Vol.2018-HPC-163 No.23
2018/3/2

電子動力学シミュレーションコードのメニーコアプロセッサとGPUにおける性能比較

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

概要：これまで，我々は電子動力学シミュレータ “ARTED” についてメニーコアプロセッサ向けの最適化・性能評価を行い，Intel Knights Landing (KNL) にて十分な性能が得られたことを報告している．本研究では，アクセラレータとして広く用いられている NVIDIA Tesla P100, V100 GPU での性能評価を行い，Intel Xeon Phi 7250 (KNL) と，Xeon Gold 6148 (Skylake-SP) も含めて性能比較を行った．V100 GPU は，全体性能について P100 と KNL に対し 2 倍の性能を達成し，アプリケーションの支配的な計算である 25 点ステンシルは KNL に対し 3 倍近くの性能を達成した．また，V100+OpenACC がステンシル計算で約 25%の実行効率を達成し，V100 は OpenACC を用いたディレクティブベースの GPU アプリケーション開発でも CUDA と遜色ない十分な性能が期待される．

1. はじめに

これまで，我々は主に「京」コンピュータを対象に開発してきた電子動力学シミュレータ “ARTED” を，Intel 社の Xeon Phi を対象として最適化を行ってきた [1]．昨年には，筑波大学と東京大学が共同設置する Joint Center for Advanced HPC (JCAHPC: 最先端共同 HPC 基盤施設) にて稼働しているピーク性能 25 PFLOPS の KNL システムである “Oakforest-PACS” の全系を用いた大規模シミュレーションによる性能評価を報告した [2]．現在，理化学研究所計算科学研究機構 (AICS) を中心に開発が進められているポスト「京」コンピュータでも汎用メニーコアプロセッサを用いることが公表されており，Oakforest-PACS は同システムへの準備環境としても注目されている．我々が最適化したアプリケーションは，メニーコアプロセッサに対して十分に最適化されており，ポスト「京」コンピュータでの高い性能も十分に期待できる．

現在，我々は ARTED と同分野でシミュレーション目標が異なるシミュレータ “GCEED” [3] と ARTED をベースに，広く光科学分野への貢献を目指し光科学シミュレータ “SALMON (Scalable Ab-initio Light-Matter simulator for Optics and Nanoscience)” を開発している [4]．我々が ARTED で行ってきたメニーコアプロセッサへの最適化は，SALMON に取り込まれ，現在は GCEED が対象としてきたシミュレーションへの適用が実験されている．SALMON は光科学という非常に応用範囲の広い分野を対象としており，大規模スーパーコンピュータシステムのみならず，中小規模の PC クラスタなどでも利用されることを目指し開発していかなければならない．

今日では，アクセラレータ型のスーパーコンピュータとして PEZY-SC2 を搭載した JAMSTEC の暁光や，Pascal GPU を搭載した東京工業大学の TSUBAME 3.0 などが稼働し [5]，本年には産業技術総合研究所にて最新の Volta GPU を搭載したクラスタが導入予定となっている [6]．我々がターゲットとするアプリケーションは前述の通りメニーコアプロセッサに対し十分な最適化を行えていると言えるが，実際に運用されている HPC システムとターゲットアプリケーションの開発目的を考えれば，アクセラレータを搭載したシステムへの対応も非常に重要である．本研究では，アクセラレータとして最も広く用いられている NVIDIA GPU を対象として，アプリケーションの性能評価を実施する．

NVIDIA が展開する Pascal や Volta GPU は，KNL が持つ MCDRAM よりも高いメモリバンド幅を持つ HBM2 を有し，メモリバンド幅律速なアプリケーションに対して有利である [7]．しかしながら，一般的な PC クラスタでは CPU と GPU は転送性能が高々 32 GB/s しかない PCIe Gen3 で接続され，CPU-GPU 間のデータ転送がボトルネックとなっている．また，CUDA はプログラマブルであるがアプリケーションコードの複雑性を増加させ，計算科学者

[1] 筑波大学大学院 システム情報工学研究科
[2] 筑波大学 計算科学研究センター
[3] Max Planck Institute for the Structure and Dynamics of Matter
[a)] hirokawa@hpcs.cs.tsukuba.ac.jp

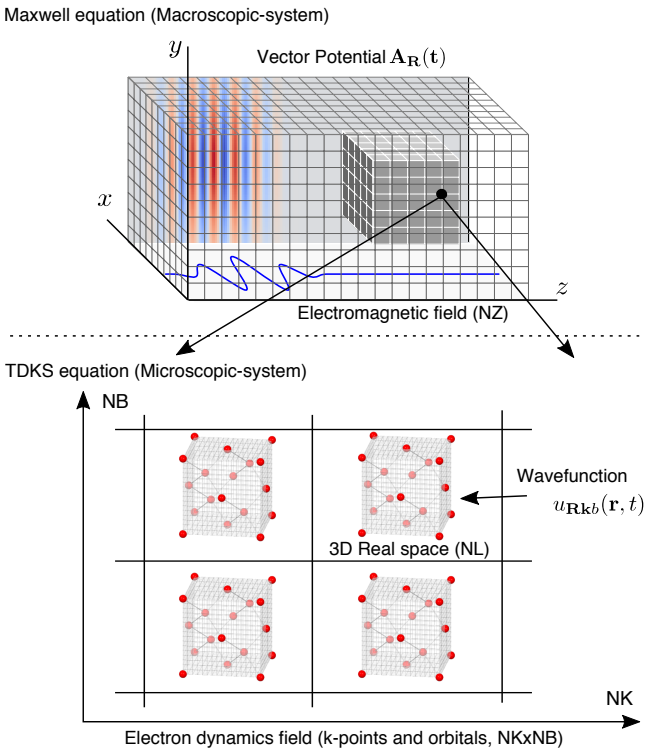

図 1　シミュレーション対象

にとってはコードの改修や管理の上で大きな障害となっている．そこで，現在では OpenACC がアクセラレータに対するディレクティブベース言語として認知されており [8]，既に我々のアプリケーションも OpenACC を用いて GPU への実装が行われている．

以上の背景の下，本研究では，我々が特に Xeon Phi に向けて最適化を行っている電子動力学アプリケーションの性能について，GPU クラスタで評価を行い，GPU とメニーコアプロセッサの性能比較を行う．はじめに，2 章で SALMON の並列化方法・最適化ターゲットについて述べたあと，これまでの研究として Xeon Phi で行ってきた最適化の概略を述べる．3 章では GPU での実装と最適化について述べ，4 章で Tesla P100/V100 GPU での性能評価と Xeon Phi (KNL) と Xeon CPU (Skylake-SP) との性能比較を行う．最後に，5 章で性能の妥当性について考察する．

2. SALMON: 光科学アプリケーション

2.1 概要

“SALMON (Scalable Ab-initio Light-Matter simulator for Optics and Nanoscience)” は，主に筑波大学計算科学研究センターと自然科学研究機構分子科学研究所にて開発されている，光と物質の相互作用の第一原理計算を目的とした光科学シミュレータである [4]．SALMON は，我々がこれまでにメニーコアプロセッサへ向けた最適化を行ってきた ARTED [9] と，ARTED から派生した GCEED [3] を統合し，それぞれ異なるスケールでのシミュレーションを 1 つのアプリケーションとして利用可能としている．SALMON のベースとなっている 2 つのアプリケーションは，対象とする問題 (方程式) の規模，並列化における分割方法などが異なるが，計算内容はほぼ同一で，我々が ARTED で最適化してきたコードをほぼそのまま活用できると期待している [1], [2]．現在，ARTED と GCEED がそれぞれ持っている計算コードの統合を行っており，本研究では，ARTED が対象としてきたシミュレーションを対象として性能評価を行う．

SALMON は，時間依存 Kohn-Sham (TDKS, Time-Dependent Kohn-Sham) 方程式において，実時間・実空間法により電子の波動関数の記述および求解を行っている．また，電子動力学と電磁気学を組合せたマルチスケール計算が可能で，電磁気学の方程式には Maxwell 方程式を時間領域差分 (FDTD, Finite-Difference Time-Domain) 法により解き，TDKS 方程式と Maxwell 方程式を結合する．**図 1** に，TDKS 方程式と 3-D Maxwell 方程式を組合せたマルチスケール計算の際の空間格子について示す．マルチスケール計算は，問題の規模から 100 ノードを超えるある程度規模の大きなシステムを必要とする [3], [9]．しかしながら，実行時間のほとんどは第一原理計算である TDKS 方程式の求解を行う時間発展計算に費やされるため，本研究では TDKS 方程式のみを求解した場合の強スケーリング性能について評価を行う．

2.2 並列化方法

図 1 に示すように，マルチスケール計算では，2 つの方程式を解くが，Maxwell 方程式はマクロ格子点数 `NZ` をパラメータとし，TDKS 方程式は下記 3 つのパラメータで構成される．

- ブロッホ波数空間格子 (`NK`)
- バンド (`NB`)
- 3 次元空間格子点 (`NL = (NX, NY, NZ)`)

従って，計算領域となる波動関数配列は (`NZ, NK, NB, NL`) の 4 次元配列になるが，本研究では TDKS 方程式だけを解くため (`NK, NB, NL`) の 3 次元配列を考える．

TDKS 方程式の求解を MPI で分散並列化する場合，最大パラメータにあたる波数空間 `NK` のみを分割し，分割した方程式を OpenMP で並列計算する．波数空間のみが分割されるため，実空間 (`NL`) は一切分割されず，袖領域の交換は不要である．ただし，波数空間を束ねる必要があるため，`NK` を分割した全 MPI プロセス間で，`MPI_Allreduce` による総和計算が用いられる．`MPI_Allreduce` のメッセージサイズは，3 次元空間格子点 (`NL`) に相当し，倍精度浮動小数点数ベクトルの総和を行う．SALMON のシミュレーションが対象とする `NL` のサイズは 16^3 や $20 \times 36 \times 52$ など小規模で，メッセージサイズは高々 1 MB 未満となり，通信はボトルネックとならない．

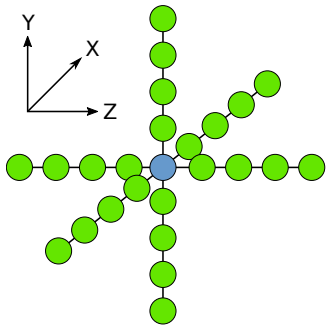

図 2　25 点ステンシル計算のメモリアクセスパターン

```
#ifdef __AVX512F__
/* AVX-512 implementation */
#define _mm512_storenrngo_pd _mm512_stream_pd
#elif __MIC__
/* IMCI implementation */
inline __m512i _mm512_loadu_si512(int const* v) {
  __m512i w = _mm512_loadunpacklo_epi32(w, v + 0);
  return      _mm512_loadunpackhi_epi32(w, v + 16);
}
#endif
```

図 3　IMCI から AVX-512 への変換

2.3　ステンシル計算

TDKS 方程式の時間発展において，最も高い計算コストを持つのが電子の波動関数に対するハミルトニアンの計算である．ハミルトニアン計算は，倍精度複素数で表現された波動関数について，25 点ステンシル計算と擬ポテンシャル計算の 2 つを計算する．**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す．

ハミルトニアン計算は，1 回の時間発展で 1 個の 3 次元実空間格子 (NL) に対し，ステンシル計算と擬ポテンシャル計算を計 4 回計算する．擬ポテンシャル計算は，物理的直感に基づいて実装されるアルゴリズムのため，最適化が非常に困難である．したがって，本節では説明を省略する．前述の通り，1 個の実空間の計算は，OpenMP の 1 スレッドで行われるため，各スレッドは複数個の実空間格子に対しハミルトニアンを逐次的に計算する．各実空間格子は，独立かつ閉じた空間のため，1 回の時間発展で行われる 4 回のステンシル計算と擬ポテンシャル計算において OpenMP のスレッド同期または MPI による通信は一切発生しない．

ハミルトニアン計算は，実行時間の最大 8 割強を占める [10] ため，我々の最適化は，メモリバンド幅に律速されるステンシル計算を中心としている．

2.4　これまでの研究: Xeon Phi への最適化

これまで行ってきた研究において，我々は Intel Xeon Phi の最初のプロダクトである Knights Corner (KNC) に対する最適化を行い [1]，それらの最適化が Knights Landing (KNL) においても有用であることを確認している [2]．特に，計算時間のうち最大 8 割強を占める 25 点ステンシル計算に傾注した最適化を行ってきた．

Fortran90 のコードを最適化し，コンパイラの自動ベクトル化の促進を行ったが，性能が不十分と考え，KNC がサポートする 512-bit SIMD 命令である IMCI (Initial Many-Core Instruction) を用いて C 言語での手動ベクトル化を実装した [1]. 特に，連続領域である Z 次元のメモリアクセス最適化を行い，メモリ帯域要求を下げ大幅な性能向上に成功した．concatinate shift 命令である `alignr` 命令を用いて，性能が低くボトルネックとなりやすい `gather` 命令を使わずに連続領域のアクセスを最適化した [11]．この最適化は，Intel が公開したレポート [12] をベースに周期境界条件に対して最適化を行ったもので，[12] とは異なり周期境界条件とそれ以外の場合の両方に適用可能である．

KNL では IMCI はサポートされておらず，AVX-512 が 512-bit SIMD 命令としてサポートされている．KNL における AVX-512 実装は，KNC 向け IMCI 実装をベースに実装したものである．IMCI から AVX-512 へ変換する場合，2 つの命令セットのフォーマットの違いを吸収しなければならない [13]．我々の IMCI 実装は，幾つかの命令セットが AVX-512 と非互換だったが，命令の置換か，5~10 行程度のインライン関数の置換で解決可能で，これらは全てコンパイル時のプリプロセッサによる置換でほぼ実装コストなしに実現した．コード変換の例を，**図 3** に示す．ここで，`__AVX512F__`は AVX-512 用，`__MIC__`は KNC 用にコンパイルした際，それぞれコンパイラによって定義されるシンボルである．

さらに，連続領域のメモリロードと同時に L1 キャッシュへのプリフェッチ命令を発行し，メモリアクセスレイテンシの隠蔽を行った．我々のコードは常に 64 Byte アラインされた状態でロード命令が発行されるため，ロードしたアドレスから 64 Byte 先のアドレスをプリフェッチすることで，キャッシュライン単位での効率的なプリフェッチが行える．また，連続領域のメモリロードに合わせて発行するため，次の反復で用いられるデータが L1 キャッシュにロードされる．

3.　アプリケーションへの GPU の適用

3.1　OpenACC+CUDA

アプリケーションを NVIDIA GPU へ対応させる場合，OpenACC または CUDA での実装が考えられる．また，本研究のターゲットは Fortran で実装されているため，どちらの場合も利用可能なコンパイラは PGI コンパイラに限られる．我々は，下記の理由から OpenACC による実装が最適であると考えている．

(1) 開発コスト: コード全てを CUDA に対応させるのは

非常に時間がかかる，また GPU-CPU 間のメモリコピーといった GPU の制御だけでも多くのコードの追加が必要

(2) 性能とのバランス: 全てを CUDA で実装することで最高性能を得られると期待されるが，最適化にかかるコストが釣り合わない可能性もある

(3) 実装者と利用者の技術レベルの乖離: HPC 研究者は CUDA で適切なコードを実装可能と期待されるが，実際の利用者は計算科学の研究者であり，CUDA コードの修正コストが高い

特に，3 つ目の理由が最も重要である．SALMON のように実装されて間もないアプリケーションの大部分は，将来変更される可能性が高いコードである．このようなアプリケーションのコード全てを CUDA で実装した場合，計算科学者がその性能を保ったまま計算内容を修正するのは，CUDA や GPU に関する知識が要求され非常に敷居が高い．OpenMP がスレッド並列 API のデファクトスタンダードとなって久しいが，近いインターフェイスを持つ OpenACC は，CUDA に比べ計算科学者に対し参入障壁を低く抑えられるものと期待される．

OpenACC の場合でも，細かい GPU への最適化が困難なため単純な移植だけで高い性能を得ることは非常に困難で，書き換えは依然として必要である．CUDA に比べ全ての書き換えや，コード構造を維持したままの対応などは容易に可能であると考えられるが，計算コストの高いカーネルでは，CUDA での実装が要求される可能性が高い [14]．本研究のターゲットにおいて，ボトルネックとなるコードはハミルトニアン計算であると予め分かっている．また，同計算は将来のコード変更の可能性が低く，ステンシル計算のメモリアクセスパターンが変わる可能性が低い．そのため，手動ベクトル化や CUDA 化など非常にコストの高い最適化を行っても，利用者への影響は無視できる範囲にある．

OpenACC version 2.0 からは，Native API (CUDA) との相互運用性が確保され，OpenACC で確保したメモリを Native API で利用できるポインタとして取得する `host_data` ディレクティブが追加されている．この仕様により，OpenACC と CUDA を共存させることが可能となった．本研究では，まず OpenACC 実装を行い，OpenACC 実装に対しボトルネックとなるコードのみを CUDA 化することを OpenACC+CUDA 実装と呼ぶ．

我々は，NVIDIA 社の成瀬 彰氏にご協力いただき，前身の ARTED において OpenACC と OpenACC+CUDA のベース実装をご提供頂いた [*1]．なお，OpenACC の実装はオリジナルの ARTED および SALMON に取り込まれている．ベース実装では，ARTED で最も支配的な計算であるハミルトニアンを GPU で計算し，プロセス内の計算領域のリダクション，ハミルトニアンと同様のステンシル計算の一部が GPU 化されている．その後，我々は SALMON において計算アルゴリズムの一部を変更したため，ベース実装では時間発展計算時に各ステップで計算領域の波動関数配列すべてを GPU から CPU にコピーする必要が生じてしまい，メモリコピーが性能のボトルネックとなっていた．したがって，我々はベース実装に対し時間発展の計算時に，計算領域である波動関数配列にアクセスする計算について GPU 化を行い，同メモリの GPU-CPU 間のメモリコピーを削除した．

[*1] https://github.com/anaruse/ARTED

3.2 スレッド数・レジスタ数の調整

本節では，ステンシル計算の最適化の前に，CUDA スレッドと 32-bit レジスタの利用率について考える．本研究のステンシル計算は，先述の通り 25 点の倍精度複素数計算で，格子点 1 点の更新に必要なデータが 158 Byte と比較的多い．Tesla P100 以降，各 CUDA スレッドが使用できる 32-bit レジスタの数は最大 255 本となり，Tesla K40 と比べてほぼ倍になった．しかしながら，対象コードは倍精度複素数の計算が中心のため，Fortran90 のコードに OpenACC ディレクティブを付与するといった単純な実装では，適切なスレッド数の設定をすると各 SM が持つレジスタを使い切ってしまう．

各スレッドのレジスタ使用率の削減を考える前に，各スレッドブロックにいくつのスレッドを生成するかを考える．Warp スケジューラは， P100 では 32 CUDA コアあたり 1 個が用意され，各スケジューラはクロックあたり 2 Warp の命令をディスパッチできる．V100 では，1 SM (Streaming Multiprocessor) あたり 4 つの Warp スケジューラを持ち，各スケジューラは 32 スレッドの命令のディスパッチを 1 クロックで行う．また， P100 と V100 は SM あたり 64×2^{10} 個の 32-bit レジスタを持つ．すなわち，スレッドあたりの最大レジスタ本数を 128 に設定すると，128 スレッドのブロック 4 つを 1 個の SM で実行可能な状態にできる．以上より，各 SM が持つ 32-bit レジスタをすべて活用し，かつ効率的なスケジューリングを行うためには，両 GPU ともに 128 スレッド/ブロックが最適なスレッド数と考えられる．しかしながら，スレッドあたりの 32-bit レジスタの数を 128 に制限すると，我々が対象とする倍精度複素数の 25 点ステンシル計算ではレジスタスピルが発生してしまう．

CUDA では，レジスタスピルが発生すると，SM のローカルメモリにデータを退避させる．ローカルメモリはスレッドあたり最大で 512 KB が利用できるが，実体はグローバルメモリ (P100, V100 では HBM2) のため，アクセスレイテンシやバンド幅はグローバルメモリと等価である．また P100 は，ローカルメモリへのアクセスを常に L2 にキャッシュしている [15]．一方で，V100 ではローカル

```
!$acc kernels
!$acc loop gang
do ikb=1,NKB
!$acc loop vector(128) collapse(2)
do ix=1,NX
do iy=1,NY
!$acc loop seq
do iz=1,NZ
  A(iz,iy,ix,ikb) = ...
```

図 4　本研究のステンシル計算の並列化方法 (OpenACC)

メモリへのアクセスがキャッシュされるという記述は見つかっていない．つまり，P100 では全 SM で共有される 4 MB の L2 キャッシュに，ローカルメモリ，すなわちスピルしたレジスタのデータが退避されるため，レジスタスピル発生時のローカルメモリへのアクセスコストが非常に高くなる．P100 でも，メモリアクセスがボトルネックとなる計算では，レジスタスピルは依然として性能低下の原因となる可能性が高い．

レジスタスピルの回避は，ループ分割による使用レジスタ本数の削減が最初に挙げられる．OpenACC でループ分割を行う場合，カーネルレベルで処理を分割することで，カーネルが利用するレジスタ数を削減することになる．しかしながら，現在のところ OpenACC の枠組みの中では，ループ分割による効果は得られていない．したがって，実装*1 では，同コードについてステンシル計算を CUDA で実装し，レジスタ使用率の削減とメモリアクセスの最適化を行われている．

3.3　CUDA でのメモリアクセス最適化

まず，並列化方法について述べる．ターゲットコードは，3 次元の実空間と波数空間を持っており，全波数空間について実空間においてステンシル計算を行う必要がある．したがって，CUDA では，波数空間をスレッドブロックに割り当て，各スレッドブロックで 1 個の実空間を計算するのが最適である．これを OpenACC に置き換えるが，それぞれスレッドブロックは gang ループ，スレッドは vector ループという名前になる．OpenACC で，上記の並列化を行うと図 4 のようなディレクティブを書くことになる．ここで，NX，NY，NZ はそれぞれ 3 次元実空間のサイズ，NKB は波数空間のインデックスを示す．`$acc kernels` は並列実行領域を示し，CUDA に変換される場合，単一また複数のカーネルが生成される．`collapse` でループを融合し，X-Y 平面を 1 つのループとして並列化する．これは，3 次元実空間が $(16, 16, 16)$ など比較的小さなブロックのため，ループ融合により並列性を確保する必要がある．最後に，最内の連続方向にあたる Z 次元は，`$acc loop seq` を指定し 1 スレッドで逐次計算する．これをベースとして，メモリアクセスの最適化を考える．

```
!$acc kernels
!$acc loop gang
do ikb=1,NKB
!$acc loop vector(128) collapse(2)
do ix=1,NX
do iy=1,NY
!$acc loop seq
do iz=1,NZ
  B(iz,iy,ix,ikb) &
    = Cx * (A(iz,iy,mod(ix+1,NLx),ikb) &
           -A(iz,iy,mod(ix-1,NLx),ikb))
end do
end do
end do
end do
```

図 5　ステンシル計算における通常のメモリアクセス (X 次元の差分計算のみ)

```
B(iz,iy,ix,ikb) = 0
!$acc kernels private(g)
!$acc loop gang
do ikb=1,NKB
!$acc loop vector(128) collapse(2)
do ix=1,NX
do iy=1,NY
!$acc loop seq
do iz=1,NZ
  g = A(iz,iy,ix,ikb)
  B(iz,iy,mod(ix-1,NLx),ikb) &
    = B(iz,iy,mod(ix-1,NLx),ikb) - Cx * g
  B(iz,iy,ix,ikb)            &
    = B(iz,iy,ix,ikb) + g
  B(iz,iy,mod(ix+1,NLx),ikb) &
    = B(iz,iy,mod(ix+1,NLx),ikb) + Cx * g
end do
end do
end do
end do
```

図 6　グローバルメモリへのリードアクセスを削減した場合 (X 次元の差分計算のみ)

本節では，実装*1 の概要を述べる．既に述べた通り，まず CUDA 実装ではループ分割を行う．3 次元空間のうち，最外にあたる X 次元のみを計算，Y-Z 平面を計算する 2 つのカーネルに分割し，それぞれメモリアクセスを最適化する．スレッドインデックスの計算など煩雑な部分を省略するため，ここでは Fortran/OpenACC で 1 次差分のステンシル計算をコード例として記載する．

X 次元は，Y-Z ストライドのアクセスとなるため，アクセスコストが極めて高い．したがって，同カーネルではメモリアクセスのパターンを変更する．通常，図 5 のように，ステンシル計算では更新対象の格子点から見たときに，近傍点へのメモリアクセスを行うのが一般的である．この場合，各反復で複数の近傍点に対しメモリアクセスが行われるが，CPU のようなキャッシュメモリを活用するアーキ

```
complex(8) :: cache_y(3)

!$acc kernels private(cache_y)
!$acc loop gang
do ikb=1,NKB
!$acc loop vector(128) collapse(2)
do ix=1,NX
do iy=1,NY
  cache_y(1) = A(iz,NLy,ix,ikb)
  cache_y(2) = A(iz,  0,ix,ikb)

!$acc loop seq
do iz=1,NZ
  cache_y(3) = A(iz,mod(iy+1,NLy),ix,ikb)

  B(iz,iy,ix,ikb) = B(iz,iy,ix,ikb) &
  + Cy * (cache_y(3)-cache_y(1)) &
  + Cz * (A(mod(iz+1,NLz),iy,ix,ikb) &
         -A(mod(iz-1,NLz),iy,ix,ikb))

  cache_y(1) = cache_y(2)
  cache_y(2) = cache_y(3)
end do
end do
end do
end do
```

図 7 Y-Z 平面のグローバルメモリへのリードアクセス削減

テクチャでは，連続した近傍点についてはキャッシュメモリへのアクセスとなり，効率的なメモリアクセスが期待できる．しかし GPU の場合，メモリアクセスは Warp 単位でまとめられるため，Warp 内のスレッドが不連続にメモリアクセスを繰り返すと性能低下に繋がる．そこで，X 次元の計算カーネルでは**図 6** のように，各反復でアクセスする 1 個の格子点が，どの格子点の更新に必要とされているか，と変更する．このアクセス方法では，各反復でグローバルメモリへのリードアクセスは 1 回になり，またループを並び替えることで全スレッドがコアレスアクセスを実現できる．図 6 では，計算の一時データもグローバルメモリに書き込まれているが，実装ではローカルメモリを使用し，グローバルメモリへのライトアクセスも各格子点に対し 1 回になるように最適化している．

Y-Z 平面の計算カーネルでも，同様にメモリアクセス回数を削減する．連続方向である Z 次元は，最低限，L1 にキャッシュされると期待されるため，Z 次元は通常通りのメモリアクセスを行う．Y 次元は，Z ストライドのアクセスになるため，キャッシュから溢れる可能性が高い．したがって，Y 次元については，必要な近傍点をローカルメモリにキャッシュし，リードアクセスを削減する．Y-Z 平面のリードアクセス削減後のコードを**図 7** に示す．計算時には，X 次元の計算，Y-Z 平面の計算の順でカーネルをキックする．

表 1 Pre-PACS-X (PPX) 諸元

CPU	Xeon E5-2690 v4×2 2.6 GHz with 14 cores
GPU	NVIDIA Tesla P100×2 or V100×2 (PCIe card version)
Memory	CPU: 64 GB DDR4-2400 GPU: 16 GB HBM2 / GPU
Network	Mellanox InfiniBand EDR 16x with Mellanox OFED 3.4-2.0.0
OS	CentOS 7.3.1611 (Compute node)
Software	PGI Compiler 17.10, CUDA 9.0, OpenMPI 2.1.2

表 2 Oakforest-PACS (OFP) 諸元

CPU	Xeon Phi 7250 1.4 GHz with 68 cores
Memory	16 GB MCDRAM, 96 GB DDR4-2400
Network	Intel Omni-Path Architecture
OS	CentOS 7.2.1511 (Compute node)
Software	Intel Compiler 18.0.1, Intel MPI 2018 update 1

表 3 ISSP System C 諸元

CPU	Xeon Gold 6148×2 2.4 GHz base clock with 20 cores
Memory	192 GB DDR4-2666
Network	Mellanox InfiniBand EDR 4x
OS	Red Hat Enterprise Linux 7
Software	Intel Compiler 18.0.1, Intel MPI 2018 update 1

4. 性能評価

4.1 評価環境

本研究では，GPU クラスタとして，筑波大学計算科学研究センター (CCS) にて稼働している，Pre-PACS-X (PPX) を用いる [16]．同システムは，CCS が長年開発している PACS シリーズの次世代機に向けた評価環境として稼働している小規模クラスタで，P100 と V100 を 2 枚搭載したノードがそれぞれ提供されている．PPX の諸元を**表 1** に示す．PPX では，Hyper-Threading をオフにした Broadwell Xeon CPU が 2 ソケット，P100 or V100 が各ソケットに PCIe で 1 台接続されており，InfiniBand EDR が片側に接続されている．現在は表 1 の構成で，P100 を接続したノードが 4 台，V100 を搭載したノードが 1 台稼働している．

Xeon Phi の評価環境には，JCAHPC にて稼働している Oakforest-PACS (OFP) を用いる [17]．同システムは日本国内最大の KNL クラスタで，TOP500 では 2017 年 11 月に公開された最新のリストにおいて世界 9 位の性能を持つ [5]．OFP の諸元を**表 2** に示す．OFP は，68 物理コアの Xeon Phi 7250 が 1 ソケット，ネットワークデバイス

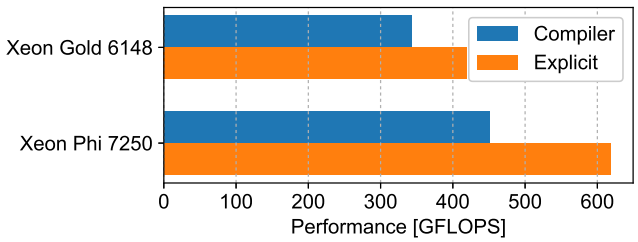

図 8　ステンシル計算への手動ベクトル化の効果

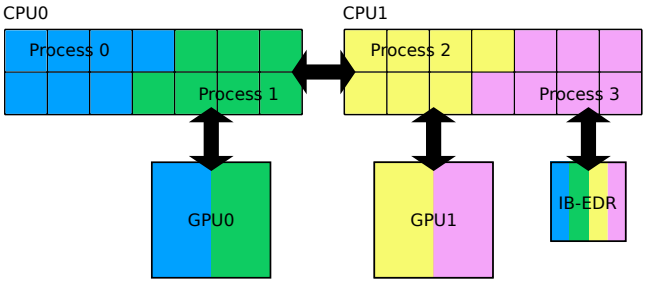

図 9　CPU/GPU のアフィニティ設定例

として Omni-Path Architecture が接続された，一般的な PC クラスタである．また，プロセッサのクラスタリングは Quadrant モード，メモリは Flat モードを用いている．

Xeon CPU との比較用に，本研究では東京大学物性研究所 (ISSP) にて試験稼働中の System C を用いる [18]．ISSP System C の諸元を**表 3** に示す．同システムは，Intel Skylake-SP (SKL) アーキテクチャの Xeon CPU を用いた計算ノード 252 台で構成された一般的な CPU クラスタである．また，SKL は Xeon CPU プロダクトでは最初に AVX-512 命令をサポートするアーキテクチャで，我々がこれまでに行ってきた Xeon Phi の手動ベクトル化実装を一切のコード変更なしに移植できる．

データセットには，GPU でも十分な並列性を確保できるように，計算領域である電子の波動関数のサイズを全体で 4 GB とした．ステンシルの計算サイズとなる 3 次元実空間格子点は 16^3 である．このデータセットでは，アプリケーション全体のメモリ使用量が 16 GB 以上となり，KNL では MCDRAM と DDR4 を組み合わせる必要がある．GPU では，アプリケーション全体ではなく担当する計算に必要なデータだけを GPU メモリに確保するため，この問題は起きていない．

KNL が持つ MCDRAM と DDR4 のように，複数の異なるバンド幅を持つメモリを組み合わせる場合，バンド幅がより高いメモリをスクラッチパッドキャッシュ的に用いることで，トータルのメモリバンド幅を向上させることが可能である．しかしながら，この手法は各メモリにどのようにデータを配置するかを全て開発者がケアしなければならず，効率的にメモリ間のデータ転送をハンドリングするのは困難を極める．本研究のアプリケーションは，スクラッチパッドキャッシュとして動作する MCDRAM を効率良く利用可能になっており，ステンシル計算だけを見ると，計算は MCDRAM で閉じている．現在まで，MCDRAM と DDR4 を組み合わせた場合と，MCDRAM のみをメインメモリとして用いた場合では性能はほぼ等価であることが分かっている．

4.2　AVX-512 Processors

最初に，これまで我々が行ってきた手動ベクトル化の効果について述べる．2.4 で述べた通り，我々は研究の初期に KNC が持つ 512-bit 長の IMCI SIMD 命令を用いてステンシル計算の手動ベクトル化による最適化を行った [1]．その後 KNL がサポートする AVX-512 SIMD 命令への移植と性能評価により，コンパイラが行うベクトル化よりも効率よく動作していることが分かっている [2]．

AVX-512 は複数のサブセットに分かれており，同じ AVX-512 をサポートするプロセッサであっても，利用できる命令が異なる点に注意が必要となる．本研究で実装した手動ベクトル化コードでは，AVX-512 対応の全プロセッサがサポートする基本命令セットである AVX-512F (Foundation) のみを利用している．すなわち，SKL など AVX-512 をサポートする全てのプロセッサで，これまで実装してきた手動ベクトル化コードは実行可能である．SKL はプロダクトが 6100 番台以降の場合，AVX-512 の演算ユニットが各コアに 2 個用意されるため，FLOP/Cycle は KNL と SKL で等価である．

図 8 に，Xeon Phi 7250 (KNL) と Xeon Gold 6148 (SKL) のステンシル計算性能について示す．それぞれ，シングルソケットでの性能を示している．“Compiler” は，Fortran90 で書かれたステンシルコードのベクトル化を Intel コンパイラに任せた場合，“Explicit” は AVX-512 Intrinsic を用いて C 言語でステンシル計算を実装した場合の性能を示している．両方のプロセッサで，Intel コンパイラが行う最適化に比べ我々の手動ベクトル化コードがより高い性能を示しており，現在のところ AVX-512 による手動ベクトル化が有効であることがわかる．しかしながら，KNL に比べ SKL では性能の差が縮まっており，プロセッサの性能向上とコンパイラが AVX-512 に対し習熟度が向上しているものと推察される．

4.3　P100, V100

GPU 間の評価では，PPX に搭載されている V100 の台数からくる制限により，1 ノード 2 GPU のみの評価を行う．GPU クラスタは，一般的な構成としてノードあたり CPU が 2 ソケット，GPU が 2 台以上接続されることが多く，GPU のアフィニティ設定は性能に大きく影響する．Kepler アーキテクチャ (GK110 GPU) から，複数の CPU プロセスで同一の GPU に計算を行わせる際，実行キュー

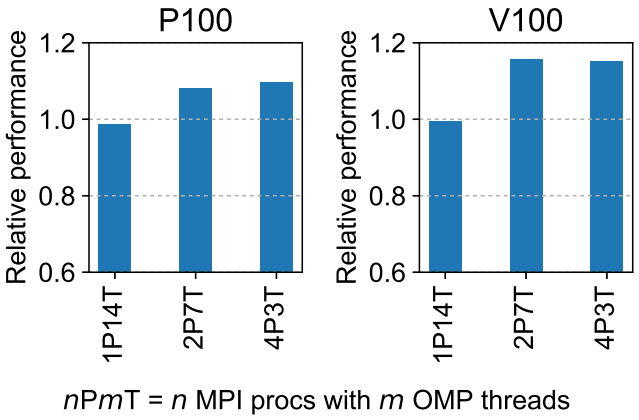

図 10　CUDA MPS の効果

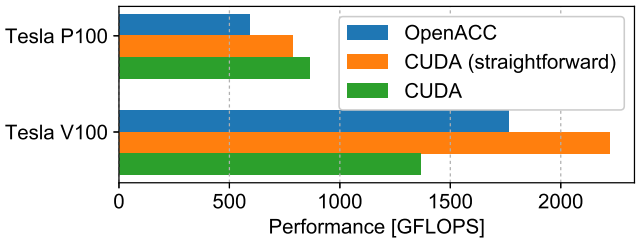

図 11　ステンシル計算の OpenACC と CUDA 実装の性能比較

のスケジューリングを効率化する仕組みとして，CUDA Multi-Process Service (CUDA MPS) が提供されている．

CUDA MPS は，ユーザまたはシステムレベルで GPU への実行キュー (CUDA カーネル) の制御を行うデーモンを立ち上げ，SM に余裕があり依存関係のないキューが投入されている場合に，同時に複数のキューを処理するようにスケジュールする．この仕組みにより，GPU は投入されたキューに対し適切に SM を割り当て，計算リソースの利用効率を引き上げることができる．ただし，CUDA MPS デーモンが CPU プロセスと GPU の仲介を行うため，同時に複数キューを処理できない，すなわち SM に空きがない場合はすべて実行時オーバーヘッドとなる．Volta アーキテクチャでは，CUDA MPS をハードウェアベースで実装し，更なる性能向上を図っている [7]．

PPX では GPU が各ソケットに 1 台接続されているため，ソケットに複数の MPI プロセスを割り当て，CUDA MPS を通じて GPU を共有する．**図 9** には，ソケットあたり 2 MPI プロセスで GPU を共有したときの CPU と GPU のアフィニティ設定例を示す．PPX ではソケットあたり 14 物理コアを持つため，ソケットあたり 2 MPI プロセスの場合，各 MPI プロセスは 7 OpenMP Thread でプロセス内 CPU 並列を行う．MPI プロセスの割当数によっては，OpenMP の合計スレッド数は CPU が提供するコア数よりも少なくなる．

CUDA MPS の効果について**図 10** に示す．このグラフは，CUDA MPS を無効にした状態で，1 MPI プロセス 14 OpenMP スレッドで得られる実行性能を基準とした相対性能を示している．グラフ横軸の nPmT は，それぞれ n プロセスで GPU を共有し，各プロセスは m OpenMP スレッドで動作することを示す．1P14T の場合，1 プロセスで 1 個の GPU を専有するため CUDA MPS はそのままオーバーヘッドとなり，1%と無視できる程度の性能低下が見られる．しかしながら，複数プロセスで GPU を共有することで CUDA MPS が適切に CUDA カーネルの実行を制御し，P100 では 1.1 倍，V100 では 1.15 倍とそれぞれ効果が得られている．

表 4　OpenACC と CUDA (straightforward) のレジスタ使用数

	OpenACC		CUDA	
Max # of registers	255	128	255	128
Used # of 32-bit registers	198	128	180	128
Spill stores+loads [B]	0	104+112	0	0

次に，ステンシル計算の OpenACC 実装と CUDA 実装の性能評価を行う．4 MPI プロセス 3 OpenMP スレッドで実行した場合の，ステンシル計算の性能を**図 11** に示す．比較用に，PGI OpenACC の最適化メッセージを参照し，OpenACC 実装とほぼ同じ動作を想定した CUDA 実装 (シングルカーネル) を “CUDA (straightforward)” として評価した．CUDA (straightforward) は，OpenACC と同じ性能が期待されるが，実際には OpenACC よりも高い性能が得られている．

表 4 に，PTX の出力情報から P100 における OpenACC と CUDA (straightforward) 実装のレジスタ使用数を示す．V100 においても，ほぼ同じ傾向を示している．各スレッドが利用できる 32-bit レジスタ数は最大 255 で，本研究では各 SM にある全レジスタを活用できるように，各スレッドで利用可能なレジスタを 128 に制限している．OpenACC では，レジスタ数を制限することでレジスタスピルが発生しているが，一方で CUDA (straightforward) 実装では，レジスタ数を制限してもレジスタスピルは発生せず，使用レジスタ数だけが減少している．この違いは，レジスタ数制限の方法によって生じている．OpenACC では，PGI コンパイラに`-ta=tesla,maxregcount:128` とオプションを指定し ptxas に対しレジスタ使用数を 128 に制限しているが，CUDA ではカーネルに`__launch_bounds__(128,4)` を付与し 128 スレッド 4 ブロックで最低限動作する，つまりレジスタ数を 128 に抑えてコンパイルするように指示している．CUDA でも，`__launch_bounds__`の変わりに ptxas のオプションで`--maxrregcount=128` を指定するとレジスタスピルの発生を確認した．恐らく，`__launch_bounds__` では制限値に応じて最適化が CUDA コンパイラによって行われ，レジスタスピルがない PTX コードが出力されると考えられる．PGI OpenACC でも，同様の指定ができれば CUDA (straightforward) と遜色ない性能が得られるはずである．

P100 の性能を検証すると，OpenACC 実装ではレジスタスピルにより，十分な性能が得られていない．レジスタスピルが発生すると，レジスタから溢れたデータはローカルメモリに退避される．ローカルメモリへのアクセスは，P100 では常に L2 キャッシュを経由するため，4 MB しかない L2 キャッシュに GPU 上の全スレッドがデータを退避させることになり，性能低下に繋がると推察される．P100 では，CUDA 実装のようにカーネルを 2 つに分割して，レジスタスピルの発生を回避したほうが高い性能を得られると考えられる．

一方で，V100 では最適化したはずの CUDA 実装は OpenACC 実装よりも低い性能となっており，現在原因を調査中である．V100 でも，OpenACC 実装は依然としてレジスタスピルが発生しており，十分な性能は期待できない．先に述べたように，P100 ではローカルメモリへのアクセスを L2 キャッシュ経由で行うことが明示されているが，V100 では現在その記述が見つかっていない．もし，V100 がグローバルメモリに直接データを退避するとすれば，レジスタスピル時に L2 キャッシュへのアクセス集中を避けることができ，性能低下を緩和できると推察される．レジスタスピル時のローカルメモリのアクセスペナルティが緩和されたとすれば，レジスタスピルを避けるためにマルチカーネルで実行する CUDA 実装よりも，シングルカーネルでレジスタスピルが発生する OpenACC 実装が高速である理由を解決する手がかりとなる．

また V100 では，L1 キャッシュとシェアードメモリが統合され合計 128 KB に変更されており，アクセスレイテンシやバンド幅も改善されている．容量増加やアクセスコストが下がったことにより，OpenACC 実装のように単純な実装でも高い性能が得られたのではないかと考えられる．P100 の場合，L1 キャッシュはテクスチャメモリと統合されており合計 24 KB だったが，V100 はシェアードメモリを一切使わなければ最大 128 KB まで L1 キャッシュに割り当てることができる．L1 キャッシュとシェアードメモリの割当配分は，カーネルで利用されるシェアードメモリのサイズから，ランタイムが自動的に判断する．ユーザからは，カーネル単位でシェアードメモリの利用率またはバイトサイズをヒントとして与えることができる．

4.4 GPU, KNL, SKL

次に，全プロセッサの性能比較を行う．KNL での詳細な性能評価については [2] を参照されたい．まず，ステンシル計算の比較を図 **12** に示す．ステンシル計算では，V100 は HBM2 メモリによる高いメモリバンド幅と，高い理論ピーク演算性能により，KNL に比べ 3 倍を超える性能を達成した．また，P100 も最適化により KNL よりも高いステンシル計算性能が得られている．SKL は，このグラフではシングルソケットでの性能を示しているが，一般的なクラスタでは 2 ソケットで 1 台の計算ノードを構築するため，それを加味すると，SKL の計算ノードは P100 1 台とほぼ同等の性能を達成しうる．

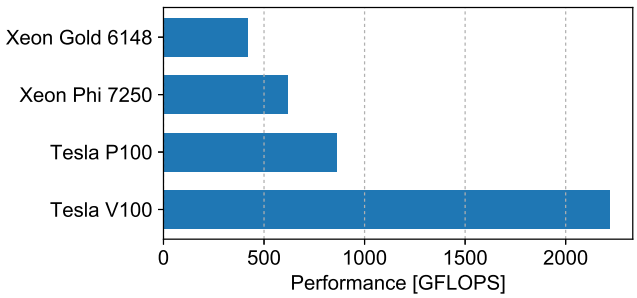

図 **12** 全プロセッサのステンシル計算の性能比較

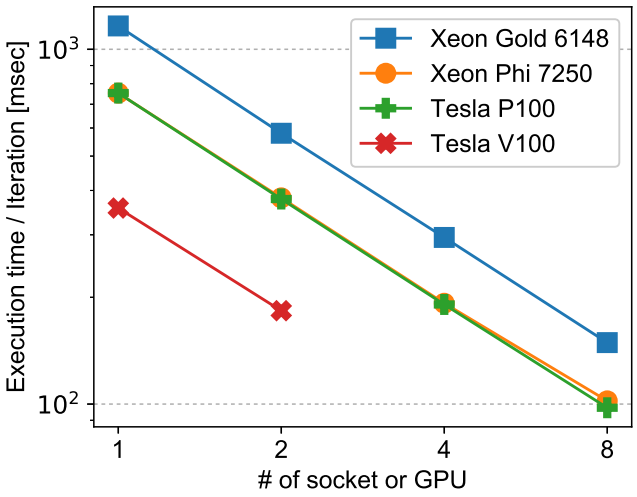

図 **13** 全システムの時間発展計算の性能比較

次に，時間発展計算の性能について図 **13** に示す．このグラフは，計算ノード単位ではなく，ソケットまたは GPU の台数でプロットされていることに注意されたい．ステンシル計算では，V100 は KNL に比べて 3 倍の性能を達成していたが，全体性能の評価では P100 と KNL がほぼ同等性能で，V100 は P100 と KNL 2 台相当の性能を得た．計算カーネル単位では，CPU に比べて高い性能を有するが，全体性能では GPU だけでなく CPU での計算や通信処理などが必要となる．CPU クラスタでは発生しない制御がオーバーヘッドとなり，結果として全体性能を鈍化させている．しかしながら，V100 は高いメモリバンド幅とピーク演算性能によりそれらの弱点を補っていると言える．

5. 考察

最後に，今回評価したプロセッサの性能について考察する．表 **5** に，各プロセッサの理論ピーク性能やメモリバンド幅，電力効率などの比較表を示す．ここで，SKL (Xeon Gold 6148) の理論ピーク性能は AVX-512 動作時のクロック数から求めている．また，KNL の MCDRAM のバンド幅を記載しているが，MCDRAM は理論メモリバンド幅

表 5　各プロセッサの比較

	SKL	KNL	P100	V100
理論性能 [GFLOPS]	1024	3046	4800	7000
理論バンド幅 [GB/s]	127.8	none	732	900
実効バンド幅 [GB/s]	98.5	490.4	551.3	839.1
Byte/FLOP	0.096	0.160	0.114	0.119
ステンシル計算効率	41.0%	20.3%	18.0%	31.8%
TDP [W]	150	215	250	250
GFLOPS/W	6.82	14.16	19.2	28

が公表されていない．また，実効メモリバンド幅は，それぞれ STREAM ベンチマークを用いて計測した [19], [20]．SKL では，NUMA を適切に制御し，1 ソケットでの実効メモリバンド幅を記載している．P100 と V100 の各理論値は，すべて PCIe カードのプロダクトでの値を記載している．

Byte/FLOP は KNL が最も高く，実行効率が高くなることが期待されるが，KNL が約 20%に対して，V100 では約 30%の実行効率を達成している．KNL の性能評価では，計算は MCDRAM で閉じているが，MCDRAM のメモリアクセスレイテンシは DDR4 に比べて高いことが一因と考えられる．さらに，V100 は P100 からメモリバンド幅は約 1.5 倍と向上しているが，ステンシルの計算性能は約 2.2 倍向上している．V100 は，演算コアが約 1.4 倍増加しており，同時実行可能なスレッド数が増え，メモリアクセスのレイテンシ隠蔽がより行いやすくなり，効率の改善に繋がっていると考えられる．SKL では，約 40%と実行効率は最も高い．SKL は Non-inclusive な L3 キャッシュを持ち，L2 キャッシュのアクセスレイテンシがこれまでのアーキテクチャに比べ短くなっている．本研究のステンシル計算は，L2 キャッシュに収まる程度の比較的小さい実空間を大多数計算するため，L2 キャッシュのアクセスレイテンシは性能に大きく影響する．

最後に，ワットパフォーマンスについて考えると，GPU はそれぞれ 250 W と比較的高い TDP を持つが，ワットパフォーマンスは KNL に比べて高い．また，SKL のワットパフォーマンスが最も低く，コアの数が 20 コア以上と多いこと，動作周波数が GPU や KNL に比べて高いことが要因と考えられる．しかしながら，SKL はこれまで実装されてきた CPU コードをそのまま実行可能で，動作周波数も高いことから，逐次性の高いコードの場合でもある程度の性能が期待できる．それとは異なり，KNL や GPU では大幅にコードを追加する必要があり，特に GPU はコード中で CPU-GPU 間のデータハンドリングをしなければならず，実装は容易ではない．OpenACC によって，これらの問題はある程度解決の方向に向かっているが，OpenACC であっても並列性を十分に確保できるようにコードの修正は必要となる．また本研究で示したように，straightforward に実装した CUDA コードが OpenACC コードよりも高い性能を得ており，OpenACC は性能だけで見ると不十分と考えられる．少なくとも，V100 は OpenACC でも約 25%の実行効率を達成しており，V100 によって実アプリケーションの GPU への実装コストを下げつつも，一定の性能を達成可能となることが期待される．

6. まとめ

本研究では，我々がこれまでメニーコアプロセッサに向けて最適化してきたアプリケーションについて，NVIDIA Tesla P100, V100 GPU での性能評価を行い，Intel Knights Landing (Xeon Phi 7250), Intel Skylake-SP (Xeon Gold 6148) との比較を行った．

まず，我々の AVX-512 SIMD 命令向けに最適化したステンシル計算が，Knights Landing だけでなく同じく AVX-512 を採用した Skylake-SP においても，Intel コンパイラのベクトル化に比べ高い計算性能を達成したことを示した．次に，P100 と V100 での性能評価を行い，P100 では CUDA による実装・最適化が必要であったのに対し，V100 は OpenACC でも十分な性能が得られることを確認した．しかしながら，依然として CUDA での実装は OpenACC に比べ高い性能を達成している．全体性能では，P100 と KNL が等価で，V100 は 1 台で P100 と KNL それぞれ 2 台相当の性能を達成した．最後に，各プロセッサの性能の妥当性などについて考察を行った．計算性能だけを見れば，V100 が最も良い性能を達成しているが，実行効率，ワットパフォーマンス，実装コストなど，非常に多くのトレードオフの関係が存在している．

謝辞　アプリケーションの GPU 適用について，NVIDIA の成瀬 彰様には基本的な実装のご提供や最適化方法などの多くのご助言を賜りました．ここに感謝申し上げます．本研究の結果の一部は，それぞれ，JCAHPC のご協力による Oakforest-PACS の利用，筑波大学計算科学研究センターの次世代アクセラレータ型スーパーコンピュータのプロトタイプシステム Pre-PACS-X の利用，東京大学物性研究所のスーパーコンピュータ System C の試験利用により得られた．本研究の一部は，JST-CREST 研究課題「光・電子融合第一原理計算ソフトウェアの開発と応用 (課題番号: JPMJCR16N5)」により支援された．

GCEED の中心的開発者であり，SALMON の主要な開発者の一人であった自然科学研究機構・分子科学研究所の信定 克幸准教授は本稿執筆中に急逝された．同氏のこれまでの研究への貢献と成果に敬意を表し，ご冥福をお祈りする次第である．

参考文献

[1] 廣川 祐太，朴 泰祐，佐藤 駿丞，矢花一浩：電子動力学シミュレーションのステンシル計算最適化とメニーコアプロセッサへの実装，情報処理学会論文誌コンピューティ

ングシステム (ACS)， Vol. 9, No. 4, pp. 1–14 (2016).
[2] 廣川 祐太，朴 泰祐，植本 光治，佐藤 駿丞，矢花一浩：電子動力学シミュレーション ARTED の KNL システム Oakforest-PACS での全系性能評価，第 160 回 HPC 研究会，Vol. 2017-HPC-160, No. 20 (2017).
[3] M. Noda, K. Ishimura, K. Nobusada and *et al.*: Massively-parallel electron dynamics calculations in real-time and real-space: Toward applications to nanostructures of more than ten-nanometers in size, *Journal of Computational Physics*, Vol. 265, No. 14, pp. 145–155 (2014).
[4] SALMON (Scalable Ab-initio Light-Matter simulator for Optics and Nanoscience): `http://salmon-tddft.jp/`.
[5] TOP500: `https://www.top500.org/`.
[6] 小川 宏高，松岡 聡，佐藤 仁， 他：AI 橋渡しクラウド - AI Bridging Cloud Infrastructure (ABCI) - の構想，第 160 回 HPC 研究会， Vol. 2017-HPC-160, No. 28 (2017).
[7] NVIDIA: Volta Architecture, `https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/`.
[8] OpenACC: `https://www.openacc.org/`.
[9] S. A. Sato and K. Yabana: Maxwell + TDDFT multiscale simulation for laser-matter interactions, *J. Adv. Simulat. Sci. Eng.*, Vol. 1, No. 1 (2014).
[10] Y. Hirokawa, T. Boku, S. A. Sato and K. Yabana: Electron Dynamics Simulation with Time-Dependent Density Functional Theory on Large Scale Symmetric Mode Xeon Phi Cluster, *The 17th IEEE International Workshop on PDSEC2016* (2016).
[11] 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, *Proceedings of WPMVP'14*, pp. 57–64 (2014).
[12] C. Andreolli: Eight Optimizations for 3-Dimensional Finite Difference (3DFD) Code with an Isotropic (ISO), `https://software.intel.com/en-us/articles/eight-optimizations-for-3-dimensional-finite-difference-3dfd-code-with-an-isotropic-iso`.
[13] Intel Intrinsics Guide: `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`.
[14] 星野 哲也，松岡聡：圧縮性流体解析プログラムの OpenACC による高速化，第 153 回 HPC 研究会，Vol. 2016-HPC-153, No. 4 (2016).
[15] NVIDIA: CUDA Programming Guide, `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.
[16] 筑波大学計算科学研究センター：`https://www.ccs.tsukuba.ac.jp/`.
[17] 最先端共同 HPC 基盤施設：`http://jcahpc.jp/`.
[18] 東京大学物性研究所：`http://www.issp.u-tokyo.ac.jp/`.
[19] J. D. McCalpin: STREAM: Sustainable Memory Bandwidth in High Performance Computers, `https://www.cs.virginia.edu/stream/`.
[20] T. Deakin, J. Price, M. Martineau and S. McIntosh-Smith: GPU-STREAM v2.0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models, *Paper presented at P$\hat{3}$MA Workshop at ISC High Performance* (2016).