報の解析やチューニング手法が有効であることを示した．

キーワード：スーパーコンピュータ「京」，格子 QCD，性能チューニング，SPARC64^{TM} VIIIfx，コンパイラ

Performance Tuning of a Lattice QCD Code on a Node of the K computer

Masaaki Terai[1,a)]　Ken-ichi Ishikawa[1,2]　Yoshinori Sugisaki[3]　Kazuo Minami[1]
Fumiyoshi Shoji[1]　Yoshifumi Nakamura[1]　Yoshinobu Kuramashi[1,4]　Mitsuo Yokokawa[1,5]

Received: December 21, 2012, Accepted: April 10, 2013

Abstract: Lattice QCD is first principle calculation to solve the dynamics between quarks and gluons based on strong interaction. The calculation is performed on four dimensional space-time which is discretized to lattice, and requires a huge amount of inversion of the sparse matrix derived from Wilson-Dirac equation. In this study, Lattice QCD code, LDDHMC uses domain decomposition HMC algorithm with mixed precision BiCGStab solver for the linear equation. This scheme is nested, consists of inner solver and outer solver. The outer solver is calculation of BiCGStab with double precision. The inner solver is preconditioning calculation of BiCGStab with single precision and is preconditioned by the Lüscher's SAP. Furthermore, the calculation for the small block of SAP is improved with SSOR. To improve the performance we extracted three kernel codes from the SSOR routine in the application codes, and analyzed bottlenecks for the kernels by profiler. Based on the profiling we obtained the problems for following points: a) SIMD instruction rate, b) integer L1D cache misses, c) floating-point L1D cache misses, d) instruction scheduling, e) barrier synchronization. As a result, the tuning improves the peak performance a core from 23.2% to 38.1% in the kernel-1, from 24.3% to 38.0% in the kernel-2, from 23.6% to 44.9% in the kernel-3. The peak performance a chip is 29.5% in the kernel-1, 30.9% in the kernel-2, 37.8% in the kernel-3. The results show effectiveness for improvement of the compiler by profiling and tuning.

Keywords: K computer, lattice QCD, performance tuning, SPARC64^{TM} VIIIfx, compiler

1. はじめに

格子量子色力学（格子 QCD）は，原子核を構成する陽子，中性子，さらにはそれらを構成する素粒子であるクォークおよびグルオンの性質を記述することができる第一原理に基づく計算手法である．格子 QCD は，量子力学に基づくダイナミクスを求めるためにファインマンの経路積分を利用する．時空間を 4 次元の立方格子として離散化し，格子点にクォークを，格子点間を結ぶリンクにグルオンを配置し，そのダイナミクスを求めることでクォークとグルオン間に働く強い力の相互作用を数値的に解くことができる．しかし，離散化された時空間の経路積分は計算量がきわめて大きいため，Hybrid Monte Carlo（HMC）アルゴリズム [1] を用いることで確率論的に積分を行う．この HMC の計算において，クォークの伝播を表す Wilson-Dirac 演算子の逆行列の計算が行われる．この計算に用いられる演算子は，複素要素を持つ大規模で非対称な疎行列となるため，逆行列計算部分は格子 QCD で最も計算時間を必要とする [2], [3].

「京」開発プロジェクト [4], [5] では，格子 QCD の実装系である LDDHMC [6] について単体性能から高並列化に至る系統的なチューニングを実施した．LDDHMC は，領域分割された HMC アルゴリズムに基づく手法（DD-HMC）[7] を採用している．特徴としては，倍精度 BiCGStab 法の前処理として，単精度の領域分割シュワルツ交代法（SAP）を適用した BiCGStab 法を使うことでほとんどの計算を単精度で行いつつ倍精度の解を求めることにある．さらに SAP の小領域に制限された行列の逆を求めるところに SSOR 法を用いることで SAP の収束を改善している．

一般的にチューニングの初期段階では，アプリケーション・コードの複雑な処理内容を単純化するために，単体プロセッサで実行できる主要計算部を含んだコード（カーネル）の抽出を行う．抽出したカーネルに対して，「京」が提供するハードウェアモニタ（プロファイラ情報）を用いてボトルネックを推察し，チューニングをカーネルに適用し効果の検証を行う．LDDHMC については，単体性能向上のため，SSOR 法の部分から 3 つのカーネルを抽出し，詳細プロファイラ機能を用いたボトルネック解析を実施した．本稿では，ボトルネック解析によって明らかになった問題点およびそれらのチューニング手法について述べる．なお，明らかになった問題点を「京」のコンパイラの改良に資することを研究の目的としている．

[1] 独立行政法人理化学研究所計算科学研究機構
RIKEN Advanced Institute for Computational Science, Kobe, Hyogo 650–0047, Japan
[2] 広島大学大学院理学研究科
Graduate School of Science, Hiroshima University, Higashihiroshima, Hiroshima 739–8526, Japan
[3] 富士通株式会社
Fujitsu Ltd., Numazu, Shizuoka 410–0396, Japan
[4] 筑波大学数理物質系
Faculty of Pure and Applied Sciences, University of Tsukuba, Tsukuba, Ibaraki 305–8571, Japan
[5] 神戸大学大学院システム情報学研究科
Graduate School of System Informatics, Kobe University, Kobe, Hyogo 657–8501, Japan
[a] teraim@riken.jp

2. カーネルコード

2.1 概要

LDDHMC が扱う full QCD シミュレーションにとって，Wilson-Dirac オペレータ D の逆行列を求める部分が計算の大部分を占めるため，最適化は大規模疎行列ソルバの高速化に帰着する．オペレータ D は次のように定義される．

$$D(n,m) = \delta^{a,b}\delta_{\alpha,\beta}\delta_{n,m} - \kappa \mathrm{Mult}(n,m)^{a,b}_{\alpha,\beta} \tag{1}$$

ここで，$n = (n_x, n_y, n_z, n_t)$ は 4 次元格子上の格子点を表す．α，β はスピノル成分（4 成分），a，b はカラー成分（3 成分），κ はクォーク質量に関するパラメータ，δ はクロネッカーのデルタである．

式 (1) における Mult 演算が計算主要部に対応する．Mult 演算は式 (2) で与えられる．

$$\begin{aligned}\mathrm{Mult}(n,m)^{a,b}_{\alpha,\beta} = \sum_{\mu=1}^{4}\Big[&(1-\gamma_\mu)_{\alpha,\beta}\big(U_\mu(n)\big)^{a,b}\delta_{n+\hat{\mu},m} \\ &+(1+\gamma_\mu)_{\alpha,\beta}\big(U^\dagger_\mu(n-\hat{\mu})\big)^{a,b}\delta_{n-\hat{\mu},m}\Big]\end{aligned} \tag{2}$$

ここで，μ は時空間の軸成分（$\mu = 1$ は x 成分，$\mu = 2$ は y 成分，$\mu = 3$ は z 成分，$\mu = 4$ は t）成分，γ_μ は，4×4 の複素数スピノル行列，$U_\mu(n)$ はゲージ場を表す 3×3 の複素行列である．$\hat{\mu}$ は μ 軸方向への単位ベクトルを表す．

オペレータ D は大規模な疎行列であるため，D^{-1} の解法は反復法である BiCGStab 法を用いる．

大規模疎行列を領域分割するために，格子点を市松模様の小領域に分割し，偶奇性を交互に与えると，D は式 (3) のようにブロック化される．

$$D = \begin{pmatrix} D_{EE} & D_{EO} \\ D_{OE} & D_{OO} \end{pmatrix} \tag{3}$$

収束を改善するために，式 (3) を解くための前処理として，シュワルツ交代法を用いる．前処理は精度を要求しないため，単精度で解く．線形方程式は式 (5) および式 (6) に基づき式 (4) のように変更される．

$$D \cdot x = b \Longrightarrow (D \cdot M_{SAP})z = b, \quad x = M_{SAP} \cdot z \tag{4}$$

$$M_{SAP} \equiv K \sum\nolimits_{j=0}^{N_{SAP}} (1 - D \cdot K)^j \tag{5}$$

$$K \equiv \begin{pmatrix} (D_{EE})^{-1} & 0 \\ -(D_{OO})^{-1}D_{OE} & (D_{OO})^{-1} \end{pmatrix} \tag{6}$$

ここで式 (6) におけるブロック要素の逆行列 $(D_{EE})^{-1}$，

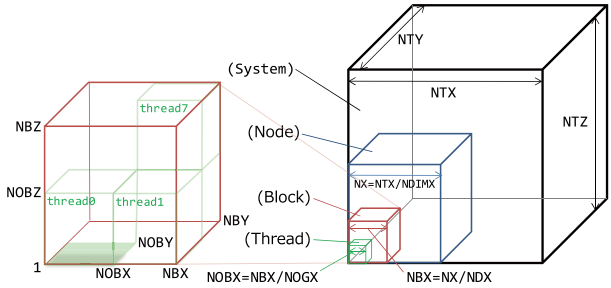


図 1　ノード分割からスレッド並列までの階層（4 次元目の T 軸は省略）

Fig. 1　Parallel decomposition in the temporal-spatial lattice (T-dimension is ignored).

$(D_{OO})^{-1}$ を解くことになる．本スキームでは，収束を早めるために，D_{EE} を上三角行列 U_{EE} と下三角行列 L_{EE} に分離し，SSOR 法を用いて固定反復している．

この SSOR 法におけるベクトルと分割領域内の前方ホッピング項の掛け算（L 行列）をカーネル 1，ベクトルと分割領域内の後方ホッピング項の掛け算（U 行列）をカーネル 2，ベクトルと分割領域内の係数行列の掛け算をカーネル 3 と呼ぶ．スレッド並列はこのカーネルの階層で実装されており，スレッド並列のためのオーダリングとしては並列性と収束性の観点から，スレッド内はデータ依存を残しつつも，スレッド間のデータ依存をなくし並列化するために Natural Block Ordering を採用している [6]．

2.2　実装

ノード分割からスレッド並列までの階層を**図 1** に示す．LDDHMC では，系を時空間 $XYZT$ についてノードで分割し，さらにブロックで分割したものに対してスレッド並列化を適用する．「京」のプロセッサ SPARC64$^{\text{TM}}$ VIIIfx [8], [9] は 1 チップ 8 コアであるので，1 ブロックは 8 スレッドで分割する．T 軸は，スレッド内では分割しない．ここで，$NT\{X,Y,Z,T\}$ は系の軸ごとの格子数，$NDIM\{X,Y,Z,T\}$ は系の軸ごとのノード数，$ND\{X,Y,Z,T\}$ は軸ごとのノードあたりのブロック数，$N\{X,Y,Z,T\}$ は軸ごとのノードあたりの格子数，$NB\{X,Y,Z,T\}$ は軸ごとのブロックあたりの格子数，$NOG\{X,Y,Z\}$ は軸ごとのスレッド数，$NOB\{X,Y,Z\}$ は軸ごとのスレッドあたりかつブロックあたりの格子数となる．

今回の計測では，格子サイズは $(NTX, NTY, NYZ, NTT) = (6,6,6,12)$ としたカーネルコードを用いて評価を行った．これは，SPARC64$^{\text{TM}}$ VIIIfx の L2 キャッシュに載る大きさである．

図 2 にカーネル 1 の擬似コードを示す．ここで，$NOBSITE$ はスレッド（$XYZT$ 軸すべて）あたりかつブロックあたりの格子数，yd，yt，u は配列変数である．配列変数 yd，yt，u の各軸は T 軸方向に直列化したスレッドごとに閉じた格子番号をインデックスに持つことで，連続

```
subroutine kernel1
!$OMP PARALLEL

do ibsite=NOBSITE-1,0,-1
  do iobx=NOBX,1,-1
    do ioby=NOBY,1,-1
      do iobz=NOBZ,1,-1
        do ibt=NBT,1,-1

          !$OMP BARRIER

          yt = (0.0e0,0.0e0)

(1)       if (ibt < NBT) then                    in  inout  in
            call s_mult_forw_t(yd,  yt,   u)
          endif

(2)       if (iobz < NOBZ) then
            call s_mult_forw_z(yd,  yt,   u)
          endif

(3)       if (iobz == 1) then
          if (ibz > 1) then
            call s_mult_back_z(yd,  yt,   u)
          endif
          endif

(4)       if (ioby < NOBY) then
            call s_mult_forw_y(yd,  yt,   u)
          endif

(5)       if (ioby == 1) then
          if (iby > 1) then
            call s_mult_back_y(yd,  yt,   u)
          endif
          endif

(6)       if (iobx < NOBX) then
            call s_mult_forw_x(yd,  yt,   u)
          endif

(7)       if (iobx == 1) then
          if (ibx > 1) then
            call s_mult_back_x(yd,  yt,   u)
          endif
          endif

          call s_mult_fclinv(yt)

          yd = y + kappa * yt

        end do
      end do
    end do
  end do
end do
end subroutine kernel1
```

図 2　カーネル 1 のコード概要

Fig. 2　An overview of the kernel-1.

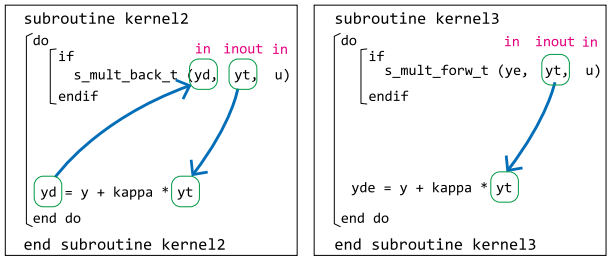


図 3　カーネル 2 および 3 のイテレーション間の依存

Fig. 3　Dependency in the kernel-2 and the kernel-3.

アクセスを可能にしている．擬似コード中で呼ばれているサブルーチン s_mult_forw_* または s_mult_back_* 中において，ゲージ場とクォーク場のベクトル行列積が行われる．

カーネル 1 および 2 では，配列変数 yd にイタレーション間のデータ依存がある．一方，**図 3** に示すカーネル 3 にはイテレーション間のデータ依存はない．

3.　単体性能チューニング

3.1　SPARC64$^{\text{TM}}$ VIIIfx プロセッサ概要

SPARC64$^{\text{TM}}$ VIIIfx プロセッサ [8], [9] は，1 チップあたり 8 コアを搭載し，各コアに 4 個の浮動小数点積和演算器と 1 ラインあたり 128 byte の 32 KB/2way の L1D キャッシュ，チップ上にはコア間で共有される 6 MB の L2 キャッ

シュを搭載したスーパースカラ型のマルチコア・プロセッサである．チップあたりの浮動小数点演算理論ピーク性能は 128 GFLOPS，理論メモリバンド幅は 64 GB/s である．SPARC V9 命令仕様 [10] を拡張しており，倍精度の浮動小数点レジスタが 256 本あり，またチップ内コア間で高速な同期を実現するハードウェアバリア機能，命令長 128 bit の SIMD 演算命令を提供する．命令実行のスループットはサイクルあたり最大 4 命令であり，SIMD 演算命令を用いることで，サイクルあたり 8 つの浮動小数点演算を実行することができる．これら SPARC V9 命令仕様をもとにした機能拡張を HPC-ACE と呼ぶ．

内積・行列積等で頻出する $d \leftarrow \pm(a \times b) \pm c$ のような連続した乗算と加減算に対して，SPARC V9 命令仕様は，4 個の浮動小数点レジスタを用いることで 1 つの命令として処理する浮動小数点積和演算命令（FMA 命令）を提供する．利点としては，内部的な丸め処理が代入の段階でのみ発生するため，丸め誤差が 1 回で済むという特徴がある．この FMA 命令には，符号反転と加減算の 4 通りの組合せがあり，それぞれに単精度と倍精度の命令が用意されている．さらに SPARC64^{TM} VIIIfx では，FMA 命令の SIMD 拡張（SIMD-FMA 命令）がされている．256 本の倍精度の浮動小数点レジスタを 128 本単位で分割し，前半分を basic 側，後半分を extended 側と呼び，SIMD 命令または SIMD-FMA 命令で意識して利用される．なお，FMA 命令の一般的な利用例としては，複素数での内積・行列積があげられる [11]．

3.2　チューニング概要

チューニングの手順としては，抽出したカーネルに対して，「京」で提供される詳細プロファイラを用いたボトルネック解析を実施した．本稿では，プロファイル情報から明らかになった問題点をチューニング対象として設定し，それに対して改善を試みた．その後，チューニングを行ったカーネルに対して再度プロファイル情報を取得し，当初の問題が解決されているか確認した．詳細プロファイラは SPARC64^{TM} VIIIfx が提供するハードウェアモニタを利用しており，その詳細はサイクルアカウンティング [8] として公開されている．この機能を用いることで命令ベースでのプロファイルが可能で，サンプリング方式で行う基本プロファイラとは異なり，より正確なコスト情報を得ることができる．1 つのボトルネックが解消した時点で，順番に次のチューニング対象を設定し，改善することで，単体性能の向上を図った．この過程において，a) SIMD 命令率，b) 整数ロードキャッシュアクセス待ち，c) 浮動小数点ロードキャッシュアクセス待ち，d) 命令スケジューリング，e) バリア同期待ちの 5 つのボトルネックがオリジナルコードで明らかになり，それぞれについてチューニングまたはコンパイラの改良を実施した．3.3 節以降で，その詳細を述べる．

```
-Kfast -Kopenmp
-Kprefetch_cache_level=1,prefetch_sequential=soft,prefetch_strong
```

図 4　コンパイラオプション
Fig. 4　The condition for compiler options.

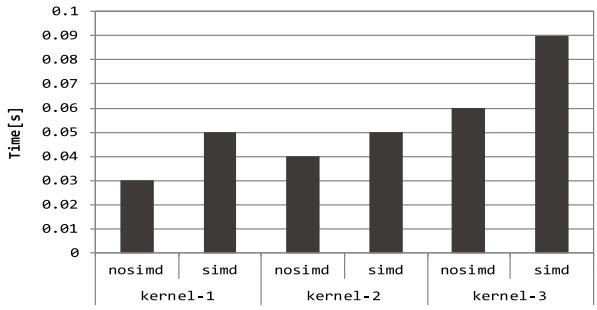


図 5　浮動小数点演算待ちの増加
Fig. 5　Results of waiting for floating-point operations.

コンパイラオプションを**図 4** に示す．なお，3.3 節から 3.6 節については 1 コアを，3.7 節については 1 チップ 8 コアを用いたチューニングを行った．

3.3　SIMD 命令率の改善

オリジナルの LDDHMC のカーネル部分は，インテル・アーキテクチャの SSE（Streaming SIMD Extensions）[12], [13] を用いることで単精度浮動小数点演算による 4 命令同時実行が実装されている．これを SPARC64^{TM} VIIIfx 上で効率的に動作させるには，SIMD 命令が適切に発行される必要がある．まずカーネル 1〜3 について，SIMD 命令率（= SIMD 命令数/有効総命令数）の評価を行った．なお SIMD 動作をしない場合との比較のために，コンパイラオプション -Knosimd を指定した場合についても示す．

プロファイルの結果，演算で占める部分にもかかわらず，すべてのカーネルについて SIMD 命令率が低い傾向が得られた．また，**図 5** に示すように，浮動小数点演算待ちが増加しており，パイプラインのストールが原因と推察される．これの改善については，命令スケジューリングの観点から 3.5 節で述べる．

まず SIMD 命令率が低い理由であるが，コンパイラが行う SIMD 化には，イタレーション間の演算に対して行うものと，同一イタレーション内の演算に対して行う 2 系統の SIMD が用意されている．その一方で，イタレーション内の分岐やイタレーション間のデータ依存があるため，SIMD 化が期待するような形で適用されていない可能性がある．同一イタレーションに対する SIMD も技術的には新しい試みで現在も開発中であるため，効果は確認できていない．一方，SIMD 拡張された FMA 命令を適用することでプロセッサが持つサイクルあたり最大 8 演算を実行する

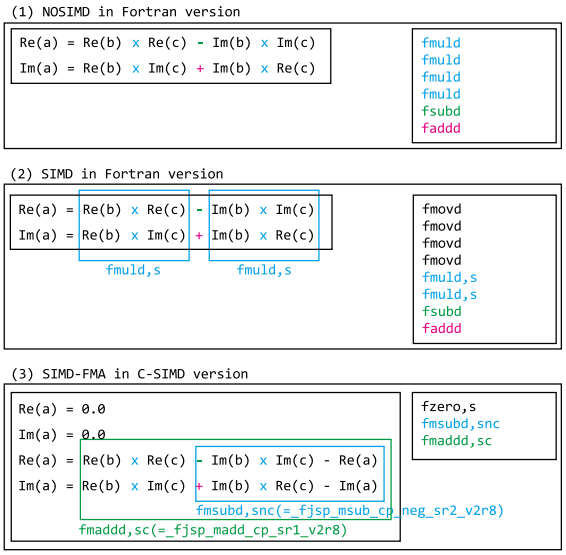


図 6　積和演算の組込み関数への書き換え例．(1) SIMD なし，(2) Fortran コンパイラによる SIMD あり，(3) 組込み関数による積和演算命令

Fig. 6　An example of tuning for multiply-add operations by C-SIMD intrinsic. (1) NOSIMD, (2) SIMD by Fortran compiler, (3) multiply-add operation by C-SIMD intrinsic.

ことができる．今回のケースのようにすでに SSE 拡張命令の組込み関数により実証的に SIMD 化が可能であることが判明している場合，アプリケーション依存になる問題点はあるが，コンパイラの最適化だけではなく，富士通 C コンパイラが提供する SIMD 組込み関数（以後，組込み関数）[14] を用いることで，命令セットを意識したチューニングが可能である．

SIMD 命令率が低い原因は，分岐に対するコンパイラの解析能力不足により，命令が生成されないことが大きいと考えるが，その一方で，当初，分岐を取り除いたとしたとしても SIMD 命令率が大きくは改善されないことを確認している．アセンブラコードを解析したところ，レジスタ間のデータコピーを行う fmovd 命令が大量に生成されていることが分かっており，もう 1 つの SIMD 命令率の低下の原因であると推察する．

まずはじめに，複素数の内積を行うアセンブラコードから浮動小数点演算とそれに関連した命令について**図 6** に整理した．SIMD 拡張および FMA 命令を用いない場合は，コンパイラは図 6-(1) に示すように fmuld 命令を 4 回，faddd 命令を 1 回，fsubd 命令を 1 回発行するコードを生成する．次に，SIMD を有効にした場合を図 6-(2) に示す．HPC-ACE では，SIMD 演算で用いられる basic 側と extended 側レジスタの組合せは固定されており，大部分の SIMD-FMA 命令のオペランドには basic 側レジスタのみ指定する仕様になっている．よって，ナイーブなコード生成を行うと，SIMD 命令の直前でレジスタ間のデータコピーのための fmovd 命令が必要となる．その一方で fmaddd,

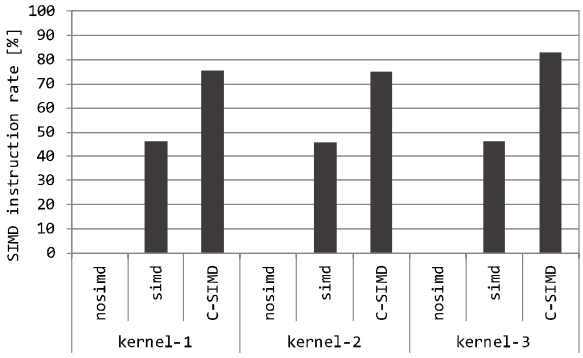


図 7　SIMD 命令率（/有効総命令数）の改善

Fig. 7　Results of SIMD instruction rate.

sc 命令と fmsubd，snc 命令のみ制限付きでオペランドに basic 側と extended 側レジスタの両方を指定することができる．組込み関数では，これらの命令をアプリケーション側から制御することができるため，積和演算に現れるすべての項と basic-extended 側レジスタを対応付けることができ，結果，fmovd 命令を抑えることができる．

以上，SIMD-FMA 命令が明示的に生成されるようにカーネルの書き換えを行った．図 6-(3) に SIMD-FMA 命令に書き換え後のコード例を示す．これを C-SIMD 版カーネルと呼ぶ．LDDHMC は Fortran で書かれているアプリケーションであるが，C と Fortran の言語間結合には Fortran 2003 が提供する機能を用いた．

図 7 に結果を示す．カーネルの書き換えにより，SIMD 命令率が向上していることが分かる．それにともない，浮動小数点演算ピーク比は，カーネル 1 で 23.2%が 24.6%，カーネル 2 で 24.3%から 25.1%，カーネル 3 で 23.6%から 29.4%に改善された．実行時間もカーネル 1 で，0.47 s から 0.44 s，0.45 s から 0.43 s，0.70 s から 0.56 s に短縮された．図 6-(2) の Fortran コンパイラが生成するコードでは，fmovd 命令が生成されない場合でも命令数が 4 で浮動小数点演算数が 6 となる．一方，図 6-(3) の C-SIMD 版の場合は fzero 命令を含めて 3 命令となり，命令数は減少する．ただし，C-SIMD 版では fmovd 命令数が減ったにもかかわらず，SIMD-FMA 命令を用いているため，fmsubd，snc 命令の第 3 項目の加算分により浮動小数点演算数が 8 に増加する．結果，浮動小数点演算数は，NOSIMD および SIMD の約 1.33 倍に増えている．

以上，fmovd 命令による不要なレジスタ間コピーは本アプリケーション以外においても影響が大きい．現状のコンパイラは，今回の知見を活かし fmovd 命令を抑制する目的で，SIMD-FMA 命令が生成されるように改良されている．

3.4　整数ロードキャッシュアクセス待ちの改善

組込み関数を用いてカーネル書き換えした後に，再度プロファイルを取得した結果，整数ロードキャッシュアクセス待ちの増加傾向が確認された．このボトルネックは，整

original code in C-SIMD

```
// definition
//
typedef struct { float c[2]; } scmplx;
typedef struct { scmplx u[6]; } ssu3mat;
typedef struct { scmplx y[12]; } ssu3vec;

typedef struct {
  scmplx c[_CLSPH+1];
} s_cchlmatf;

// single to double conversion
//
_fjsp_v2r8 conv stod(const scmplx *c){
  return (_fjsp_stod_v2r8(*((_fjsp_v2r4 *)c)));
}

// double to single conversion
//
scmplx conv_dtos (const scmplx *c){
  _fjsp_v2r4 c = _fjsp_dtos_v2r4(r);
  return *(scmplx *)(&c);
}
```

図 8　ユーザ定義型の構造体定義（型変換あり）
Fig. 8　An overview of user-defined typedef struct.

tuned code in C-SIMD

```
// definition
//
typedef struct { _fjsp_v2r4 u[6]; } ssu3mat;
typedef struct { _fjsp_v2r4 y[12]; }
ssu3vec;

typedef struct {
  _fjsp_v2r4 c[_CLSPH+1];
} s_cchlmatf;

// single to double conversion
//
_fjsp_v2r8 conv stod(const _fjsp_v2r4 *c){
  return (_fjsp_stod_v2r8(*c));
}

// double to single conversion
//
_fjsp_v2r8 conv_dtos (const _fjsp_v2r8 *c){
  _fjsp_v2r4 c = _fjsp_dtos_v2r4(r);
  return *(&c);
}
```

図 9　組込み関数提供の構造体定義（チューニング後）
Fig. 9　An overview of intrinsic-defined typedef struct (tuned).

数型の変数をロード，ストアした際に L1D キャッシュミスが発生することが原因であり，浮動小数点演算が主なコードでは比較的限定しやすく，間接参照にともなうインデックス計算が原因であることが多い．しかし，今回はインデックス計算を行っていない．LDDHMC のオリジナルコードでは，格子 QCD のクォーク場やゲージ場を表現するために構造体が多用されている．また，3.3 節で述べたように SSE 命令セットを意識したデータ構造に加えて，メモリプレッシャを小さくするためにカーネル部分では倍精度演算を単精度で扱っているが，SPARC64^{TM} VIIIfx が提供する SIMD 機構は単精度であっても倍精度レジスタを利用するため，計算する際に型変換を行う必要がある．そのため，オリジナルのコードでは構造体要素に対するデータ処理が行われている．

構造体変数へのアクセス部分に着目し，アセンブラコードによる解析を行った結果，**図 8** に示すような組込み関数でサポートされた型（_fjsp_v2r4）から，ユーザ定義の構造体（scmplx）への代入処理において，整数ロード・ストア命令（lduw，stw）が生成されていることが明らかになった．これらの構造体はどちらも 1 組の単精度実数型変数からなり，ここでは複素数の実数部と虚数部の組として用いられている．

本来は，構造体間の代入はメンバ変数の型を意識した処理が必要となるが，現在のコンパイラではユーザ定義による構造体メンバ変数の型の解析処理時間を削減するために，変数の型によらず整数型命令でコピー処理していたことが判明した．その結果，整数ロードキャッシュアクセス待ちが生じていた．

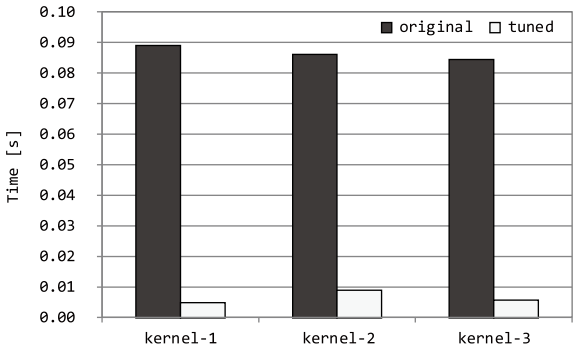


図 10　整数ロードキャッシュアクセス待ちの改善
Fig. 10　Results of waiting for integer L1D cache misses.

本稿での対策としては，**図 9** に示すようにオリジナルコード中で使用されているユーザ定義型を廃止し，処理系によってサポートされた型に書き換えを行った．結果，**図 10** に示すように整数ロード・ストア命令が生成されなくなり，整数ロードキャッシュアクセス待ちが解消した．非常に単純な対策だが，浮動小数点演算ピーク比がカーネル 1 で 24.0%から 32.7%，カーネル 2 で 25.7%から 33.9%，カーネル 3 で 29.5%から 35.8%に向上した．

3.5　浮動小数点ロードキャッシュアクセス待ちの改善

次に浮動小数点ロードキャッシュアクセス待ちについて検討する．このボトルネックは，浮動小数点数型の変数をロード・ストアした際に L1D キャッシュミスにより L2 キャッシュへのアクセス待ちが発生することが原因である．

ここで浮動小数点ロードキャッシュアクセス待ちとプリフェッチについて補足する．L1D キャッシュミスが発生した場合，L2 キャッシュからデータの読み込みを行う．これ

表 1　プリフェッチ生成による性能改善（カーネル 1）
Table 1　Results for software prefetch tuning (kernel-1).

	浮動小数点演算ピーク比 [%]	プリフェッチ命令数	L1D ミス dm 率 [%]	L1D ミス swpf 率 [%]
original	33.2	8.65E+06	46.9	18.7
noprefetch	32.7	2.03E+03	45.2	0.01
option-1	33.9	1.51E+07	18.4	51.5
option-2	34.0	1.51E+07	17.0	56.2

option-1: -Kprefetch_cache_level=1,prefetch_seq=soft

option-2: -Kprefetch_cache_level=1,prefetch_seq=soft,prefetch_strong

をリフィルと呼び，128 byte のライン単位でアクセスが行われる．L1D キャッシュはノンブロッキングキャッシュであるが，L1D キャッシュと L2 キャッシュではデータ読み出しのレイテンシに 6 倍程度の差があるので，L2 キャッシュから L1D キャッシュへのデータ転送待ちによってパイプラインがストールする場合がある．これを「浮動小数点ロードキャッシュアクセス待ち」と呼ぶ．これを防ぐためにあらかじめリフィルすることをプリフェッチと呼ぶ．SPARC64^{TM} VIIIfx は，ロード・ストア命令のキャッシュアクセス状況の監視を行い，連続アクセスであると判断した場合にプリフェッチを行うハードウェア機構（hwpf: hardware prefetch）を持つ．これとは別に命令によるプリフェッチ（swpf: software prefetch）を提供する．これはコンパイラが生成する．hwpf は自動的に行われるものであるが，連続アクセスを記憶するエントリ数に上限があり，また，キャッシュラインが連続でない場合 hwpf は実行されないため，swpf を用いることで性能向上が期待される．

プロファイラが用いるハードウェアモニタは，これらの L1D キャッシュミスのイベントについて，hwpf によるものを「L1D ミス hwpf 数」として，プリフェッチ命令によるものを「L1D ミス swpf 数」として，ロード・ストア命令によるものを「L1D ミス dm（デマンド）数」としてカウントする．一般的なチューニングとしては，L1D ミス swpf 数を大きくすることで，L1D ミス dm 数を小さくする方法をとることが多い．

表 1 にカーネル 1 の original のプロファイル情報を示す．ハードウェアモニタを用いたプロファイル機能により，L1D キャッシュミスの情報を取得することができる．分母を L1D キャッシュミス数とし，ロード・ストア命令による L1D キャッシュミス数の割合を L1D ミス dm 率，同様にプリフェッチ命令による L1D キャッシュミス数の割合を L1D ミス swpf 率とする．カーネル 1〜3 は全体的に L1D ミス dm 率が高く，カーネル 1 で 46.9%，カーネル 2 で 20.6%，カーネル 3 で 19.5%となっており，とくにカーネル 1 が高い．この現象については，適切なプリフェッチ命令が生成されていないことが主要因と推察する．

当初，コンパイラオプションによるソフトウェアプリフェッチの生成では L1D ミス dm 率の減少にはほとんど影響を与えなかった．次に，カーネル 1 のコード上に含まれるマクロと関数のインラインを手動で展開し，適正なプリフェッチ数を見積もったところ 19 となった．一方，コンパイラが提供するリスト情報から得られるプリフェッチ数は 8 となっており，明らかに過小評価されていることが分かった．過小評価の原因は，マクロ展開およびインライン展開前のコードに対してプリフェッチ数を見積もっていたことにある．現状は展開後のコードに対してプリフェッチ数を見積もるようにコンパイラの改良を行った．

結果，表 1 に示すように，カーネル 1 についてプリフェッチ命令数が増加し，L1D ミス dm 率が低下した．また，浮動小数点ロードキャッシュアクセス待ちの時間は，original で 0.04 s だったものが，option-1 で 0.026 s，option-2 で 0.025 s まで減少した．浮動小数点演算ピーク比についても向上の傾向が得られた．

3.6　浮動小数点演算待ちの改善

次に浮動小数点演算待ちの削減を試みる．浮動小数点演算待ちの要因としては，イタレーション間の命令スケジューリングに問題があるためパイプラインのストールが発生していると 3.3 節において推察した．

ここで浮動小数点演算待ちと命令スケジューリングについて補足する．SPARC64^{TM} VIIIfx はサイクルあたり最大 4 命令のコミットが可能であるが，実際は演算器とレジスタ等のプロセッサ固有のリソースによる制約があり，もう 1 つはプログラム上の実行順序に起因する制約により，命令コミット数はサイクルあたり 0 命令から 4 命令のいずれかになる．詳細プロファイラでは，1 命令，2–3 命令，4 命令コミットの 3 つの分類でコスト情報が提供される．それ以外はメモリアクセス待ち，キャッシュアクセス待ち，演算待ち，ストア待ち，バリア同期待ち等の「待ち」になる．3.3 節のように SIMD 命令による演算器の利用効率を改善した場合，命令コミット数は減少するが，命令スケジューリングに必要なリソースが不足するため，副作用として浮動小数点演算待ちが増加することが多い．コンパイラによる命令スケジューリングの最適化は，命令列のオーダリングを入れ替えることでリソースを有効利用し，「待ち」を小さくすることを目的としている．

代表的な命令スケジューリングの最適化としてソフトウェアパイプライニングがある．この手法は，ループアンローリングを行いイタレーション内の命令列を増加させることで命令スケジューリングの最適化の幅を広げる．本質的には，シーケンシャルに実行されるブロックのボディを大きくすることで，依存関係のない命令列の組合せを増やすことを目的としている．

ここでカーネルについて精査すると，1) カーネル 1 と 2

(1) original (An example from the kernel-1)

```
if (ibt < _NBT-1) {
    __s_mult_forw_t_hpc_((ut_ptr),(yd_ptr+1),yy);
}
ut_ptr--;

if (iobz < _NOBZ-1) {
    __s_mult_forw_z_hpc_((uz_ptr),(yd_ptr+_NBT),yy);
}
```

(2) tuned (An example from the kernel-1)

```
if ((ibt < _NBT-1) && (iobz < _NOBZ-1)) {
  __s_mult_forw_t_hpc_((ut_ptr),(yd_ptr+1),yy);
  __s_mult_forw_z_hpc_((uz_ptr),(yd_ptr+_NBT),yy);
}
else if (ibt < _NBT-1) {
  __s_mult_forw_t_hpc_((ut_ptr),(yd_ptr+1),yy);
}
else if (iobz < _NOBZ-1) {
  __s_mult_forw_z_hpc_((uz_ptr),(yd_ptr+_NBT),yy);
}
ut_ptr--;
```

図 11　イタレーション内の分岐結合（カーネル 1）．(1) オリジナル，(2) 分岐結合

Fig. 11　Tuning by branch fusion in an iteration (kernel-1). (1) original, (2) tuned for branch fusion.

についてイタレーション間にデータ依存があること，2) 最内回転数が 6 と小さいこと，3) イタレーション内に分岐が含まれていることが特徴付けられる．ただし，1) についてはアルゴリズム由来であること，2) については問題サイズに依存した事項であり，またショートループの命令スケジューリングの改善は容易でないため，ここではコーディングによる変更を中心に検討を行った．

カーネル 1 および 2

性能阻害要因は，カーネル 1 と 2 については，イタレーション間にデータ依存があるため，次イタレーションの命令を取り込むことができないことにある．そのため，**図 11** のように，イタレーション内の分岐を手動で結合することで，命令スケジューリングを促進させる方法を採用した．

分岐結合の結果，命令スケジューリングが改善され，浮動小数点演算待ちの減少が確認された．また，副次的な効果によりストールが減少し，1 命令コミットから 2–3 命令コミットの割合が増加していることも確認できた．結果，浮動小数点演算ピーク比は，カーネル 1 で 34.9%から 38.1%に，カーネル 2 で 32.9%から 34.8%に改善した．

表 2 に分岐結合の効果について経過時間の内訳を示す．浮動小数点演算待ちが小さくなり，1 命令コミットが改善されていることが分かる．その他のコストが増加しているが，大部分は浮動小数点ロードキャッシュアクセス待ちである．一方で，浮動小数点演算待ちの時間がカーネル 1 の場合 0.048 s から 0.025 s に減少しているのに対して，カーネル 2 の場合 0.069 s から 0.061 s と，カーネル 2 への改善効果が小さいことが分かる．ここで再度カーネルを精査すると，カーネル 1 とカーネル 2 の処理内容はほぼ同等であるが，処理方向が前進か後退かの違いがある．精査すると，

表 2　分岐結合（カーネル 1 および 2）

Table 2　Results for branch fusion tuning (kernel-1 and kernel-2).

	カーネル 1		カーネル 2	
	改善前[s]	改善後[s]	改善前[s]	改善後[s]
4 命令コミット	0.055	0.043	0.049	0.048
2-3 命令コミット	0.077	0.100	0.085	0.090
1 命令コミット	0.079	0.064	0.082	0.061
浮動小数点演算待ち	0.048	0.025	0.069	0.061
その他	0.041	0.042	0.035	0.042
全体	0.300	0.274	0.320	0.302

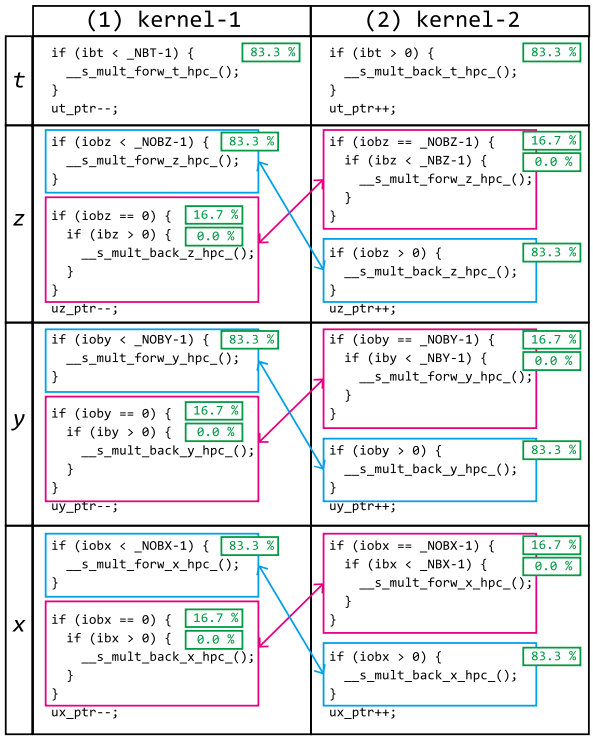


図 12　分岐順序の入替え（IF 文の右側長方形内に真率．矢印間で真率が逆転）

Fig. 12　Tuning by interchange for branch ordering.

図 12 に示すように，実行時にカーネル内に複数ある分岐の真率の並びに違いが生じていることが判明した．

分岐は時空間 x，y，z，t の軸ごとに境界判定を行っており，独立しているので，処理順序の入替えが可能である．カーネル 2 をカーネル 1 と同じ真率の順になるように書き換えを行った結果，カーネル 2 の浮動小数点演算待ちの時間 0.069 s から 0.034 s に減少し，浮動小数点演算性能についても，32.9%から 38.0%に向上が確認された．結果，カーネル 1 とカーネル 2 における改善効果は同等なものが得られた．

カーネル 3

カーネル 3 については，イタレーション間にデータ依存がないため，次イタレーションの命令を取り込むことができる．**図 13** にオリジナルコードを示す．ここでははじめ

original (An example from the kernel-3)

```
for (int  ibt = 0;  ibt <  _NBT;  ibt++){
                :
     if (ibt < _NBT-1) {
       __s_mult_forw_t_hpc_((ut_ptr),(y_ptr+1),yy);
     }
     ut_ptr++;
                :
     y_ptr++;
}
```

図 13　イタレーション間の分岐結合（カーネル 3・オリジナル）
Fig. 13　An example of branch fusion between iterations (kernel-3 original).

(1) loop unrolling

```
for (int  ibt = 0;  ibt <  _NBT;  ibt=ibt+3){
                :
     if (ibt < _NBT-1) {
       __s_mult_forw_t_hpc_((ut_ptr),(y_ptr+1),yy);
     }
                :
     if (ibt+1 < _NBT-1) {
       __s_mult_forw_t_hpc_((ut_ptr+1),(y_ptr+2),y2);
     }
                :
     if (ibt+2 < _NBT-1) {
       __s_mult_forw_t_hpc_((ut_ptr+2),(y_ptr+3),y3);
     }
                :
     ut_ptr++; ut_ptr++; ut_ptr++;
                :
     y_ptr++; y_ptr++; y_ptr++;
}
```

(2) branch fusion

```
for (int  ibt = 0;  ibt <  _NBT;  ibt=ibt+3){
                :
     if ((ibt < _NBT-1) && (ibt+1 < _NBT-1) && (ibt+2 < _NBT-1)) {
        __s_mult_forw_t_hpc_((ut_ptr),(y_ptr+1),yy);
        __s_mult_forw_t_hpc_((ut_ptr+1),(y_ptr+2),y2); // next iteration
        __s_mult_forw_t_hpc_((ut_ptr+2),(y_ptr+3),y3); // next-next interation
     }
     else if ((ibt < _NBT-1) && (ibt+1 < _NBT-1)) {
        __s_mult_forw_t_hpc_((ut_ptr),(y_ptr+1),yy);    // next iteration
        __s_mult_forw_t_hpc_((ut_ptr+1),(y_ptr+2),y2); // next-next iteration
     }
     else if ((ibt < _NBT-1) && (ibt+2 < _NBT-1)) {
        __s_mult_forw_t_hpc_((ut_ptr),(y_ptr+1),yy);
        __s_mult_forw_t_hpc_((ut_ptr+2),(y_ptr+3),y3); // next-next iteration
     }
     else if ((ibt+1 < _NBT-1) && (ibt+2 < _NBT-1)) {
        __s_mult_forw_t_hpc_((ut_ptr+1),(y_ptr+2),y2); // next-iteration
        __s_mult_forw_t_hpc_((ut_ptr+2),(y_ptr+3),y3); // next-next iteration
     }
     else if (ibt < _NBT-1) {
        __s_mult_forw_t_hpc_((ut_ptr),(y_ptr+1),yy);
     }
     else if (ibt+1 < _NBT-1) {
        __s_mult_forw_t_hpc_((ut_ptr+1),(y_ptr+2),y2); // next-iteration
     }
     else if (ibt+2 < _NBT-1) {
        __s_mult_forw_t_hpc_((ut_ptr+2),(y_ptr+3),y3); // next-next iteration
     }
     ut_ptr++; ut_ptr++; ut_ptr++;
                :
     y_ptr++; y_ptr++; y_ptr++;
}
```

図 14　イタレーション間の分岐結合（カーネル 3）．(1) ループアンローリング，(2) 分岐結合
Fig. 14　Tuning for the kernel-3. (1) loop unrolling, (2) branch fusion.

に**図 14**-(1) に示すように手動でループアンローリングし，図 14-(2) のように展開元ループと展開されたループの分岐を結合することでイタレーション内の命令スケジューリングを促進させた．

表 3 に経過時間の内訳を示す．カーネル 3 についても浮動小数点演算待ちが減少し，1 命令コミットから 2–3 命令コミットへの改善傾向が得られた．カーネル 1 および 2 に比べると削減値が大きいことが分かる．イタレーション

表 3　ループアンローリングと分岐結合（カーネル 3）
Table 3　The loop unrolling and branch fusion (kernel-3).

	カーネル 3	
	改善前[s]	改善後[s]
4 命令コミット	0.075	0.065
2-3 命令コミット	0.107	0.148
1 命令コミット	0.110	0.059
浮動小数点演算待ち	0.091	0.032
その他	0.026	0.054
全体	0.409	0.358

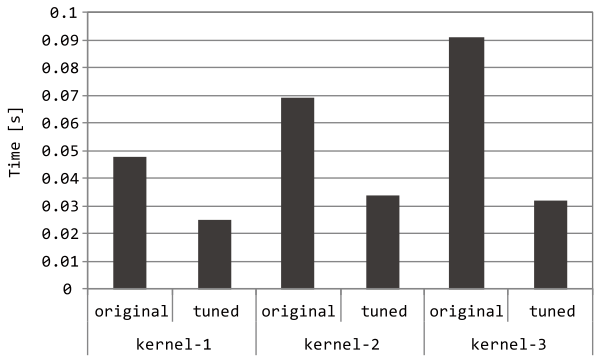


図 15　浮動小数点演算待ちの改善
Fig. 15　Results of waiting for floating-point operation.

間の依存がないため，命令スケジューリングの最適化の幅が大きいためと考える．なお，その他がカーネル 1 および 2 同様増加しているが，大部分は浮動小数点ロードキャッシュアクセス待ちである．

これらのチューニングにより実行時間が，カーネル 1 で 0.300 s から 0.274 s，カーネル 2 で 0.320 s から 0.277 s，カーネル 3 で 0.409 s から 0.358 s に短縮された．**図 15** に浮動小数点演算待ちの改善結果をまとめる．浮動小数点演算ピーク比について，カーネル 1 で 38.1%，カーネル 2 で 38.0%，カーネル 3 で 44.9%に達した．

3.7　バリア同期待ち時間の改善

最後に 1 チップ（8 コア）で実行した場合のチューニングについて述べる．3.6 節で示したチューニング実施後に，1 コアでの浮動小数点演算ピーク比は，カーネル 1 で 38.1%，カーネル 2 で 38.0%，カーネル 3 で 44.9%となり，カーネル 1〜3 全体で 39.5%となった．これを 1 チップで実行した場合に，浮動小数点演算ピーク比が 30.7%まで低下した．1 コアと 1 チップ実行のプロファイル情報を比較すると，ボトルネック要因として，バリア同期待ち，浮動小数点ロードキャッシュアクセス待ち，命令フェッチ待ちの増加が確認された．その中でもバリア同期待ちの増加が顕著に確認された．また，バリア同期待ち以外のコストについてもスレッド化が起因になっていると推察し，後段で述べる簡易バリアコードを用いて検証を行った．

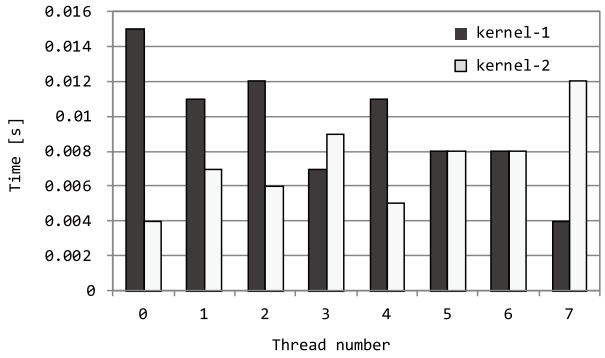


図 16　スレッド間のバリア同期待ち時間（カーネル 1 および 2）
Fig. 16　Results of waiting for barrier synchronization between threads (kernel-1 and kernel-2).

図 16 にカーネル 1 および 2 における，スレッドごとのバリア同期待ち時間を示す．カーネル 1 および 2 には，スレッド並列化した際に各スレッドが持つ境界部分の計算量に偏りがあり，さらに最内ループに omp barrier 指示子によるバリア同期処理が存在するため，スレッドあたり最大 0.015 s 程度のバリア同期待ちが確認できる．これらがカーネル 1 と 2 のボトルネックの主要因であり，プロファイル情報にバリア同期待ちとして現れる原因である．一方，カーネル 3 にはスレッド間の演算量に偏りはないため，最内ループにバリア同期があっても 0.002 s 程度と小さく，ここでは無視する．

本来，スレッド間の同期はハードウェアバリアを利用しているため通常は問題になりにくいが，今回のようにバリア同期が最内にあり，最外まで含めたループの総回転数が 21 万程度と比較的大きい場合は，バリア同期待ち時間がボトルネックとして現れる．その一方で，並列性と収束性を考慮したオーダリングに関するアルゴリズムを採用しているため，一般的にはバリア同期を取り除くことは容易でないと判断し，実装面から再度調査し検討を行った．

まずはじめに，**図 17**-(1) 標準ライブラリを用いる版と，(2) 標準ライブラリ版に含まれる最低限のバリア同期処理を抜き出した簡易バリアコード版を用いて調査を行った．ここではバリア同期の呼び出し回数をカーネルと同じに設定している．なお，図 17-(2) の簡易バリアコードをそのまま用いた場合は，コンパイラによってインライン展開されないため，レジスタの退避と復元によるレジスタスピルが発生し，性能が低下する現象を確認している．このような場合は，空きレジスタを意識したハードコーディングによる書き換えが必要となる．

図 18 に標準ライブラリ版と簡易バリアコード版のプロファイル情報を示す．全体の時間は異なっているが，浮動小数点ロードキャッシュアクセス待ちと整数ロードキャッシュアクセス待ちが，簡易バリアコードにおいても同程度含まれており，標準ライブラリで確認された「待ち」の本質的な問題が含まれていることが分かる．

(1) barrier code by standard libarary

```
#pragma omp parallel
{
  for(int j=0; j<217217; j++) {
#pragma omp barrier
  }
}
```

(2) barrier code by inline assembler

```
#pragma omp parallel
{
  for(int j=0; j<217217; j++) {
        asm("set        0x00,%l1") ;
        asm("ldxa       [%l1]0xef,%l2") ;
        asm("and        %l2,1,%l2") ;
        asm("not        %l2") ;
        asm("and        %l2,1,%l2") ;
        asm("stxa       %l2,[%l1]0xef") ;
        asm("membar #StoreLoad") ;
        asm("loop1 :") ;
        asm("ldxa       [%l1]0xef,%l3") ;
        asm("and        %l3,1,%l3") ;
        asm("subcc      %l3, %l2, %g0") ;
        asm("bne,a      loop1") ;
        /*asm("nop") ;*/
        asm("sleep") ;
  }
}
```

図 17　スレッド間バリア同期のチューニング．(1) 標準ライブラリ版，(2) 簡易バリアコード版
Fig. 17　Tuning for barrier synchronization between threads. (1) standard library, (2) tuned barrier code.

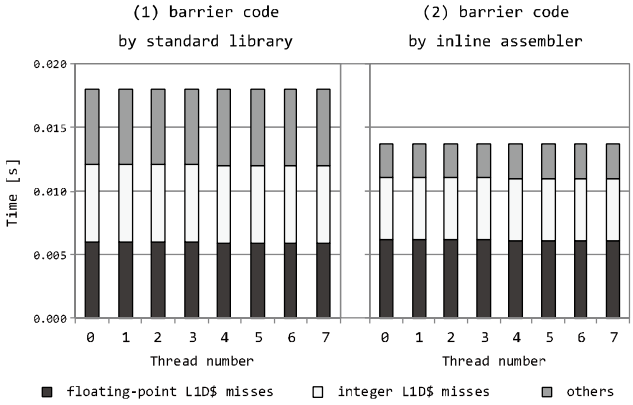


図 18　バリア同期のプロファイル情報．(1) 標準ライブラリ版，(2) 簡易バリアコード版
Fig. 18　Results of profiling for barrier synchronization. (1) standard library, (2) tuned barrier code.

さらに簡易バリアコードのアセンブラコードについて分析を行った結果，浮動小数点ロードキャッシュアクセス待ちは，member #StoreLoad 命令によって生じていることが判明した．また，命令フェッチ待ちの増加については，イタレーション内に omp barrier 指示子が実行されることで，バリア同期処理ライブラリ内の sleep 命令実行時にパイプラインがフラッシュすることで命令フェッチ待ちが発生することが原因であることも判明した．以上，スレッド化時に確認された浮動小数点ロードキャッシュアクセス待ち，整数ロードキャッシュアクセス待ち，命令フェッチ待ちの増加は，バリア同期自体が原因であることが分かった．

表 4 にカーネル 1 についてのバリア同期待ちチューニングの結果を示す．比較のためにバリアなしの場合についても計測した．経過時間は 1 コアの場合で 0.277 s となっており，1 チップ 8 スレッドで実行することで，標準ライ

表 4　バリア同期待ち時間の改善（カーネル 1）
Table 4　Results of waiting for barrier synchronization (kernel-1).

	1 コア [s]	1 チップ 8 スレッド		
		標準ライブラリ [s]	簡易バリアコード [s]	バリアなし [s]
全体	0.277	0.071	0.057	0.043

ブラリの場合で 0.071 s，簡易バリアコードで 0.057 s に短縮となっており，チューニングの効果が確認できた．

なお，アルゴリズムとコーディングを再検討することで，カーネル 1 と 2 については最内ループにあった omp barrier 指示子を 1 つ外側のループに移動できることが判明した．その変更を適用した結果，カーネル 1 については 0.044 s，カーネル 2 については 0.043 s に短縮され，バリアなしと同程度まで短縮した．

3.8　チューニング全体について

すべてのチューニングを適用することで，浮動小数点演算ピーク比について，カーネル 1 で 1 コアあたり 23.2%から 38.1%，カーネル 2 で 24.3%から 38.0%，カーネル 3 で 23.6%から 44.9%に達した．その後，スレッド並列化後の性能最適化を行うことで，1 チップあたりカーネル 1 で 29.5%，カーネル 2 で 30.9%，カーネル 3 で 37.8%の実行効率の改善が得られた．

チューニングの副作用について補足する．チューニングでは 1 つのボトルネックを改善した結果，別の新しいボトルネックが生じることが一般的である．今回のチューニングでは，L2 キャッシュにすべてのデータが載っており，よりレイテンシの大きなメモリアクセスによる「待ち」がほとんど見られなかった．唯一顕著だったのは，3.3 節において SIMD 命令率の改善の結果，演算密度が高くなることで浮動小数点演算待ちが大きくなる副作用が確認されたが，これについては命令スケジューリングに着目した改善によって影響を抑えることができた．

4.　議論

3 章で述べた 5 つのボトルネックに対する改善方法について，SPARC64^{TM} VIIIfx 以外の命令セットアーキテクチャへの適用可能性と，コンパイラに反映するうえでの LDDHMC 以外のアプリケーションについての汎用性について言及する．最後にチューニング全体について総括する．

4.1　SIMD 命令率の改善

SIMD 命令率の改善では，積和演算に着目し，SIMD-FMA 命令の促進を目的に富士通 C コンパイラが提供する SIMD 組込み関数を利用した．ここでは，同様のチューニングが他の計算機環境に対して可能か，プロセッサとコンパイラのサポートの両面について言及する．

POWER7 [15] プロセッサは，Power 命令セットを採用するスーパースカラ型プロセッサであり，浮動小数点数に対する SIMD 演算を Vector-Scalar Extension（VSX）[16] 命令セットによりサポートしている．この VSX には FMA 命令が含まれており，IBM XL C/C++ コンパイラは，この命令セットをサポートする組込み関数が用意されている [17], [18]．また，Intel-64 命令セットを採用するプロセッサでは，現在開発中の Intel 社の Haswell マイクロアーキテクチャ（以後，Haswell）[19] より FMA 命令が採用される予定であり，また AMD 社の Bulldozer マイクロアーキテクチャ [20] においては，すでに FMA 命令に対応している．なお，Intel C/C++ コンパイラ（以後，Intel コンパイラ）は，すでに Haswell 向けの FMA 命令の組込み関数を提供している [21], [22]．それぞれの FMA 命令は，命令セットアーキテクチャごとにオペランドの与え方に若干の相違が見られるが，機能としてはほぼ同様のものがサポートされており，本稿で示した SPARC64^{TM} VIIIfx 向けのチューニングと同様の方法を適用することはそれほど困難ではないと考える．

また本稿では，チューニングで行った SIMD-FMA 命令の積極的な生成を，コンパイラの最適化として行うように改良した．これは SPARC64^{TM} VIIIfx では，basic 側と extended 側のレジスタに値をそれぞれ配置し SIMD-FMA 命令を用いることでレジスタ間のコピー命令を抑止することが，実行時間の削減に有効であるという事実に基づいている．命令数の削減はレイテンシの削減に関連するので，一般的にこのチューニング手法は他アーキテクチャでも有効であると考える．ただし，この手法は命令セットアーキテクチャの特性に大きく依存しているため，必要となるレジスタ間のコピー命令の総レイテンシが SIMD-FMA 命令よりも小さい場合はそれほど効果がない可能性がある．

さらに，この手法を用いることで得られる利点について，コンパイラの観点で補足すると，インラインアセンブラで同等の記述をするとプログラミング言語で記述されたブロックとインラインアセンブラのブロックが分断されることでコンパイラによる最適化が阻害される可能性があるのに対して，組込み関数を用いることで変数の依存関係を静的に解析できる余地を残すことができるため，コンパイラによる最適化の幅を広げることができる．そのような意味で，組込み関数を用いた最適化は，アセンブラと高級プログラミング言語を用いてコーディングする方法の中間的な特徴を持たせる効果があると考える．

最後に，FMA 命令が対象とする積和演算は科学技術計算のアプリケーションにおいて一般的であるので，その効果は十分汎用性があると考える．

4.2　整数ロードキャッシュアクセス待ちの改善

多くのコンパイラでは，SIMD 命令の bit 長に合ったベクトル型定義が用意されている．富士通 C コンパイラの組込み関数の場合は，主に倍精度浮動小数点数が 2 個（128 bit 長），単精度浮動小数点数が 2 個（64 bit 長）格納することができるベクトル型定義が提供されている．Intel コンパイラにおいても，単精度および倍精度浮動小数点数を複数個格納可能な 128 bit 長および 256 bit 長のベクトル型定義があり，本質的に同様なものが提供されている．

LDDHMC カーネルでは，組込み関数が提供する型と，構造体によるユーザ定義型の間で型変換（代入）を行うコーディングがされていた．これにコンパイラによるメンバ型の解析不足が重なり，代入処理において整数型のロード・ストア命令を生成していた．結果的に整数ロードキャッシュアクセス待ちが生じていた．

一般的に言語間結合やライブラリ呼び出し，または異なる背景で設計されたプロシージャとの接合箇所は実装が不自然なものになりやすい．LDDHMC カーネルは，元々 Fortran でコーディングされており，構造体によるユーザ定義型が使用されていた．一方，チューニングのために組込み関数を用いるように書き換えた結果，カーネル部分については構造体によるユーザ定義型と組込み関数が提供する型が共存した状態でコーディングされていた．今回のようなケースは，コンパイラが解析しやすいようにリファクタリングすることが，アプリケーションを問わず一般的に重要であると考える．

その一方，構造体によるユーザ定義型の変数を引数および戻り値に使用することは典型的なパターンであり，また今回のチューニングにおいて最も性能改善にインパクトがあったという点で，きわめて大きなボトルネックになる可能性があることをこのケースは示していると考える．これについてはコンパイラ改良の今後の課題としている．

4.3　浮動小数点ロードキャッシュアクセス待ちの改善

スーパースカラ型プロセッサの場合，1 サイクルに複数の命令を発行・実行することができるため，実行ユニットへのデータ供給が滞りなく行われることが重要である．一方，ロード・ストア命令により L1D キャッシュミスし，L2 キャッシュまたはメモリへのアクセスが発生した場合は「待ち」が発生する．この待ちを小さくするためにデータのプリフェッチを行う．SPARC64^{TM} VIIIfx プロセッサの場合，通常はハードウェアプリフェッチとソフトウェアプリフェッチが協調して機能する．ハードウェアプリフェッチは，ユーザプログラムに対して透過的に機能するのに対して，ソフトウェアプリフェッチはコンパイラがプログラムを静的解析し，命令として生成することで機能する．

アプリケーションから見たとき，配列に対する連続アクセスが頻出するステンシル計算のような場合，配列アクセスのパターンから適切なプリフェッチが先行して行われることが望ましい．たとえば，$y \leftarrow y + a(i) + b(i)$ のように右辺の 2 個の一次元配列を，左辺のスカラ変数に総和を求める場合は，参照する 2 つの配列に対してプリフェッチが先行して発行されることが望ましい．適切なプリフェッチ数より少ない場合は，L1D ミス dm 率が増加し，L2 キャッシュへのアクセス待ちが生じる可能性がある．反対にプリフェッチ数が過剰な場合は，メモリバンド幅を浪費するか，プリフェッチ命令自体のコミット時間により性能が低下する可能性がある．そのような意味で，適切なプリフェッチ数をコンパイラが把握することは重要である．

今回明らかになったボトルネックは，ソースコードから静的に解析して得られるプリフェッチ数に対して，実際に発行されるプリフェッチ命令が不足していたことが原因である．さらにいえばソースコードの静的な解析が不十分であったことが本質的な問題であり，改良のポイントは単純である．LDDHMC カーネルに限らず，配列アクセスが存在するアプリケーションならば，プリフェッチが性能に与える影響を無視することはできないため，一般性のある問題として，コンパイラの改良を行った．

4.4　浮動小数点演算待ちの改善

命令スケジューリングの最適化において，コンパイラはプログラムのシーケンシャルな部分（基本ブロック）を抽出し，その中の命令列に対して順序づけを行うことで，サイクルあたりの命令実行数を大きくする．一方，本カーネルでは最内ループ中に境界判定のための分岐が複数存在し，とくにカーネル 1 と 2 については最内イテレーション間に依存があるため，十分な大きさの基本ブロックが確保できない状態となっていた．これに対して，ループアンローリングおよび分岐結合を行うことで基本ブロックのサイズを大きくし，コンパイラによる命令スケジューリングの最適化を促進させた．

一般的に，分岐が含まれるプログラムについては，コンパイラは基本ブロックを超えた命令スケジューリングとして，トレーススケジューリング等を用いることが多い．トレーススケジューリングは，分岐命令ごとに分岐方向を予測し，最も確率が高いと判断された複数の基本ブロックを通る経路を新しい基本ブロックとして統合することで，シーケンシャルなブロックを大きくする手法である．

トレーススケジューリングは中間コードレベルでの分岐結合を行うことであり，一方，今回のチューニングはソースコードレベルでの分岐結合であり，確率予測を行わないという点を除けば本質的には類似した手法であると考える．そのような観点で，手法としては一般的であり他のアプリケーションに対しても汎用性があると考える．

また，分岐予測のための情報がない状態でのコンパイラによる静的な分岐方向の確率予測は基本的に困難があるこ

とと，現象論的な事実から，富士通コンパイラはつねに分岐しないナイーブな予測に基づいて命令列を生成し，命令スケジューリングをしていると推察する．その一方で，スーパースカラ型プロセッサの命令スケジューリングはハードウェア支援とコンパイラの協調で実現されるものであるので，分岐予測のためのハードウェア支援機能を持っていたとしても，コンパイラによる命令スケジューリングの最適化の重要性は変わらないと考える．よって，真率の高い順に分岐順序を入れ替える手法は，「京」の計算機環境では他のアプリケーションに対しても有効な場合が多いと考える．ただし，分岐予測以外に，投機実行，アウトオブオーダ実行等のハードウェア機構と命令スケジューリングは密接に関わっているため，他アーキテクチャについては本手法が有効でない場合があると考える．

さらに補足すると，命令スケジューリングの最適化は，ソースコードおよび中間コードレベル以外に，機械語の命令レベルでも行われる．浮動小数点レジスタが他の命令セットアーキテクチャに比べて多い SPARC64^{TM} VIIIfx の場合は，コンパイラが機械語レベルで行うレジスタ割付けの自由度が高いため，コンパイラの本質的な改良点であると考える．

4.5　バリア同期待ちの改善

チューニングでは，5 重にネストされたループの最内に置かれていたバリア位置を，アルゴリズム上問題ないと判断したうえで，1 つ外側のループに移動する方法を採用した．バリア位置の見直しは，アプリケーションやアーキテクチャに関係なく，ループ中のバリア呼び出し回数に大きな影響を与えるため，経過時間に与える影響は大きい．よって，本手法は他のアプリケーションに対しても，一般性は有していると考える．

その一方で，アプリケーションによってはバリアが移動できない場合がある．そのような場合は，今回示したような簡易バリアコードを用いることで回避する可能性はあると考える．ただし，簡易バリアコードによるバリア同期待ちの改善は，SPARC64^{TM} VIIIfx に限定したチューニングであるので，他アーキテクチャへの適用は困難であると考える．

4.6　チューニング全体について

評価に用いたカーネルは，格子 QCD の主要演算部で BiCGStab 法の収束性の改善を目的とした前処理部分であり，数値線形代数の典型的なパターンと考える．本稿では，一般的にベンチマークコードを用いて行うコンパイラの改良を，用途が明確なアプリケーションのカーネルを用いて評価し，改善点としてフィードバックした．カーネルに対するチューニングは，プログラム構造のブロック単位またはステートメント，さらには命令レベルでの評価を実施した．複雑なプロシージャが含まれるアプリケーション全体に比べると十分に単純化されており，ベンチマークが持つ汎用的な要素を有していると考える．一方，クォーク場やゲージ場を表現するのに都合がよいため，構造体によるユーザ定義等，アプリケーションが持つ合目的な要素も残しており，アプリケーションを意識した単体性能の評価を行うには適していると考える．また，「京」を用いた最適化では，1 つのアプリケーションについて系統的に整理されているものはまだ少ないため，他のアプリケーションでも類型の問題を確認した場合の参考になると考える．

5.　まとめ

格子 QCD コードである LDDHMC の単体性能向上について「京」の環境で提供される詳細プロファイラ情報をもとにボトルネックを推察し，チューニングを適用，検証を行った．本カーネルで確認された以下の 5 つのボトルネックについて問題点を明らかにし，チューニングを実施した．チューニングの過程で明らかになった解析結果や手法の一部分については，コンパイラの改良として反映された．

なお文献 [6] では本稿で述べたチューニングをもとにループ構造の簡略化と変更を行い，さらに通信を含めたアプリケーション全体の性能について報告している．

1) SIMD 命令率が低くなる現象には 2 つの原因が含まれていた．1 つはイタレーション内に分岐が含まれ，とくにカーネル 1 および 2 についてはイタレーション間にデータ依存があるため，コンパイラによる SIMD 生成が適切に行われていなかった．これらはコンパイラの解析能力不足が否めないが，アルゴリズムに起因するところが大きい．もう 1 つは単純な積和演算を行う場合でも SIMD-FMA 命令を生成せず fmovd 命令を生成するコンパイラに原因があった．今回のように SIMD 化がすでに実証されている場合は，アプリケーション側で組込み関数を用いることで改善できることを示した．また，今回の解析・チューニング結果に基づいて，より一般的なケースにおいて，fmovd 命令を抑制し，SIMD-FMA 命令が生成されるようにコンパイラの改良を行った．
2) 整数ロードキャッシュアクセス待ちの増加については，組込み関数の型とユーザ定義の構造体間の代入時に生成される整数ロード・ストア命令が原因であった．本質的にはコーディングで回避できる問題であり，ユーザ定義型ではなく処理系が提供する型を用いることで改善できることを示した．一方で，コンパイラの構造体に対する解析処理の改善が課題であることが判明した．
3) 浮動小数点ロードキャッシュアクセス待ちについては，コンパイラのプリフェッチ数の見積り方法に原因

があった．適切なプリフェッチ数を生成するようにコンパイラを改良することで改善されることを示した．

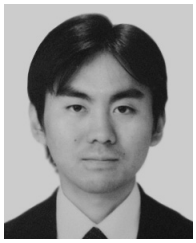
4) 浮動小数点演算待ちについては，イタレーション内に分岐があり，ループ長が短いため，コンパイラによる命令スケジューリングの最適化が阻害されていた．本質的にはアルゴリズムと問題サイズに原因があり，さらに真率の順序が命令スケジューリングに悪影響を与えていた．これについてはループ展開，分岐結合，さらに必要に応じて分岐順序の入替えを行うことで，改善できることを示した．

5) バリア同期の必要性はアルゴリズムに起因するが，プロファイラ情報から確認されたバリア同期待ち時間の大半はコーディングの問題であり，バリア位置を変更することで改善された．さらに別の方策として，簡易バリアコード等を用いることで改善できることを示した．

謝辞　「京」における格子 QCD コードのチューニングは，理化学研究所と筑波大学との「大規模シミュレーションによる次世代スーパーコンピュータの性能評価に関する共同研究」に基づき進められたものです．本報告に際し，富士通株式会社次世代 TC 開発本部の皆様に感謝いたします．とくに，システムソフトウェア開発者の立場で，ご討論いただいた，青木正樹氏，杉山浩一氏，瀧澤伸悟氏に深く感謝いたします．また本稿の結果は，独立行政法人理化学研究所計算科学研究機構が保有するスーパーコンピュータ「京」の試験利用によるものです．

参考文献

[1] Duane, S., Kennedy, A.D., Pendleton, B.J. and Roweth, D.: Hybrid Monte Carlo, *Phys. Lett. B*, Vol.195, No.2, pp.216–222 (1987).
[2] Ukawa, A., et al.: Computational cost of full QCD simulations experienced by CP-PACS, *Nuclear Phys. B* (*Proc. Suppl.*), Vol.106, pp.195–196 (2002).
[3] Nakamura, A.: Lattice QCD simulations as an HPC Challenge, *ISHPC 2005 and ALPS 2006*, LNCS 4759, pp.441–451 (2008).
[4] Yokokawa, M., Shoji, F., Uno, A., Kurokawa, M. and Watanabe, T.: The K computer: Japanese next-generation supercomputer development project, *ISLPED2011*, pp.371–372 (2011).
[5] Miyazaki, H., Kusano, Y., Okano, H., Nakada, T., Seki, K., Shimizu, T., Shinjo, N., Shoji, F., Uno, A. and Kurokawa, M.: K computer: 8.162 PetaFLOPS massively parallel scalar supercomputer built with over 548k cores, *ISSCC*, pp.192–194 (2012).
[6] Boku, T., Ishikawa, K.-I., Kuramashi, Y., Nakamura, Y., Minami, K., Shoji, F., Takahashi, D., Terai, M., Ukawa, A. and Yoshie, T.: Multi-block/multi-core SSOR preconditioner for the QCD quark solver for K computer, PoS (*Lattice 2012*) 188; arXiv:1210.7398 [hep-lat].
[7] Lüscher, M.: Solution of the Dirac equation in lattice QCD using a domain decomposition method, *Comp. Phys. Comm.*, Vol.156, pp.209–220 (2004).
[8] SPARC64^{TM} VIIIfx Extensions (2010).
[9] Maruyama, T., Yoshida, T., Kan, R., Yamazaki, I., Yamamura, S., Takahashi, N., Hondou, M. and Okano, H.: SPARC64 VIIIFX: A new-generation octocore processor for petascale computing, *IEEE Micro*, Vol.30, Issue 2, pp.30–40 (2010).
[10] SPARC Joint Programming Specification (JPS1): Commonality (2002).
[11] Linzer, E.N. and Feig, E.: Implementation of Efficient FFT Algorithms on Fused Multiply-Add Architectures, *IEEE Trans. Signal Processing & Analysis*, Vol.41, Issue 1, pp.93–107 (1993).
[12] Intel C++ Intrinsic Reference (2007).
[13] Intel Architecture Optimization Reference Manual (1999).
[14] Parallelnavi Technical Computing Language C User's Guide (2012).
[15] Kalla, R., Sinharoy, B., Starke, W.J. and Floyd, M.: POWER7: IBM's next-generation server processor, *IEEE Micro*, Vol.30, No.2, pp.7–15 (2010).
[16] Power ISA Version 2.06 (2010).
[17] IBM XL C/C++ for AIX, V12.1 Compiler Reference (2012).
[18] AIX Version7.1, Assembler Reference Language (2012).
[19] New Instruction Set Extensions, Instruction Set Innovation in Intels Processor Code Named Haswell (2012).
[20] AMD64 Technology, AMD64 Architecture Programmer's Manual Volume 6: 128-Bit and 256-Bit XOP and FMA4 Instructions (2009).
[21] Intel Architecture Instruction Set Extensions Programming Reference (2012).
[22] Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C (2013).



寺井 優晃　（正会員）

2006 年北陸先端科学技術大学院大学情報科学研究科修了．同年理化学研究所に入所．2010 年次世代スーパーコンピュータ開発実施本部にてスーパーコンピュータ「京」の開発に従事．2012 年より理化学研究所計算科学研究機構開発研究員（現職）．「京」に関するアプリケーションの高度化およびツールに関する開発研究に取り組む．博士（情報科学）．

石川 健一

1972 年生．1999 年広島大学大学院理学研究科物理学専攻修了．2001 年筑波大学物理学系助手．2003 年広島大学理学研究科物理科学専攻講師．2005 年同大准教授（現職）．専門は格子 QCD を用いた素粒子物理学の研究．格子 QCD における大規模並列計算と連立方程式の解法についての研究も行っている．日本物理学会会員．博士（理学）．

杉崎 由典

1985 年静岡大学工学部情報工学科卒業．同年富士通静岡エンジニアリング入社．現在，富士通（株）．prolog，VPPFortran ライブラリ，HPF ライブラリ開発．HPC 系の性能解析および性能チューニングに従事．

南 一生

1981 年日本大学理工学部物理学科卒業．同年富士通株式会社入社．主に原子力分野のシミュレーションコードのスパコンへの性能最適化の仕事に従事．2000 年財団法人高度情報科学技術研究機構入社．地球シミュレータ用ソフトウェア性能最適化研究に従事．2008 年理化学研究所次世代スーパーコンピュータ開発実施本部開発グループアプリケーション開発チームリーダー，2012 年理化学研究所計算科学研究機構運用技術部門ソフトウェア技術チームヘッド．2011 年ゴードン・ベル賞受賞．

庄司 文由

1998 年金沢大学大学院自然科学研究科単位取得退学．1998 年広島大学情報教育研究センター助手．2001 年広島大学情報メディアセンター助手．2006 年理化学研究所次世代スーパーコンピュータ開発実施本部開発研究員．2010 年同チームリーダー．2012 年理化学研究所計算科学研究機構運用技術部門副部門長兼システム運転技術チームチームヘッド．京コンピュータの運用および高度化に従事．博士（理学）．

中村 宜文

2005 年金沢大学大学院自然科学研究科物質構造化学専攻博士後期課程修了．同年ドイツ電子シンクロトロン研究所に博士研究員として入所．2008 年レーゲンスブルク大学理論物理学研究所に博士研究員として入所．2010 年筑波大学計算科学研究センター研究員を経て現職．理化学研究所計算科学研究機構研究員．計算素粒子物理学の研究に取り組む．物理学会員．博士（理学）．

藏増 嘉伸

1995 年東京大学大学院理学系研究科物理学専攻修了．博士（理学）．高エネルギー加速器研究機構素粒子原子核研究所助手，筑波大学計算科学研究センター講師，准教授を経て同大学教授（現職）．2010 年より理化学研究所計算科学研究機構連続系場の理論研究チームチームリーダーを兼任．専門は計算素粒子物理学．

横川 三津夫　（正会員）

1984 年筑波大学大学院理工学研究科修士課程修了．同年日本原子力研究所入所後，原子力分野における高速数値シミュレーションの技術開発に従事．1997 年地球シミュレータ研究開発センターにて地球シミュレータ開発に参画．2002 年産業技術総合研究所グリッド研究センター．2006 年理化学研究所次世代スーパーコンピュータ開発実施本部にて，スーパーコンピュータ「京」の開発に従事．2012 年神戸大学システム情報学研究科．2002 年，2011 年ゴードン・ベル賞受賞．工学博士．