電子動力学シミュレーションのステンシル計算に対するメニーコアプロセッサ向け最適化

廣川 祐太[1,a)]　朴 泰祐[3,1]　佐藤 駿丞[2,†1]　矢花 一浩[3,2]

概要：近年，Intel Xeon Phi などメニーコアプロセッサを搭載した PC クラスタが運用されているが，同プロセッサの性能特性から実アプリケーションにおいて高い性能を得るのは非常に困難である．本研究では，電子動力学シミュレータ ARTED での支配的な計算である波数空間と軌道に関して並列化された 3 次元実空間格子の 25 点ステンシル計算を，メニーコアプロセッサに対し最適化することを目的とする．まず，元のターゲットシステムである京コンピュータ (SPARC64 VIIIfx) に対し最適化を行い，コンパイラによる自動ベクトル化を促進することで 14.94 GFLOPS から 27.2 GFLOPS に性能が向上した．この実装を用いて，メニーコアプロセッサの Intel Xeon Phi (Knights Corner) を対象に，自動ベクトル化と Intrinsics を用いた手動ベクトル化による最適化を行った．元実装が 30.06 GFLOPS であるのに対し，手動ベクトル化実装にて 224.45 GFLOPS と 20.9 %のピーク演算性能比を達成した．また，次世代プロセッサの Knights Landing への実装などについても考察する．

キーワード：性能最適化, ステンシル計算, メニーコアプロセッサ, Intel Xeon Phi, 京コンピュータ

Optimization of Stencil Computation in Electron Dynamics Simulation for Many-Core Processor

Yuta Hirokawa[1,a)]　Taisuke Boku[3,1]　Shunsuke A. Sato[2,†1]　Kazuhiro Yabana[3,2]

Abstract: Recently, PC clusters equipped with the many-core processors such as Intel Xeon Phi are actively operated. However, it is not easy to achieve high sustained performance on real applications because of special characteristics of this sort of processor. In this paper, we focus on an electron dynamics simulation code named ARTED in which 25-points 3-D stencil computation in real space grid parallelized over wave number space as well as orbitals is the core part of computation. First, we optimized its stencil computation to K computer (SPARC64 VIIIfx processor) that is the original target system of ARTED. As a result, the performance improved to 27.2 GFLOPS from 14.94 GFLOPS with automatic vectorization by compiler. Using this implementation, we applied explicit vectorization with intrinsics on its stencil computation part considering the features of current Intel Xeon Phi by Knights Corner architecture. As a result, we improved the sustained performance on a single Xeon Phi from poor original 30.06 GFLOPS to 224.45 GFLOPS on stencil computation which corresponds to approximately 20.9 % of theoretical peak performance of single Xeon Phi. We also discuss on a future implementation on next generation of Knights Landing architecture.

Keywords: Performance optimization, Stencil computation, Many-core processor, Intel Xeon Phi, K-computer

[1] 筑波大学大学院システム情報工学研究科
Graduate School of Systems and Information Engineering, University of Tsukuba
[2] 筑波大学大学院数理物質科学研究科
Graduate School of Pure and Applied Sciences, University of Tsukuba
[3] 筑波大学計算科学研究センター
Center for Computational Sciences, University of Tsukuba
[†1] 現在，筑波大学計算科学研究センター
Presently with Center for Computational Sciences, Univer-

1. 序論

今日，消費電力性能比からメニーコアプロセッサが注目され，約 60 の物理コアを持つ Intel Xeon Phi プロセッサを搭載したシステムが広く運用されている [1]．Xeon Phi プロセッサは，Many Integrated Cores (MIC) アーキテクチャに基づく x86 アーキテクチャベースのプロセッサで，現プロダクトは Knights Corner (KNC) と呼ばれる．KNC 上では，Linux カーネルが動作しているため，Xeon CPU で開発されたアプリケーションをコードの変更なく容易に実行可能だが，Xeon CPU に対してその性能特性は大きく異なる．ここで重要なのは，Xeon CPU 用コードの単純な移植では KNC の高い演算性能を得るのは非常に困難で，KNC に対する最適化が強く要求されることである [2], [3], [4]．一方，現在，次世代 Xeon Phi として Knights Landing (KNL) が開発されている．KNC は PCI-Express で接続されるためホスト CPU を要求し，Xeon CPU と KNC のヘテロジニアスクラスタしか実装できなかった．KNL では，KNL 自体がホスト CPU として利用できるため，KNL のみを利用したクラスタが実装可能である．また，ポスト京コンピュータでも，メニーコア CPU を用いたシステムが想定されている．現在運用されている KNC クラスタは，これらのメニーコアシステムへ向けた評価と実装準備環境としても期待されている．

本研究では，KNC をターゲットに，電子動力学シミュレーションコードに現れる 3 次元 25 点ステンシル計算に対し，シングルコアレベルの最適化を行う．同計算は，空間的に周期境界条件を持つ電子軌道の時間発展を，時空間で差分化して解く計算が主要で，周期境界条件を持つ深さ 4 の中心差分計算のため，本研究は，同様の計算パターンをもつアプリケーションにも寄与すると考えられる．また，本研究の成果が KNC 専用ではなく，今後のメインプロセッサとなることが予測される KNL など他のメニーコアプロセッサにおいても有効であることを考察する．

2. 対象とするステンシル計算

本研究では，筑波大学計算科学研究センターで開発中の電子動力学シミュレーションコード ARTED (Ab-initio Real-Time Electron Dynamics simulator) [5] において主要な計算時間を占める，電子軌道にハミルトニアン演算子を作用する際に現れるステンシル計算の最適化を行う．ARTED は，光科学分野の中心テーマである超短パルス光と物質の相互作用に対し，時間依存密度汎関数理論に基づき第一原理計算を行うアプリケーションである．パルス光が駆動する電子動力学を記述するとともに，パルス光の電磁場と電子の運動を，マルチスケール手法を用いて同時に

sity of Tsukuba
a) hirokawa@hpcs.cs.tsukuba.ac.jp

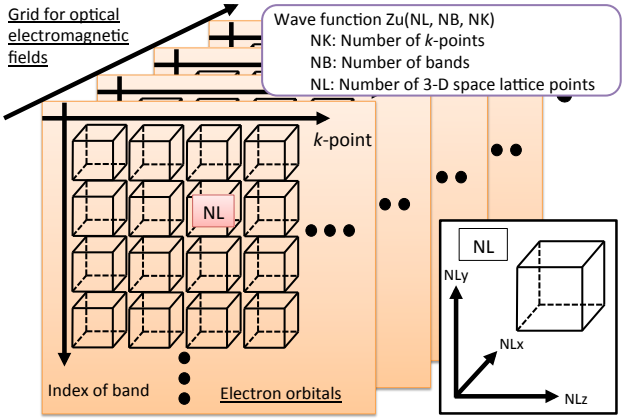

図 1 ARTED の計算領域のイメージ

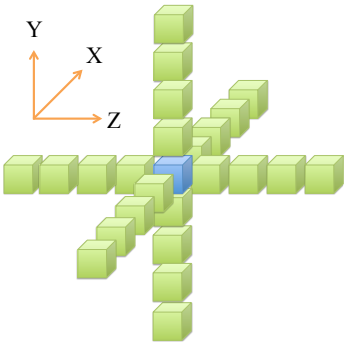

図 2 25 点ステンシル計算のメモリアクセスパターン

記述することが可能である．電子動力学に関しては，電子軌道 (Electron orbitals) に対する時間依存コーン・シャム (TDKS, Time-Dependent Kohn-Sham) 方程式を実時間・実空間法を用いて解き，光電磁場 (Optical electromagnetic fields) に対しては有限差分時間領域 (FDTD, Finite Difference Time-Domain) 法を用いて解く [6]．これらのシミュレーションでは，TDKS 方程式に基づく電子軌道の時間発展計算に必要とされる，ハミルトニアンの電子軌道に対する作用の計算 (以下，ハミルトニアン計算と呼ぶ) が，演算時間の大半を占めている．ハミルトニアン計算は，電子軌道に対するラプラシアンの作用を含み，少ない空間格子点数で高い計算精度を得るために，深さ 4 の中心差分を用いたステンシル計算が行われる．**図 1** に，電子軌道に対する配列 `Zu(NL, NB, NK)` のイメージを示す．電子軌道は，光電磁場の格子点毎に用意され，ブロッホ波数空間の格子点 (`NK`)，バンド (`NB`)，および差分計算が行われる 3 次元実空間格子点 (`NL`) の添字を持つ．光電磁場の各格子点で電子軌道の時間発展計算が行われ，光電磁場自身の計算コスト，および光電磁場格子点間の通信コストは低い．

ARTED における並列計算上の大きな特徴は，比較的サイズの小さい実空間格子点ではなく，光電磁場格子，電子軌道の波数空間格子とバンド添字に対して MPI と OpenMP でのハイブリッド並列を適用している点である．光電磁場格子および波数空間に対し MPI で並列分散を行い，分散

```
integer,intent(in)     :: NL
integer,intent(in)     :: IDX(-4:4,0:NL-1)
integer,intent(in)     :: IDY(-4:4,0:NL-1)
integer,intent(in)     :: IDZ(-4:4,0:NL-1)
real(8),intent(in)     :: A,B(0:NL-1)
real(8),intent(in)     :: Cx(4),Cy(4),Cz(4)
real(8),intent(in)     :: Dx(4),Dy(4),Dz(4)
complex(8),intent(in) :: E(0:NL-1)
complex(8),intent(out):: F(0:NL-1)
complex(8),parameter   :: zI = (0.d0, 1.d0)
integer     :: i
complex(8) :: v(3),w(3)

do i=0,NL-1
  ! x computation
  v(1)=Cx(1)*(E(IDX(1,i))+E(IDX(-1,i)))&
  &    +Cx(2)*(E(IDX(2,i))+E(IDX(-2,i)))&
  &    +Cx(3)*(E(IDX(3,i))+E(IDX(-3,i)))&
  &    +Cx(4)*(E(IDX(4,i))+E(IDX(-4,i)))
  w(1)=Dx(1)*(E(IDX(1,i))-E(IDX(-1,i)))&
  &    +Dx(2)*(E(IDX(2,i))-E(IDX(-2,i)))&
  &    +Dx(3)*(E(IDX(3,i))-E(IDX(-3,i)))&
  &    +Dx(4)*(E(IDX(4,i))-E(IDX(-4,i)))
  ! y computation
  ...
  ! z computation
  ...
  ! store
  F(i) = B(i)*E(i) + A*E(i)      &
  &     - 0.5d0*(v(1)+v(2)+v(3)) &
  &     - zI*(w(1)+w(2)+w(3))
end do
```

図 3 ステンシル計算のオリジナル実装

した波数空間およびバンドに対し OpenMP で並列計算を行う．つまり，波数空間およびバンドはインデックスであり，このインデックスの数分，実空間の計算をシングルスレッドで行う．波数空間を MPI で分散しているため，一般的にステンシル計算で問題となるプロセス間での袖領域通信が不要な代わりに全実空間格子点の総和通信を行う必要がある．しかしながら，実空間格子は波数空間格子および軌道に対しそれほど大きくはないため通信コストが低く，大規模並列システム向けとなっている．元のターゲットシステムは京コンピュータ [7] で，我々は KNC クラスタに対して同アプリケーションの最適化を行っている [8]．

波数空間格子の規模は，シミュレーション対象や要求精度により異なるが，最終的には各実空間格子に対するステンシル計算の性能が全体性能を決定する．実空間格子のサイズは，シリコン結晶 (Si) を対象とした場合 $(16, 16, 16)$，α クォーツ (SiO_2) を対象とした場合 $(20, 36, 50)$ がアプリケーションの計算精度上の最小値となる．電子軌道配列は倍精度複素数で，周期境界条件による 25 点ステンシル計算が行われる．**図 2** に，25 点ステンシル計算のメモリアクセスパターンを示し，**図 3** に，ARTED のステンシル計算

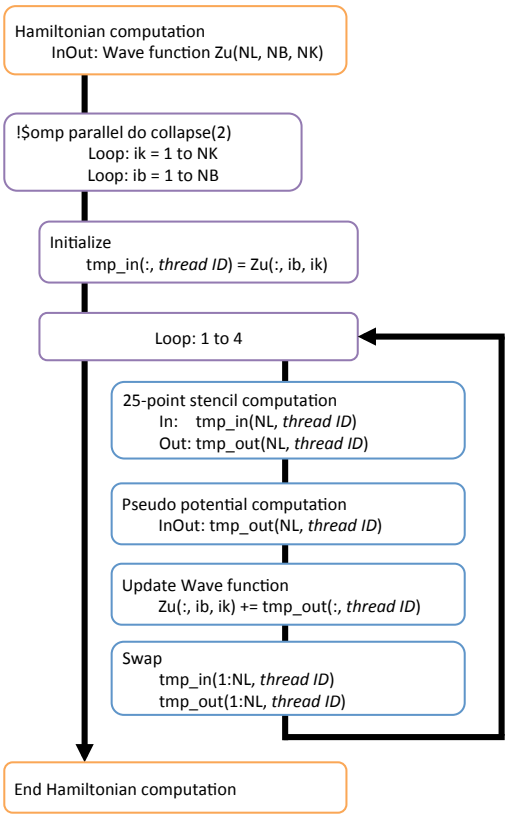

図 4 ハミルトニアン計算全体の流れ

のオリジナル実装を示す．ただし，Fortran では一般的に配列のインデックスが 1 で始まる 1-origin だが，最適化のため 0 始まりの 0-origin で受け取っている．Byte/FLOP 値は約 2.68 となり，メモリバンド幅律速な問題である．

電子軌道の時間発展計算に 4 次のテイラー展開法が用いられているため，1 ステップの時間発展に 4 回のハミルトニアン計算が必要となる．ハミルトニアン計算の流れについて，**図 4** に概略を示す．ハミルトニアン計算はステンシル計算と擬ポテンシャル計算で構成されている [5]．まず，各スレッドの一時計算領域 (`tmp_in`) に，計算する実空間をコピーする．コピーした後，ステンシル計算と擬ポテンシャル計算を行い，計算結果を格納した配列 (`tmp_out`) を用いて電子軌道配列の更新を行い，これを 4 回繰り返す．前述の通り，波数空間並列で，各実空間を逐次的に計算するため OpenMP を用いて波数空間およびバンドのループ両方を並列化する．1 個の実空間の計算は OpenMP の 1 スレッドで行われるため，各スレッドは 1 回の時間発展で 4 回のステンシル計算が含まれるハミルトニアン計算を逐次的に行い，複数個の実空間の計算を行う．各実空間は独立であり，1 回の時間発展で行われる 4 回のステンシル計算において OpenMP のスレッド同期または MPI による通信が発生しない．擬ポテンシャル計算はメモリバンド幅律速ではないため，実効メモリバンド幅に近い値を得るのは困難と予想される．しかしながら，ステンシル計算がハミル

表 1 SPARC64 VIIIfx の諸元

Core	8
L1 Data Cache/Core	32 KB
L2 Cache	6 MB (shared)
GFLOPS	128
Memory Bandwidth	64 GB/s
Byte/FLOP	0.5
Compiler	Fujitsu K 1.2.0-19
Optimize Option	`-O3 -Kfast,ocl,openmp`

トニアン計算の 8 割以上を占めており [8]，本研究ではステンシル計算に対し最適化を行う．

次に，図 3 のステンシル計算の詳細を説明する．A，B，C，D は全て係数で，実空間格子は E，F に格納され，E に入力データを格納しており，F に計算結果を書き込む．実空間格子のメモリ配置は，Z 次元が連続，Y 次元が Z-stride，X 次元が ZY-stride となっている．本研究のステンシル計算は，配列 v，w に係数が異なる近傍点の加減算結果が各次元で格納されており，2 つのベクトル演算が行われている．周期境界を考慮したインデックス計算を省略するため，間接参照配列 IDX，IDY，IDZ に予め計算した近傍点のインデックスを格納している．必要な演算を洗い出すと，複素数の加減乗算，複素数のスカラ倍が必要となり，演算数は格子 1 点の更新に対し 158 FLOP である．本研究では，性能評価にシリコン結晶 (Si) を用いる．Si は，1 個のステンシル計算領域である実空間格子のサイズ (NL=NLx×NLy×NLz) が (NLx, NLy, NLz)= (16, 16, 16) となる．MPI と OpenMP によって並列化される波数空間格子は，実アプリケーションでは 4^3 から 24^3 程度，バンドは 16 と設定するが，本研究では KNC のメモリ制限の関係でそれぞれ 8^3，16 と設定する．本研究の評価においては，サイズが 16^3 の逐次ステンシル計算を $8^3 \times 16$ 個と，非常に多くの個数を並列計算することが最も重要である．

本研究では，まず同コードが元のターゲットシステムである京コンピュータで十分な性能が得られているか検証する．その後 KNC に対し最適化を行い，性能評価と考察を行う．また，本研究中の全ての性能評価は 1 台のプロセッサ上で行う．

3. 京コンピュータ上の最適化

まず，図 3 の元実装が京コンピュータ上 (SPARC64 VIIIfx プロセッサ) で十分に最適化されているかを検証する [9]．**表 1** に，プロセッサの諸元を示す．同プロセッサでは，128-bit SIMD 命令の HPC-ACE が提供されているが本研究の最適化対象は倍精度複素数計算のため，1 要素しか計算できない．そのため，本研究では HPC-ACE を用いた手動ベクトル化は行わず，コンパイラによる自動ベクトル化 (Compiler Vectorization) を促進する最適化を行う．

```
real(8),    intent(in) :: B(0:NLz-1,0:NLy-1,0:NLx-1)
complex(8),intent(in) :: E(0:NLz-1,0:NLy-1,0:NLx-1)
complex(8),intent(out):: F(0:NLz-1,0:NLy-1,0:NLx-1)
integer,intent(in)    :: modx(0:NLx*2+4-1)
integer,intent(in)    :: mody(0:NLy*2+4-1)
integer,intent(in)    :: modz(0:NLz*2+4-1)
integer    :: ix,iy,iz
complex(8) :: v,w

#define IDX(dt) iz,iy,modx(ix+(dt)+NLx)
#define IDY(dt) iz,mody(iy+(dt)+NLy),ix
#define IDZ(dt) modz(iz+(dt)+NLz),iy,ix

do ix=0,NLx-1
do iy=0,NLy-1
do iz=0,NLz-1
  ! z computation
  v=(Cz(1)*(E(IDZ(1))+E(IDZ(-1))) &
  & +Cz(2)*(E(IDZ(2))+E(IDZ(-2))) &
  & +Cz(2)*(E(IDZ(3))+E(IDZ(-3))) &
  & +Cz(4)*(E(IDZ(4))+E(IDZ(-4))))
  ...
  ! y computation
  v=(Cy(1)*(E(IDY(1))+E(IDY(-1))) ... ) + v
  ...
  ! x computation
  ...
  ! store
  F(iz,iy,ix) = B(iz,iy,ix)*E(iz,iy,ix) &
  &           + A*E(iz,iy,ix) - 0.5d0*v - zI*w
end do
end do
end do
```

図 5 コンパイラによる自動ベクトル化に最適化したステンシル計算の実装

3.1 コードの問題点と修正

コンパイラによるベクトル化は，あくまでも記述されたコードを基に最適化を行うため，記述によってはベクトル演算が複雑化し期待した性能が得られていない可能性がある．図 3 の通り，各次元で長さ 4 のベクトル演算が計 6 回行われている．近傍点のアクセスに用いている 4 Byte 整数の間接参照配列 IDX，IDY，IDZ により，コンパイラはメモリアクセスパターンを把握できず，メモリアクセスが非常に非効率的となっている．また，24 個の近傍点にアクセスするため，メモリから 96×NL [B] のインデックス値のロードが必要となり，メモリ帯域を非常に圧迫してしまう．評価に用いる空間格子のサイズは 16^3 で，データサイズにすると 64KB となり，各点に対する係数 B も合わせて，各スレッドで 96 KB のデータをメモリからロードする．L1D キャッシュへのブロッキングを行うと，オーバーヘッドが大きく性能低下につながったため，SPARC64 VIIIfx での計算には適用していない．L2 キャッシュはコア間共有だが，プロセッサ全体で 6 MB，コアあたり 768 KB が利用できる．必要な係数なども含んでも，1 個のステンシル計

表 2 L1D キャッシュミスおよびメモリプリフェッチ発行数

	元実装	最適化実装	＋パディング
ミス率 [%]	7.7	4.23	2.02
プリフェッチ数	1.48×10^7	1.41×10^{11}	1.41×10^{11}
ロードストア数	1.13×10^{12}	1.09×10^{12}	1.09×10^{12}

表 3 100 反復時のキャッシュメモリアクセス待ち時間 [sec]

実装	浮動小数点数	整数	トータル
元実装	50.67	40.11	90.78
最適化実装	43.86	4.63	48.49
＋パディング	8.71	4.80	13.51

表 4 100 反復時の演算待ち時間 [sec]

実装	浮動小数点数	整数	トータル
元実装	9.03	0.57	9.60
最適化実装	8.05	1.55	9.60
＋パディング	5.81	1.48	7.29

表 5 SPARC64 VIIIfx でのステンシル計算性能

実装	GFLOPS	ピーク比 [%]
元実装	14.94	11.67
最適化実装	17.88	13.97
＋パディング	27.20	21.25

算で必要なデータは L2 キャッシュに収まっている．図 2 に示す通り，メモリは Z 次元が連続となっているが，元実装では最もメモリ距離が遠い X 次元から計算しており，キャッシュの有効利用が行えていない可能性がある．そこで，Z, Y, X とキャッシュを利用しやすいように計算順序を変更した．

以上より，ステンシル計算の最適化を行ったコードを**図 5** に示す．1 次元配列として確保している 3 次元実空間格子を，カーネル中では (NLx, NLy, NLz) の 3 次元配列とした．この変更で，ステンシル計算時にインデックス計算が簡素化されるため，コンパイラがメモリアクセスパターンを把握しやすくなることが期待される．

本研究のステンシル計算は周期境界を持つため，インデックス計算には剰余が必要となる．実空間の各次元のサイズは，本研究では 2 のべき乗の 16 のため，論理積演算で代替可能かつ剰余演算よりも高速に求められるが，被除数が 2 のべき乗である必要がある．格子点を増やして 2 のべき乗にすることも可能だが，計算領域が肥大化しシミュレーションに制限が発生する可能性がある．そこで，各次元で剰余テーブル modx，mody，modz を用意し，剰余計算を省略した．剰余テーブルは，次元のサイズを N とすると $[0, N \times 2 + 4 - 1]$ の範囲の剰余計算結果が保存されている．本研究では，剰余テーブルは各次元で 144 Byte が必要で，これは全てのインデックスを保管している間接参照配列に対し非常に小さく，性能と可用性のバランスが取れている．

最後に，計算領域のサイズが 2 のべき乗のため，キャッシュスラッシングが多発する可能性がある．したがって，Y 次元についてパディングを行い，キャッシュスラッシングを回避する．

3.2 最適化の効果

各最適化について，京コンピュータで提供されている詳細プロファイラを用い，8 スレッド (コア) 並列実行で評価を行った．ハミルトニアン計算には擬ポテンシャル計算が含まれるが，ステンシル計算に対し実行時間は比較的小さいため，本評価では計算を省略する．つまり，図 4 中で，擬ポテンシャル計算を省略したハミルトニアン計算を 100 反復行い，キャッシュミスや演算時間などの検証を行う．

まず，元実装でのロードストア命令に対するキャッシュミスを計測すると，L1D ミスが 7.7 %，L2 ミスが 0.03 %となった．予測通り，各実空間格子は L2 キャッシュに収まっており，L1D キャッシュミスの削減が重要となっている．ロードストア命令比での L1D キャッシュミスを**表 2** に示す．ここでは，図 3 の元実装，図 5 の最適化実装，最適化実装に対しパディングを行った際の結果を示している．最適化実装は元実装に対し，ロードストア命令数と L1D キャッシュミスを削減できている．また，プリフェッチ命令の発行数が 10^7 オーダから 10^{11} オーダに増加しており，間接参照配列の除去によってプリフェッチが効果的に動作している．次に，キャッシュメモリへのアクセス待ち時間について，**表 3** に示す．元実装では，100 反復時に浮動小数点数と整数のアクセス待ち時間が合わせて 90 秒発生していた．最適化実装では，間接参照配列を用いず近傍点のインデックスを直接計算することによって整数のアクセス待ち時間が約 1/10 まで削減された．浮動小数点数は最適化実装でも依然として待ち時間が非常に長いが，パディングによって 1/5 まで削減されている．ここで，演算待ち時間を**表 4** に示すが，インデックスを直接計算することによって，整数演算の待ち時間が増加している．しかしながら，キャッシュメモリのアクセス待ち時間と合わせて 100 秒近いボトルネックが約 20 秒まで削減されており，間接参照のコストが非常高いということが分かる．計算順序を非連続方向から計算した場合，ロードストア命令の発行数が 1.25 倍に増加し，キャッシュメモリのアクセス待ち時間が 2 倍，演算待ち時間が 5 倍近く増加した．非連続方向から計算したことにより，コンパイラは計算をパイプライン化できないという警告を出力しており，効率的なベクトル化および演算ができていなかったと考えられる．

最後に，演算性能について**表 5** に示す．元実装では 14.94 GFLOPS だったが，最適化およびパディングを行うことで 27.20 GFLOPS，ピーク性能比約 21 %まで改善した．本研究では，図 5 の最適化した自動ベクトル化実装を用いて，KNC への最適化を行う．

表 6 COMA に搭載されている Knights Corner の諸元

Core	60 × 4 Thread
L1 Data Cache/Core	32 KB
L2 Cache	512 KB / Core
GFLOPS	1074
Memory Bandwidth	352 GB/s
Byte/FLOP	0.3277
Compiler	Intel 15.0.2
Optimize Option	`-O3 -openmp -restrict -ansi-alias -fno-alias -opt-assume-safe-padding`

表 7 Knights Corner での自動ベクトル化性能

実装	GFLOPS	ピーク比 [%]
元実装	30.06	2.80
自動ベクトル化実装	109.17	10.16
近傍点ロード最適化	125.81	11.71
non-temporal store	129.00	12.01
L1D ブロッキング	129.48	12.06
メモリパディング	130.44	12.15

表 8 倍精度浮動小数点数の `Load`/`Gather` 命令数

実装	`Load`	`Gather`
自動ベクトル化実装	568	96
近傍点ロード最適化	456	64

表 9 L1D キャッシュミス数とヒット率

実装	ミス数	ヒット率 [%]
元実装	1.67×10^{11}	88.7
自動ベクトル化実装	7.83×10^{10}	85.8
近傍点ロード最適化	5.21×10^{10}	88.5
non-temporal store	4.76×10^{10}	89.6
L1D ブロッキング	4.79×10^{10}	89.0
メモリパディング	4.55×10^{10}	89.3

4. Knights Corner への実装

4.1 自動ベクトル化コードの最適化

前述の最適化実装を用いて，KNC へのステンシル計算の最適化を行う．[10] や [11] といった先行研究を参照すると，高レベル最適化を行ったとしてもピーク性能比 10 %と，KNC において高い演算性能を得るのは非常に困難が予測される．本研究では，筑波大学計算科学研究センターの KNC クラスタ COMA を用いて KNC への最適化を行った [12]．KNC の諸元について，**表 6** に示す．まず図 5 のコードで，元実装に対し KNC にて性能が得られていることを確認する．KNC でのメモリのアラインメントは，実空間格子配列 `E(NLz, NLy, NLx)` としたとき，`E(1, :, :)` が 64 バイト境界にアラインされるように設定した．**表 7** に，60 コア 240 スレッドで実行したときの KNC でのステンシル計算性能を示す．本研究では，KNC は 1 コアあたり 4 スレッド，計 240 スレッドで全ての性能評価を行う．図 3 の元実装は，ピーク性能比で 2.8 %しか得られておらず，KNC でも間接参照のコストが非常に高い．自動ベクトル化実装は，図 5 を KNC で実行したときの性能で，元実装の約 3.6 倍に相当する 109.17 GFLOPS が得られた．同実装は，SPARC64 VIIIfx に対して最適化を行ったものだがシングルコアレベルの最適化のため，KNC においても有効と考えられる．この実装に対し，KNC へのさらなる最適化を行う．

図 5 では，近傍点のロードがベクトル演算と同時に行われているため，一時配列に近傍点をロードした後に，ベクトル演算を行うように変更した．アセンブラコードから，ロードおよび `Gather` 命令の出現数を**表 8** に示す．KNC では，メモリのアラインが取られていない状態でのロードに対して `loadunpackhi/lo` という 2 つの命令が用いられる．そのため，表での `Load` 命令数はアラインされたロード命令と合わせて 3 命令の総数を示している．アセンブラコードレベルでは，命令出現数が減少しており，処理内容が簡素化され効率が良くなったかロード回数が減少したと考えられ，125.81 GFLOPS に性能が向上した．SPARC64 VIIIfx では，性能低下に繋がるため適用していない．

`F` には書き込みのみが行われ，書き込んだデータは計算中で利用されない．そこで，最内の Z 次元ループに `vector nontemporal(F)` ディレクティブを記述し，キャッシュを経由しない non-temporal store [13] とすることで，キャッシュメモリをさらに有効利用し 129 GFLOPS となった．SPARC64 VIIIfx では効果がなかったが，L1D に対して YX 次元に 2 次元ブロッキングを行うと，若干ながら性能が改善し 129.48 GFLOPS となった．スラッシングを回避するために SPARC64 VIIIfx と同様に Y 次元に対しメモリパディングを行ったが 130.44 GFLOPS と小さな改善にとどまった．Z 次元のパディングを行った場合，KNC ではベクトル化時にループサイズがベクトル長で割り切れなくなってしまい，アドレスが 64 Byte 境界からずれてしまうため，効率的なベクトル化が困難となる．

L1D キャッシュミス数とヒット率について，Intel VTune プロファイラ [14] での計測結果を**表 9** に示す．元実装に対し，自動ベクトル化実装はミス数のオーダが 1 桁下がっている．近傍点のインデックスを直接計算したため，間接参照配列に使用していたキャッシュメモリを全て近傍点に利用できたことが要因と考えられる．近傍点ロード最適化の効果が最も高く，L1D キャッシュミスが 2×10^{10} 程度減少した．他の最適化も適用することでミス数が減少しているが，L1D ブロッキング時には増加しており，効果が非常に小さいと考えられる．メモリパディングも行い，最終的な L1D キャッシュヒット率は 89.3 %となった．

KNC では，ソフトウェアプリフェッチの適切な挿入が性能に大きく影響する可能性がある [13]．本研究では，ソフトウェアプリフェッチを `prefetch` ディレクティブやコ

```
inline __m512d dcomplex_mul_c(__m512d a) {
  __m512i b, c, d = (__m512i)a;
  b = _mm512_set4_epi64(1LL<<63,0,1LL<<63,0);
  c = _mm512_shuffle_epi32(d,_MM_PERM_BADC);
  return (__m512d) _mm512_xor_si512(c,b);
}
```

図 6 倍精度複素数積 $(a, bi)(0, -i)$ を展開した実装

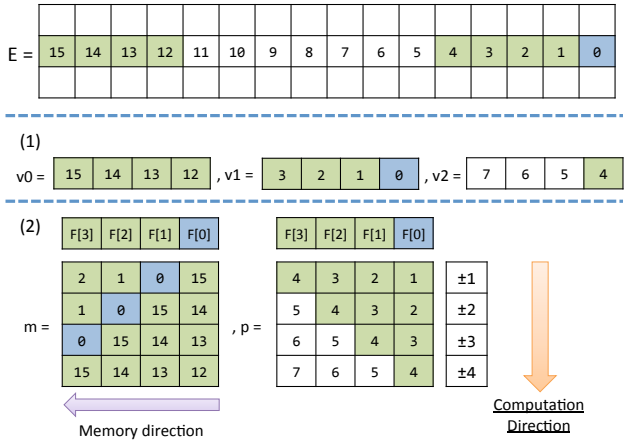

図 7 ベクトル演算での Z 次元のメモリアクセス

ンパイラオプション，手動ベクトル化実装では Intrinsics を用いて挿入し性能評価を行ったが，手動挿入による優位な結果は得られていない．

4.2 手動ベクトル化コードの実装と最適化

次に，KNC で提供されている 512-bit SIMD 命令の Initial Many-Core Instructions (IMCI) を用いた手動でのベクトル化 (Explicit Vectorization) を行う．本研究では，自動ベクトル化実装の最適化から，次に示す方法で手動ベクトル化を行う．まず，F に対して non-temporal store を用いるが，ストア先のアドレスが 64 Byte でアラインされていなければならない．演算ベクトル長は倍精度複素数のため 4 となり，4 つの格子点を同時更新する．また，各次元で近傍点を先にロードし，その後演算を行う．前提条件として，連続なメモリアクセスとなる Z 次元について格子サイズを 4 の倍数 (ベクトル長) とし，最適化を行った．本研究では，主に以下の 3 点の最適化を行い，それぞれの効果について検証する．

- 倍精度複素数積の最適化
- 非アラインメモリアクセスの最適化
- 近傍点のインデックス計算のベクトル化

本研究の計算では，複素数が用いられているが，IMCI では Intel SSE/AVX と異なり，複素数積を高速に計算するための命令が提供されていない [15]．KNC で複素数積を行う際には，masked 命令を使って実部あるいは虚部のみを計算する必要があり，ベクトル演算ユニットの潜在性能の半分しか利用できない．これは Intel コンパイラによる自動ベクトル化，Intrinsics を用いた手動ベクトル化の両方で問題となる．この問題から，KNC の複素数演算で高い性能を得るのは実数演算の場合よりも困難と考えられる．図 3 では，配列 w と定数 zI で積が必要となるが，定数積のため，式展開により演算を省略できる．複素数積 $(a, bi)(0, -i)$ を展開すると $(b, -ai)$ となり，(1) 実部と虚部を入れ替える，(2) 虚部の符号を反転させる，の 2 ステップで計算可能となる．**図 6** に，式展開を行った IMCI 実装を示す．b は定数として扱われ，XOR を用いて虚部の実数の符号ビットを反転する．

```
/* Stencil computation loop */
for(ix = 0 ; ix < NLx ; ++ix) {
for(iy = 0 ; iy < NLy ; ++iy) {
double complex const* e = E[ix*NLy*NLz + iy*NLz];
...
for(iz = 0 ; iz < NLz ; iz += 4) {
  __m512i v0, v1, v2;
  __m512d m[4], p[4];

  /* (1) */
  v0 = _mm512_load_epi64(e + modz[iz-4+NLz]);
  v1 = _mm512_load_epi64(e + iz);
  v2 = _mm512_load_epi64(e + modz[iz+4+NLz]);

  /* (2) */
  m[0] = (__m512d)_mm512_alignr_epi32(v1,v0,12);
  m[1] = (__m512d)_mm512_alignr_epi32(v1,v0, 8);
  m[2] = (__m512d)_mm512_alignr_epi32(v1,v0, 4);
  m[3] = (__m512d)v0;
  p[0] = (__m512d)_mm512_alignr_epi32(v2,v1, 4);
  p[1] = (__m512d)_mm512_alignr_epi32(v2,v1, 8);
  p[2] = (__m512d)_mm512_alignr_epi32(v2,v1,12);
  p[3] = (__m512d)v2;

  /* Z-dir. computation */
  for(n = 0 ; n < 4 ; ++n) {
    __m512d a = _mm512_add_pd(p[n], m[n]);
    __m512d b = _mm512_sub_pd(p[n], m[n]);
    v = _mm512_fmadd_pd(C[n], a, v);
    w = _mm512_fmadd_pd(D[n], b, w);
  }
  /* Y-dir. computation */
  /* X-dir. computation */
  /* non-temporal store to F */
} } }
```

図 8 Z 次元のメモリアクセスを Intrinsics で実装したコード

連続方向である Z 次元の計算では，必ず非アラインメモリアクセスが発生するため，コンパイラは Gather 命令を発行しデータを集める．しかしながら，同命令はレイテンシが高く，性能上のボトルネックとなっている可能性が高い [16]．そこで，本研究では Alignr 命令を用いて Gather 命令を用いずに Z 次元の近傍点を集める．Alignr 命令は 2 つの 512-bit ベクトルを連結して 1024-bit ベクトルとし，32-bit 単位で右論理シフトを行った後，下位 512-bit を返

す命令である．

ここで，メモリの連続方向と非連続方向での演算パターンの違いについて考える．連続方向では，Alignr 命令を用いて 1 点の更新に必要な近傍点を 1 つのベクトルに集めることが可能である．しかし，非連続方向では 1 回の Load 命令を用いて 4 点の更新に必要な近傍点のうち，±4 のいずれか 1 点を一度に取得できる．この場合，非連続方向はメモリに対して垂直方向に演算すれば，4 点を同時更新することができるが，連続方向では水平方向に演算する必要がある．SIMD での水平方向演算は IMCI には実装されておらず，AVX および SSE では加算が実装されているが演算コストが高い．そのため，連続方向でのメモリアクセスを，非連続方向と同じ垂直方向の演算となるように最適化を行う．

連続方向の Z 次元の非アラインメモリアクセスを，非連続方向と同じ演算パターンとなるように実装したイメージが**図 7** となる．(1) まず，Load 命令を用いて更新対象の前の 4 点，更新対象の 4 点，その次の 4 点をそれぞれベクトル v0，v1，v2 としてロードし，(2) Alignr 命令と v0，v1，v2 を用いて，各点の更新に必要な近傍点ベクトルを生成する．この時，更新点 ± 4 でそれぞれ 1 つの行ベクトルを構成する．そのため，負方向と正方向の近傍点で 4×4 の正方行列 m および p が生成される．例えば，更新点 F[0] では，近傍点 15–12 および 1–4 が必要となる．生成した正方行列を見ると，非連続方向でのメモリパターンと同様に，必要な近傍点は 1 列に揃っており，ベクトル演算はメモリに対して垂直方向に演算を行うだけでよく，水平方向への演算が必要ない．このメモリアクセスを，Intrinsics を用いて実装すると**図 8** となる．Alignr 命令の 32-bit シフト回数は即値で，コンパイル時には値が確定している必要がある．

近傍点は，各方向に ± 4 点あり，各次元で 8 点，計 24 点となる．Z 次元は前述の最適化を行っているため，Y および X 次元の 16 点のインデックスを求める必要がある．インデックスは 4 Byte 整数の表現可能範囲で十分足りるため，16 点のインデックスを 512-bit ベクトル演算でまとめて求めることができる．本研究では，Y および X 次元の近傍点のインデックス計算をまとめて一度ベクトル演算で求め，スカラでのインデックス計算を極力少なくした．Z 次元の計算では，前述の最適化でメモリから 512-bit ベクトル 3 本のみをロードするため，図 8 に示すようにスカラでインデックス計算を行っている．

手動ベクトル化実装に対し，各最適化を個別に適用した結果を**表 10** に示す．単純に手動ベクトル化を行った初期実装では 113.65 GFLOPS となり，自動ベクトル化よりも性能が低い．L1D キャッシュミスは全ての最適化で約 3.75×10^{10}，ヒット率は 92 %程度となり，自動ベクトル化実装に対し良好な結果となっている．

表 10 手動ベクトル化コードの最適化の効果

実装	GFLOPS	ピーク比 [%]
初期実装	113.65	10.58
複素数積最適化	114.89	10.70
非アラインアクセス最適化 (A)	179.39	16.70
インデックス計算ベクトル化 (B)	142.74	13.29
(A) + (B)	221.38	20.61
全最適化適用時	224.45	20.90

表 11 各プロセッサでのステンシル計算性能

Processor	Vectorize	GFLOPS	ピーク比 [%]
KNC 7110P	Explicit	224.45	20.90
Ivy-B E5-2670v2	Explicit	114.65	57.32
Haswell E5-2670v3	Compiler	170.74	44.46
SPARC64 VIIIfx	Compiler	27.20	21.25

初期実装に対し複素数積展開を行うと，114.89 GFLOPS と若干ながら性能が向上し，複素数積を計算するコストが非常に小さいといえる．本研究では，各点の更新に対し複素数積を一度しか行わず，演算数が少ないためで，複素数積が非常に多く必要となる計算では注意が必要となる．非アラインメモリアクセス最適化は，179.39 GFLOPS と本研究の最適化の中で最も高い効果が得られ，Gather 命令が大きなボトルネックとなっていることが推測される．インデックス計算をベクトル化することで 142.74 GFLOPS の性能が得られ，インデックスをベクトル演算で求める利点があることが分かった．さらに，非アラインメモリアクセスの最適化と，インデックス計算のベクトル化の 2 つを組み合わせ，221.38 GFLOPS を達成した．初期実装に対し約 107 GFLOPS の性能向上が得られており，2 つの性能の増分が約 95 GFLOPS で，10 GFLOPS ほど追加の性能向上が得られている．命令数は変わっておらず，組み合わせによってレジスタの有効利用や計算の待ち時間が減少しさらなる性能向上に繋がったと考えられる．3 つの最適化全てを適用することで，初期実装に対し 1.97 倍の 224.45 GFLOPS とピーク性能比 20.9 %を達成した．

5. 考察

5.1 各プロセッサでの性能

KNC の性能の妥当性について，Xeon CPU との比較を行い検討する．本研究では，Ivy-Bridge と Haswell プロセッサを CPU として取り上げ，Intel コンパイラ 16.0.0 を用いて計測した．各プロセッサでの性能を**表 11** に示す．Ivy-Bridge では，KNC での実装を基に AVX の手動ベクトル化実装を行ったが，本研究ではその詳細は省略する．Haswell の場合，手動ベクトル化実装に対し良好な結果となったため自動ベクトル化実装の性能を記載している．KNC がピーク性能比約 20 %程度であるのに対して，Ivy-Bridge と Haswell ではピーク性能比がそれぞれ 57 %，44 %と高い実効性能を達成している．性能差がキャッシュ

メモリ性能に大きく影響されているのではないかという仮定の元に，以下，考察を行う．

まず，ステンシル計算領域である 1 個の実空間のサイズは 16^3 より 64 KB である．L1D キャッシュサイズは全てのプロセッサで 32 KB であり，これは実空間の半分のサイズであり入り切らないため，L2 キャッシュから L1D キャッシュへのバンド幅が重要と考えられる．ステンシル計算では，各点に対する係数が 32 KB 必要で，書き込みは non-temporal store を行うため 96 KB 程度が 1 回の計算で必要となる．しかしながら，スレッド当たりの L2 キャッシュサイズが最小の KNC であっても，そのサイズは 128 KB/スレッドであるため，ステンシル計算に必要なデータはほぼ L2 に収まっている．計算領域である電子軌道関数のサイズは，本研究では波数空間格子とバンドをそれぞれ 8^3, 16 としたので，512 MB となり，Ivy-Bridge と Haswell が持つ L3 キャッシュ 25–30 MB にも収まらない．しかしながら，ステンシル計算のサイズは 64 KB で，L3 キャッシュには 400–480 個の実空間を載せることが可能である．本研究のステンシル計算では，計算の前処理として電子軌道配列から計算用の一時配列配列に線形コピーを行う．線形コピーは非常に単純かつ連続なメモリアクセスとなるため，1 つの実空間に対しハミルトニアンを計算している間に，次に計算する実空間がプリフェッチされ，L3 キャッシュに既にロードされていることは十分に考えられる．そのため，L3 キャッシュを持たない KNC および SPARC64 VIIIfx では，DRAM から L2 キャッシュへのバンド幅に律速され，Ivy-Bridge と Haswell では，L3 キャッシュから L2 キャッシュへのバンド幅に律速されると考えられる．

電子軌道配列の更新で利用するならば，ステンシル計算での non-temporal store の利用は，キャッシュ利用の妨げとも考えられる．しかしながら，通常のストア命令を用いるとステンシル計算中に入力配列が L2 キャッシュから追い出される可能性があり，数 10 GFLOPS 程度の性能低下が起こる．電子軌道配列の更新は線形加算のため，ステンシル計算のように複雑なキャッシュ利用がない．non-temporal store により，ペナルティが比較的低い電子軌道配列の更新中にキャッシュミスのタイミングがずれることで性能向上に繋がったと考えられる．

5.2 Knights Landing など他のメニーコアプロセッサへの適用

2016 年内には，米国の国立エネルギー研究科学計算センター (NERSC) 等にて KNL を搭載した大規模システムが稼働する予定となっている．また，京コンピュータの後継として現在 FX100 システムが名古屋大学や核融合研究所にて稼働しており，同システムは合計 34 コアを持つ SPARC64 XIfx プロセッサを搭載している．

KNL では，KNC とは異なり AVX-512 が SIMD 命令として提供されるため，本研究で実装した手動ベクトル化コードの実装を修正する必要がある．しかしながら，IMCI と AVX-512 には不完全ではあるが下位互換性があり，ほとんどの命令は同じフォーマットで利用できる．本研究の実装では，四則演算命令を除外すると 13 命令を使用しており，そのうち 3 命令が IMCI と AVX-512 でフォーマットが異なる．修正は，単純な命令置換，または数行のインライン関数レベルでの置換にとどまっており，修正コストは非常に小さい．既に，現在の IMCI 実装に対し修正を行い，Intel のエミュレータ [17] 上で動作することを確認している．また，AVX-512 では機能ごとにサブセットが定義されているが，本研究の実装では AVX-512 対応の全プロセッサで提供される AVX-512F のみが必要で，AVX-512 対応のプロセッサがあれば性能評価が可能である．

SPARC64 XIfx は，32 個の演算コアと 2 個のアシスタントコアが 1 つのチップに搭載されているメニーコアプロセッサである [18]．同プロセッサでは，L1D キャッシュサイズが 2 倍の 64 KB となり，バンド幅も大幅に改善され，単純な移植でもある程度の性能向上が期待される．また，新しい SIMD 命令として HPC-ACE2 を提供し，AVX と同じ SIMD 長の 256-bit SIMD 演算が可能となっている．今回，IMCI 実装の AVX への変換は省略したが，HPC-ACE2 を用いることで手動ベクトル化実装と高速化が可能と考えている．

6. 結論

本研究では，電子動力学シミュレーションにて現れる倍精度複素数の 25 点ステンシル計算について，KNC への最適化を行った．まず，アプリケーションの元のターゲットシステムである京コンピュータのプロセッサ SPARC64 VIIIfx にて，コンパイラによる自動ベクトル化を促進し 14.94 GFLOPS から 27.20 GFLOPS に性能が改善した．同実装を，KNC 上で更に最適化し，30.06 GFLOPS から 130.44 GFLOPS に性能が向上した．更なる最適化で，KNC で利用できる SIMD 命令の IMCI を用いて手動ベクトル化を行った．手動ベクトル化実装では複素数積の展開，非アラインされたメモリアクセスの最適化，インデックス計算のベクトル化を行うことで 224.45 GFLOPS，ピーク演算性能比で 20.9 %を達成した．また，Xeon プロセッサとの比較を行い，Haswell プロセッサに対し 1.3 倍の性能が得られていることを確認した．Haswell や Ivy-Bridge で高いピーク性能比が得られているのは，L3 キャッシュによる恩恵と考えられる．

最後に，Knights Landing および SPARC64 XIfx への適用について考察した．今後，これらのプロセッサへの適用および性能評価を行うことを予定している．本研究で実装した最適化コードは COMA および京コンピュータに対する最適化実装として，オープンソースソフトウェアとして

の公開を予定している．

謝辞 本研究の評価環境は，筑波大学計算科学研究センター平成 28 年度学際共同利用プログラム課題「時間依存密度汎関数理論によるパルス光と物質の相互作用」，HPCI 平成 28 年度「京」一般利用課題「極限的パルス光と物質の相互作用を記述するマルチスケール第一原理計算」，文部科学省ポスト「京」重点課題 (7)「次世代の産業を支える新機能デバイス・高性能材料の創成」による．本研究の一部は JST-CREST 研究領域「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」，研究課題「ポストペタスケール時代に向けた演算加速機構・通信機構統合環境の研究開発」による．本研究の最適化にあたり，京都大学学術情報メディアセンターの中島浩教授に多くのアドバイスを頂きました．深く感謝申し上げます．

参考文献

[1] TOP500: http://www.top500.org/.
[2] S. Heybrock, B. Joó, D. D. Kalamkar, M. Smelyanskiy, K. Vaidyanathan, T. Wettig and P. Dubey: Lattice QCD with Domain Decomposition on Intel Xeon Phi Co-processors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, IEEE, (online), DOI: 10.1109/SC.2014.11 (2014).
[3] E. Aprà, M. Klemm and K. Kowalski: Efficient Implementation of Many-body Quantum Chemical Methods on the Intel Xeon Phi Coprocessor, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, (online), DOI: 10.1109/SC.2014.60 (2014).
[4] H. Nakashima: Manycore challenge in particle-in-cell simulation: How to exploit 1 TFlops peak performance for simulation codes with irregular computation, *Computers & Electrical Engineering*, Vol. 46, pp. 81–94 (online), DOI: 10.1016/j.compeleceng.2015.03.010 (2015).
[5] S. A. Sato and K. Yabana: Maxwell + TDDFT multiscale simulation for laser-matter interactions, *J. Adv. Simulat. Sci. Eng.*, Vol. 1, No. 1, pp. 98–110 (2014).
[6] M. Schultze, K. Ramasesha, C. D. Pemmaraju, S. A. Sato, D. Whitmore, A. Gandman, J. S. Prell, L. J. Borja, D. Prendergast, K. Yabana, D. M. Neumark and S. R. Leone: Attosecond band-gap dynamics in silicon, *Science*, Vol. 346, No. 6215, pp. 1348–1352 (online), DOI: 10.1126/science.1260311 (2014).
[7] RIKEN AICS: K Computer, http://www.aics.riken.jp/en/k-computer/.
[8] 廣川 祐太，朴 泰祐，佐藤 駿丞，矢花 一浩：Xeon Phi クラスタにおける Symmetric 並列実行による電子動力学シミュレーションの性能評価，情報処理学会研究報告，Vol. 2015-HPC-151, No. 18 (2015).
[9] T. Maruyama: SPARC64(TM) VIIIfx: Fujitsu's New Generation Octo Core Processor for PETA Scale Computing, *HotChips 21* (2009).
[10] 松田 元彦，丸山 直也，滝沢 真一郎：Xeon Phi (Knights Corner) の性能特性とステンシル計算の評価，情報処理学会研究報告， Vol. 2014-HPC-143, No. 32 (2014).
[11] 伊奈 拓也，朝比 祐一，井戸村泰宏：テラフロップス級メニーコアアーキテクチャにおけるステンシル計算の最適化手法の開発，情報処理学会研究報告，Vol. 2015-HPC-152, No. 10 (2015).
[12] 筑波大学計算科学研究センター：COMA, http://www.ccs.tsukuba.ac.jp/files/coma-general/coma_outline.pdf.
[13] R. Krishnaiyer, E. Kultursay, P. Chawla, S. Preis, A. Zvezdin and H. Saito: Compiler-Based Data Prefetching and Streaming Non-temporal Store Generation for the Intel(R) Xeon Phi(TM) Coprocessor, *IPDPSW 2013*, pp. 1575–1586 (2013).
[14] Intel: VTune Amplifier XE, https://software.intel.com/en-us/intel-vtune-amplifier-xe.
[15] D. Takahashi: Implementation and Evaluation of Parallel FFT Using SIMD Instructions on Multi-core Processors, *IWIA 2007*, pp. 53–59 (2007).
[16] J. Hofmann, J. Treibig, G. Hager and G. Wellein: Comparing the Performance of Different x86 SIMD Instruction Sets for a Medical Imaging Application on Modern Multi- and Manycore Chips, *WPMVP '14*, pp. 57–64 (2014).
[17] Intel Software Development Emulator: https://software.intel.com/en-us/articles/intel-software-development-emulator.
[18] T. Yoshida and *et al.*: Sparc64 XIfx: Fujitsu's Next-Generation Processor for High-Performance Computing, *HotChips 25* (2014).