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]　石川 健一[†2†1]　杉崎 由典[†3]　南　一生[†1]
庄司 文由[†1]　中村 宜文[†1]　藏増 嘉伸[†4†1]　横川 三津夫[†1†5]

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 法を使うことで殆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 22.7%から 37.1%，カーネル 2 で 23.7%から 37.1%，カーネル 3 で 23.0%から 43.8%に改善した．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%の実行効率の改善が得られた．本稿では，プロファイル情報によって明らかになった問題点及びそれらのチューニング手法について報告する．

Performance Tuning of a Lattice QCD code on a node of the K computer

MASAAKI TERAI[†1]　KEN-ICHI ISHIKAWA[†2†1]　YOSHINORI SUGISAKI[†3]
KAZUO MINAMI[†1]　FUMIYOSHI SHOJI[†1]　YOSHIFUMI NAKAMURA[†1]
YOSHINOBU KURAMASHI[†4†1]　MITSUO YOKOKAWA[†1]

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 ratio, 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 22.7% to 37.1% in the kernel-1, from 23.7% to 30.9% in the kernel-2, from 23.0% to 43.8% in the kernel-3. Finally, the peak performance a chip is 29.5% in the kernel-1, 30.9% in the kernel-2, 37.8% in the kernel-3. This paper describes the problems obtained by profiling and the tuning.

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極めて大きいため，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].

†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
†2 広島大学 大学院理学研究科
Hiroshima University
†3 富士通株式会社
Fujitsu Ltd.
†4 筑波大学 数理物質系
University of Tsukuba
†5 神戸大学大学院システム情報学研究科
Kobe University

「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 法を使うことで殆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，最適化施策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及びそれらのチューニング手法について報告する. 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．

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．

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

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．

$$\mathrm{Mult}(n,m)^{a,b}_{\alpha,\beta} = \sum_{\mu=1}^{4}\Big[(1-\gamma_\mu)_{\alpha,\beta}\left(U_\mu(n)\right)^{a,b}\delta_{n+\hat{\mu},m} + (1+\gamma_\mu)_{\alpha,\beta}\left(U^\dagger_\mu(n-\hat{\mu})\right)^{a,b}\delta_{n-\hat{\mu},m}\Big] \qquad \text{(式 2)}$$

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．

$$D = \begin{pmatrix} D_{EE} & D_{EO} \\ D_{OE} & D_{OO} \end{pmatrix} \qquad \text{(式 3)}$$

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．

$$D\cdot x = b \implies (D\cdot M_{SAP})z = b,\ x = M_{SAP}\cdot z \qquad \text{(式 4)}$$

$$M_{SAP} \equiv K\sum_{j=0}^{N_{SAP}}(1-D\cdot K)^j \qquad \text{(式 5)}$$

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

ここで式 6 におけるブロック要素の逆行列 $(D_{EE})^{-1}, (D_{OO})^{-1}$ 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，スレッド並列の為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^{TM}VIIIfx[8]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^{TM}VIIIfx の L2D キャッシュに載る大きさである．

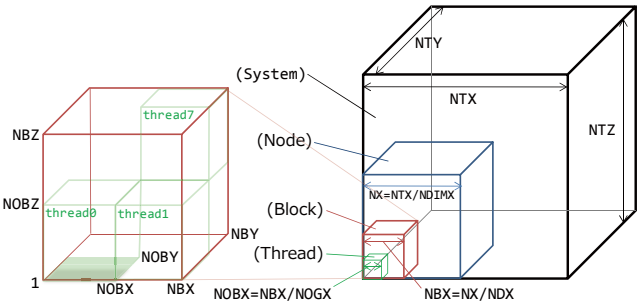

図 1　ノード分割からスレッド並列までの階層
（4 次元目の *T* 軸は省略）

```
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
            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 の擬似コード

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，ここで *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毎に閉じた格子番号をインデックスに持つことで，連続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．

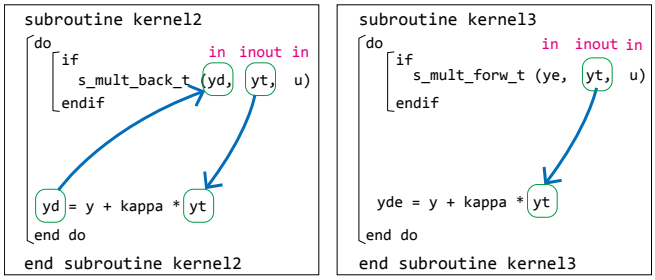

図 3　カーネル 2 および 3 のイテレーション間の依存

3. 単体性能チューニング

3.1 SPARC64^{TM}VIIIfx プロセッサ概要

SPARC64^{TM}VIIIfx プロセッサは，1 チップあたり 8 コアを搭載し，各コアに 2 個の演算器と 32KB の L1D キャッシュ，チップ上にはコア間で共有される 6MB 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 128GFLOPS, 理論メモリバンド幅は 64GB/s である．SPARC V9 命令仕様[9]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 128bit 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 命令仕様を元にした機能拡張を 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一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 通りの組み合わせがあり，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 側と呼ぶ．FMA 命令の一般的な利用例

としては，複素数での内積・行列積が挙げられる．

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，最適化施策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，当初の問題が解決しているか確認した．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，その詳細は公開されている[8]．一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，オリジナルコードにおいて 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明らかになり，それぞれについて最適化施策を実施した．3.3 節以降で，その詳細を述べる．

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．

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

図 4　コンパイラオプション

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)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．

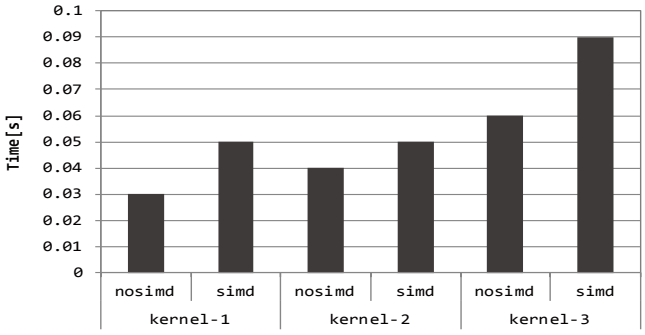

図 5　浮動小数点演算待ちの増加

プロファイルの結果，演算で占める部分にも関わらず，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 演算を実行することができる．今回のケースのように既に 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，富士通製コンパイラが提供する C 組込み関数[10]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．

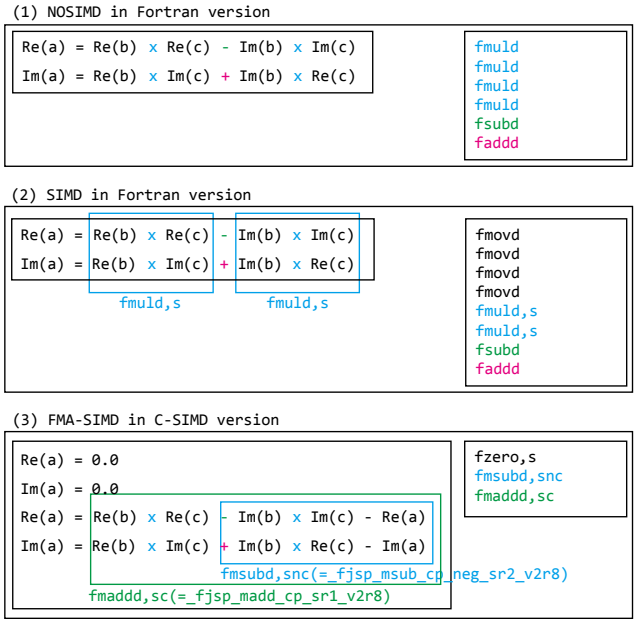

図 6　積和演算の C 組込み関数への書き換え

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し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，もう一つの SIMD 命令率の低下の原因であると推察

する．

まず始めに，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 側レジスタの組み合わ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移動のための fmovd 命令が必要となる．その一方で fmaddd,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．C 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，積和演算に現れる全ての項と 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．図 6-(3)に SIMD-FMA 命令に書き換え後のコード例を示す．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．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．

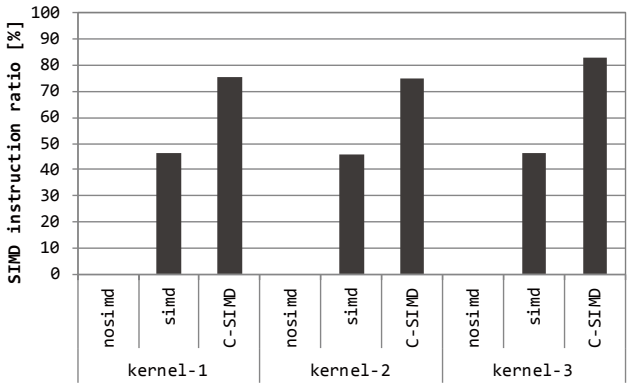

図 7　SIMD 命令率（／総有効命令数）の改善

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．それに伴い，浮動小数点演算ピーク比は，カーネル 1 で 22.7%が 24.0%，カーネル 2 で 23.7%から 24.5%, カーネル 3 で 23.0%から 28.7%に改善した．実行時間もカーネル 1 で，0.49s から 0.44s，0.46s から 0.43s，0.71s から 0.56s 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関わらず, 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 と増加する．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

C 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，間接参照に伴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なっていない．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．その為，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 SIMD 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．

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の構造体のメンバ変数の型の解析処理時間を削減するために，変数の型に依らず整数型命令でコピー処理していたことが整数ロードキャッシュアクセス待ちとして現れた原因であることが判明した．これはコンパイラ改良の今後の課題であると考える．

本稿での対策としては，ソース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ることで，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．結果を図 9 に示す．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 31.9%, カーネル 2 で 33.1%, カーネル 3 で 35.0%に向上した．

(1) 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);
}
```

(2) 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 8　整数 L1D キャッシュミスの解析

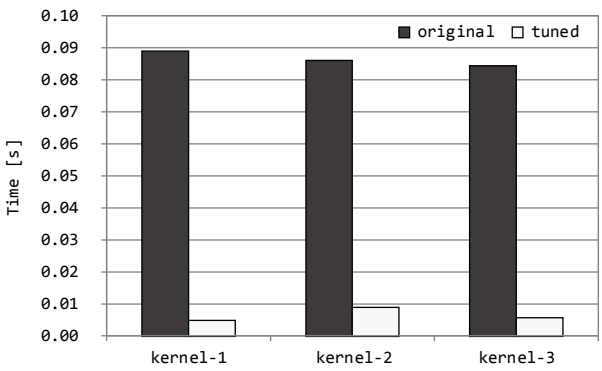

図 9　整数ロードキャッシュアクセス待ちの改善

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．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（デマンド）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(software prefetch)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，特にカーネル 1 が高い．この現象について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 L1D ミスによる時間は，original で 0.04s だったものが，option-1 で 0.026s, option-2 で 0.025s 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．

表 1　プリフェッチ生成による性能改善（カーネル 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	32.4	8.65E+06	46.9%	18.7%
noprefetch	31.9	2.03E+03	45.2%	0.01%
option-1	33.1	1.51E+07	18.4%	51.5%
option-2	33.2	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

以上，スーパースカラ型のプロセッサの場合，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．よって，コンパイラが適切な数のプリフェッチ命令

を生成することで L1D ミス dm 率を減少させることは，本アプリケーションに限らず一般的な性能改善手法であり，それがコンパイルによって適用されることは意義がある．なお，現状提供されているコンパイラには，本研究で必要と判明したプリフェッチは生成されるように改善されている．

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．

ここでカーネルについて精査すると，1) カーネル 1 と 2 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) ループ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，ここではコードの変更で対応可能な 3）について注目し，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 10 のように，ループ内の分岐を手動で結合することで，イタレーション内の命令スケジューリングを促進させる方法を採用した．

(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 10　イタレーション内の分岐結合

分岐結合の結果，浮動小数点待ちの改善が確認された．また，命令スケジューリングが改善したこと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 34.08%から 37.2%に，カーネル 2 で 32.12%から 34.02%に改善した．

ここでカーネル 1 とカーネル 2 の効果について比較すると，浮動小数点演算待ちの時間がカーネル 1 の場合 0.05s から 0.03s に減少しているのに対して，カーネル 2 の場合 0.07s から 0.06s と，カーネル 2 への効果が僅かながら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 の違い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．図 11 に示すように，この違いが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を生じさせていることが判明した．

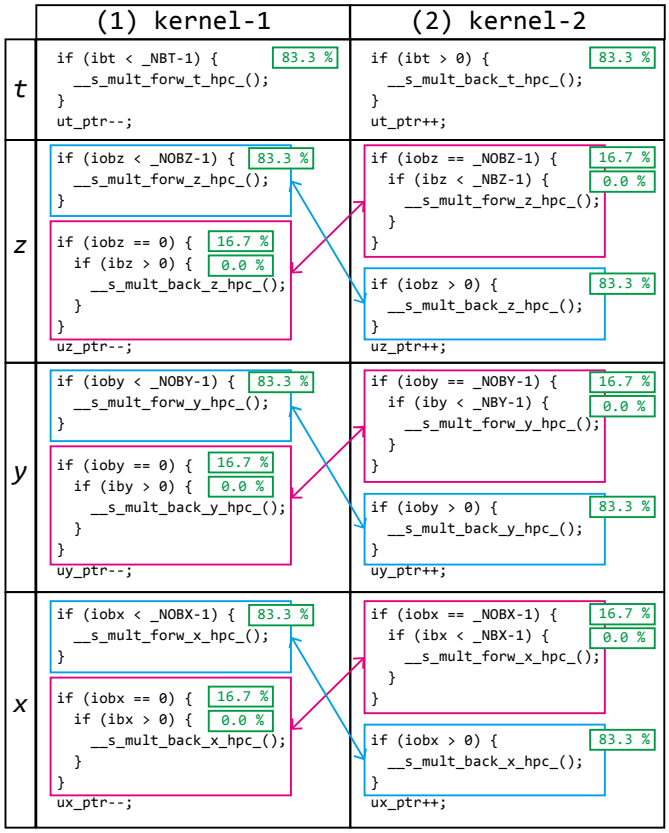

図 11　分岐順序の入替え
（IF 文の右側長方形内に真率．矢印間で真率が逆転.）

分岐は時空間 *x,y,z,t* の軸ごとに境界判定を行なっており，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 0.07s から 0.03s に減少し，浮動小数点演算性能についても，32.1%から 37.1%に向上が確認された．結果，カーネル 1 とカーネル 2 の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．ここでは始めに図 12-(2) に示すように手動でループ展開し，図 12-(3)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．

最適化施策の結果，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[8]への改善傾向が得られた．浮動小数点演算待ちの改善を図 13 に示す．この施策により実行時間が，カーネル 1 で 0.3s から 0.28s，カーネル 2 で 0.32s から 0.28s，カーネル 3 で 0.41s から 0.36s に短縮した．浮動小数点演算ピーク比について，カーネル 1 および 2 で 37.1%，カーネル 3 で 43.8%に達した．

(1) 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++;
}
```

(2) tuned: loop unrolling (An example from the kernel-3)

```
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++;
}
```

(3) tuned: loop fusion (An example from the kernel-3)

```
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 12　イタレーション間の分岐結合

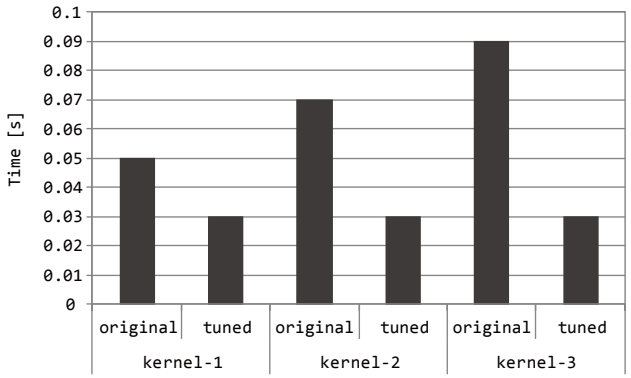

図 13　浮動小数点演算待ちの改善

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．

プロファイル情報から，1 コアでの浮動小数点演算ピーク比は 3.6 節で示した最適化施策後に，カーネル 1 および 2 で 37.1%，カーネル 3 で 43.8%となり，カーネル 1~3 全体で，38.6%が得られている．一方，これを 1 チップで実行した場合に，バリア同期が原因でカーネル全体の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 L1D キャッシュミス, 命令フェッチ待ちの増加が確認された. 図 14 にスレッドごとのバリア同期待ち時間を示す. 結果, 1 チップで見ると, 各スレッドほぼ 0.02s 程度のバリアが確認できた.

カーネル 1 と 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と，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．これがカーネル 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．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大きな問題にはなっていない．

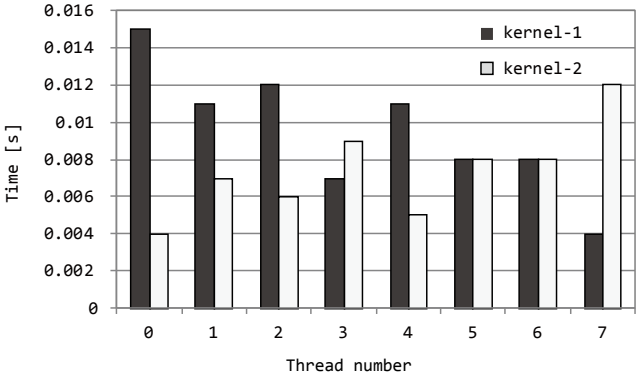

図 14　スレッド間のバリア同期待ち時間

本来，スレッド間の同期はハードウェアバリアを利用し

ているためプロファイル情報では問題になりにくいが，今回のようにループ中のバリア同期回数が比較的大きい場合は，バリア同期待ち時間として現れる．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ことが多い．以上，本稿では実装面から再度調査，検討を行った．

調査には，図 15-(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（ハードバリア固定コード）のみを抜き出した簡易バリアコード版を用いた．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．

図 16 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，簡易バリアコードにおいても，ボトルネック要因となった浮動小数点 L1D キャッシュミスと整数ロードキャッシュアクセス待ちがカーネルと同程度確認できたため，ここでは簡易バリアコードにバリア同期の本質的な問題が含まれているものと考える．

(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 15　簡易バリアコードの概要
(1) 標準ライブラリ版 (2) 簡易バリアコード版

この簡易バリアコードを分析した結果，浮動小数点 L1D キャッシュミスは，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．

なお，簡易バリアコードの留意点であるが，図 15-(2)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．

施策適用の結果，カーネル 1 および 2 の実行時間が 1 コア 0.28s から 1 チップ 0.07s に短縮，カーネルが 0.36s から 0.05s に短縮した．

さらに，アルゴリズムとコーディングを再度検討することで，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ため，それについても適用した結果，カーネル 1 および 2 については 0.04s に短縮した．

以上，すべての施策を適用することで，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 22.7%から 37.1%, カーネル 2 で 23.7%から 37.1%, カーネル 3 で 23.0%から 43.8%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 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．

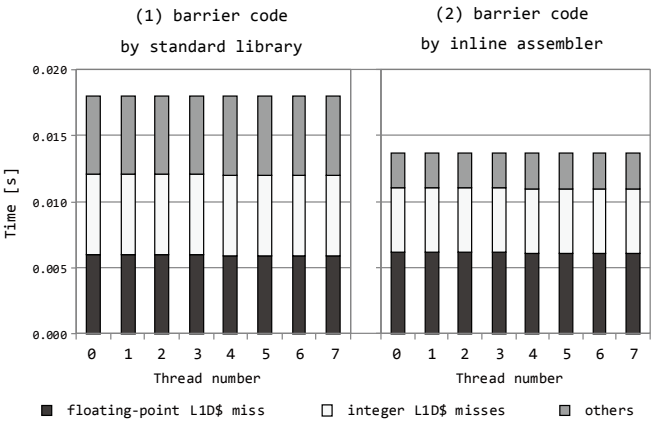

図 16　バリア同期のプロファイル情報
(1) 標準ライブラリ版 (2) 簡易バリアコード版

最内ループのバリア同期の回数を減らせる場合は問題の改善は容易であるが，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．また，「京」を用いた最適化について，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が確認した場合の参考になると考える．

4. 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元にボトルネックを推察し，最適化施策を適用，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．問題点

の一部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] では本稿で報告したチューニングを元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 FMA-SIMD 命令を生成せず fmovd 命令を生成していた．これはコンパイラに原因があった．今回のように SIMD 化が既に実証されている場合は，アプリケーション側で C 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，C 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り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．ただし，コンパイラの構造体に対する解析処理については不足が否めないので対策が必要と判断した．
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．なお，本改善は本番環境に既に反映済みである．
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) バリア同期が必要であるの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 本報告の格子 QCD コードの最適化は，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致します．特に，システムソフトウェア開発者の立場で，御討論頂いた，青木正樹氏，杉山浩一氏，瀧澤伸悟氏に深く感謝致します．また本論文の結果は，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) S.Duane, et al., *Hybrid Monte Carlo*, Phys. Lett. B, vol.195, 2, pp.216-222, (1987).
2) A.Ukawa et al., *Computational cost of full QCD simulations experienced by CP-PACS*, Nuclear Phys. B (Proc. Suppl.), 106, pp.195-196, (2002).
3) A.Nakamura, *Lattice QCD simulations as an HPC Challenge*, ISHPC 2005 and ALPS 2006, LNCS 4759, pp.441-451, (2008).
4) M.Yokokawa, et al., *The K computer: Japanese next-generation supercomputer development project*, ISLPED2011, pp.371–372, (2011).
5) H.Miyazaki, et al., *K computer: 8.162 PetaFLOPS massively parallel scalar supercomputer built with over 548k cores*, ISSCC, pp.192-194, (2012).
6) T.Boku et al., *Multi-block/multi-core SSOR preconditioner for the QCD quark solver for K computer*, PoS (Lattice 2012) 188 ;arXiv:1210.7398 [hep-lat].
7) M.Lüscher, *Solution of the Dirac equation in lattice QCD using a domain decomposition method*, Comp. Phys. Comm., 156, pp.209-220, (2004).
8) *SPARC64™ VIIIfx Extensions*, (2010).
9) *SPARC Joint Programming Specification (JPS1): Commonality*, (2002).
10) *Parallelnavi Technical Computing Language C User's Guide*.