Vol.2012-HPC-136 No.10
2012/10/3

ナノ粒子群形成アプリケーションのOpenACCによる実装と性能評価

菅原 誠[†1] 小松一彦[†2,†3] 平澤将一[†1,†3]
滝沢寛之[†1,†3] 小林広明[†2]

本論文では，熱プラズマによるナノ粒子群創製プロセスにおける集団的粒子形成過程をシミュレーションするナノ粒子群形成アプリケーションを OpenACC と OpenCL を用いて実装し，両者を比較検討する．OpenACC は既存のプログラムにディレクティブを追記することにより容易に GPU を利用することが可能である．それに対して，OpenCL はより低い抽象度でのプログラミングが可能である．プログラム可能な抽象度がそれぞれ異なるため，実現可能な最適化技法が異なる．各最適化技法の性能評価により，OpenACC では CPU 実行時の最大約 1.9 倍の性能向上を，OpenCL では最大約 5.6 倍の性能向上を達成できることが分かった．また，現状の OpenACC において達成可能な性能限界と，高い性能を得るためには，OpenCL のような低い抽象度での最適化が必要であることを議論する．

Implementation and Evaluation of the Nanopowder Growth Simulation with OpenACC

Makoto Sugawara,[†1] Kazuhiko Komatsu,[†2,†3] Shoichi Hirasawa,[†1,†3]
Hiroyuki Takizawa[†1,†3] and Hiroaki Kobayashi [†2]

This paper presents an implementation of the plasma-assisted nanopowdergrowth simulation with OpenACC. OpenACC provides compiler directives to allow an existing application to use GPUs. On the other hand, OpenCL is a lower-level programming model. Since OpenACC and OpenCL offer programming models of different abstraction levels, they require different optimizations for a given application code. Therefore, in this paper, several versions of a practical application, the nanopowder growth simulation, are implemented using different optimizations. Then, the performance impact of each optimization is discussed through some experimental results. The evaluation results show that OpenACC and OpenCL can achieve 1.9x and 5.6x performance improvements, respectively. It is also demonstrated that the current version of OpenACC requires low-level performance tuning such as OpenCL programming in order to achieve a high performance comparable with OpenCL.

1. はじめに

近年，描画処理用プロセッサ（Graphics Processing Unit: GPU）をアクセラレータとして利用し，アプリケーションの高速化を実現する複合型計算システムが普及しつつある．しかし，複合型計算システムに搭載された GPU を利用するためには，NVIDIA 社製の GPU を対象としたフレームワークである Compute Unified Device Architecture(CUDA)[1] や複合型計算システム全般を対象とした Open Computing Language(OpenCL)[2] 等を用いて，既存のアプリケーションを実装し直す必要がある．既存のアプリケーションの再実装はコスト・移植時間などの観点から容易ではないため，より簡単に複合型計算システムを利用することができる手段が求められている．

このような背景から，既存のプログラムにプログラマからの指示（ディレクティブ）を追記することにより，容易に GPU を利用可能な OpenACC 等のディレクティブベースの開発環境[3][4][5] が注目されている．高い抽象度の OpenACC においても，高い性能を達成するためには GPU アーキテクチャを考慮した最適化が必要である．

本論文では，実アプリケーションであるナノ粒子群形成アプリケーション[6] を OpenACC を用いて実装し，OpenCL で実装した場合と性能を比較する．その比較結果に基づいて，現状の OpenACC で達成可能な実効性能やその記述性の限界を議論する．

本論文の構成は以下のとおりである．2 章では，本論文で実アプリケーションの例として用いるナノ粒子群形成アプリケーションについて述べる．3 章では，

†1 東北大学大学院情報科学研究科
Graduate School of Information Sciences, Tohoku University
†2 東北大学サイバーサイエンスセンター
Cyberscience Center, Tohoku University
†3 科学技術振興機構戦略的創造研究推進事業
Japan Science and Technology Agency, Core Research for Evolutional Science and Technology

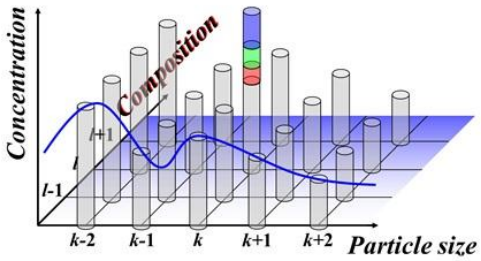

図 1 coagul ルーチンの概要

複合型計算システム向けのソフトウェア開発環境として，OpenCL および OpenACC を紹介する．4 章では，ナノ粒子群形成アプリケーションを OpenACC および OpenCL で実装し，そこで用いられる最適化技法について述べる．5 章では性能評価を行い，それぞれの最適化技法による性能への影響を明らかにする．6 章では結論と今後の課題を述べる．

2. ナノ粒子群形成アプリケーション

ナノ粒子群形成アプリケーションは，熱プラズマを用いたナノ粒子群創製プロセスにおける集団的粒子形成過程を数理モデル化したもので，これまで未知であった合金ナノ粒子群の成長過程を具体的にとらえることに世界で初めて成功したモデルである[6]．計算過程には，二成分核生成理論，二成分核凝縮理論，および異組成粒子間の凝集現象が組み込まれ，これらを数値的に解くため，二方向ノーダル解法という計算手法を用いている．実行時間が非常に長いアプリケーションであり，様々な粒子創製条件や材料の組み合わせに対する数値実験を行うためには，シミュレーションの高速化が必要とされている．

実行時間の約 9 割を占める coagul ルーチンは，図 1 に示す 2 次元空間に展開された各点において粒子間凝集を計算するルーチンである．各点の計算は独立に計算可能であり，並列度の高いデータ処理であるため，coagul ルーチンを GPU で処理することにより高い性能向上が期待できる．

3. 複合型計算システムのソフトウェア開発環境

3.1 OpenCL

OpenCL は，Khronos によって策定されているベンダ非依存の複合型計算システム向け標準プログラミングフレームワークである．OpenCL は特定のベンダに依存しないプログラミング環境であるため，NVIDIA の GPU に依存する CUDA 等に比べて，可搬性の高いプログラムを記述することが可能である．

OpenCL で記述されるプログラムは，ホストである CPU において実行されるホストコードと，デバイスである GPU において実行されるデバイスコードから成る．デバイスコードは複数のカーネルから構成されている．OpenCL は，複数の演算ユニットが一つのデバイスコードを並列に実行する SPMD(Single-Program Multiple-Date) 方式である．SPMD 方式の並列処理を行うために，OpenCL では演算ユニットで実行される処理を **work-item** として定義している．work-item は階層的にまとめて管理されており，1 から 3 次元的に配置することが可能である．複数の work-item をまとめて **work-group** として扱い，すべての work-group を **NDRange** という単位で扱う．

また，SPMD 方式の並列処理を記述するために，各 work-group および各 work-item を示す識別子である**インデックス**が用いられる．work-group のインデックスを work-group ID，work-item のインデックスを work-item ID と呼ぶ．それぞれの work-item が work-item ID に基づいて計算対象のデータを決めることにより，SPMD 方式の並列処理を実現する．work-item ID や work-group ID は，カーネルがホストコードから起動された際に決定される．また，カーネル起動時に生成される work-item 数や work-group サイズは，ホストコードで明示的に指示される．

ホストとデバイスはそれぞれ別のメモリ空間をもっており，ホストコード上でデバイスメモリの確保やホストとデバイス間のデータ転送など，カーネルを実行するために必要な処理を記述する．さらに，OpenCL のデバイスは特徴の異なる複数のメモリ空間を利用することが可能であり，それらを適材適所で使い分けることが，高い性能向上を達成するためには必要である．

プライベートメモリは，高速にアクセス可能であるが容量が非常に小さく，プログラマが明示的に使用することはできない．ローカルメモリは，容量が小さく高速なオンチップメモリであり，プログラマが明示的に確保することで使用することができる．再利用性のあるデータをローカルメモリに格納することで，オフチップメモリへのアクセス回数を削減できるため，ソフトウェアマネージドキャッシュ（Software-managed Cache）とも呼ばれる．オフチップメモリであるグローバルメモリは，デバイス内で唯一ホストからのデータの読み書きが可能なメモリ空間であり，容量は大きいが低速である．ホストとデバイス間でデータ転送を行うためには，あらかじめグローバルメモリにデータを格納しなければならない．

表 1 OpenCL と OpanACC の並列性の対応

OpenCL	OpenACC
work-group	gang
work-item	worker

GPU を利用しない既存のプログラムを GPU で処理されるカーネルに変換するためには，各データを明示的に OpenCL のメモリに割り当て，並列化したいループの変数を work-item ID に置き換える必要がある．このため一般に OpenCL で記述したプログラムは，元のプログラムと比較してプログラム行数が増加する．

3.2 OpenACC

既存のプログラムを複合型計算システムへ移植するアプローチの一つとして，C や Fortran で記述されたプログラムにディレクティブを追記することにより GPU を利用するアプローチが主流になりつつある．OpenACC は現在標準化が進められている GPU 等のアクセラレータのためのディレクティブの規格である．

OpenACC では，ディレクティブにより gang，worker，vectors の 3 段階の並列性が存在する．gang は OpenCL の work-group に対応し，worker が work-item に対応している．それぞれの並列性の対応関係を表 1 に示す．ディレクティブには，並列処理を指示するディレクティブのほかに，CPU と GPU 間のデータ転送等のデータの管理を行うディレクティブも用意されている．OpenACC を用いて GPU を利用するためには，プログラム上で GPU に処理を割り当てたい部分にディレクティブを挿入する．GPU に処理を行わせたい各ループに，gang，worker を指定することで GPU の演算ユニットに割り当てることが可能である．gang や worker の物理的な割り当ては各コンパイラに依存する．gang，worker の数をそれぞれディレクティブにより指示することで，処理の並列度を明示的に指示することができる．

OpenCL では，初期化・メモリ確保等の記述をする必要があるが，OpenACC を用いた場合ではディレクティブで GPU 上においてデータが有効な範囲を指示するだけでよい．また，GPU 上での処理も元のプログラムに数行のディレクティブを追記し，gang や worker に処理を割り当てるだけでよいため，移植作業に要する時間は大幅に短縮される．

4. OpenACC および OpenCL を用いた実装

本論文では実用的なアプリケーションの例としてナノ粒子群形成アプリケーションを考え，OpenACC と OpenCL により実装を行い，各最適化を施す．

```
1  // COMPUTATION KERNEL OF COAGUL ROUTINE    //
2  !$ACC UPDATE DEVICE(DNP,DNPOLD,BETA, ・・・)
3  !$ACC KERNELS
4  !$ACC LOOP GANG (NFRAC)
5  DO N=1,NFRAC
6  !$ACC LOOP WORKER (NNODE)
7  DO K=2,NNODE
8
9     DO M=1,NFRAC
10       DO L=1,NFRAC
11          DO J=2,NNODE
12             DO I=2,NNODE
13                IF ( ・・・) THEN
14                   DDNPCO=DDNPCO+BETA(I,J,L,M)* ・・
15                END IF
16             END DO
17          END DO
18       END DO
19    END DO
20
21    DO L=1,NFRAC
22       DO I=2,NNODE
23          IF ( ・・・) THEN
24             DDNPCO=DDNPCO−BETA(I,K,L,N)* ・・・
25          END IF
26       END DO
27    END DO
28
29    DNP(K,N)=DNP(K,N)+DDNPCO
30    IF (DNP(K,N).LT.SMALL) DNP(K,N)=ZERO
31
32 END DO ! K
33 END DO ! N
34 !$ACC END KERNELS
35 !$ACC UPDATE HOST(DNP)
```

図 2 OpenACC のディレクティブを追記した coagul ルーチン

表 2 opt0 におけるプログラム行数

	全体の行数	GPU に関する行数
OpenACC	2963	32
OpenCL	3330	152

4.1 OpenACC と OpenCL を用いた実装の概要

ナノ粒子群形成アプリケーションの CPU 実行時に，実行時間の約 9 割を占める coagul ルーチンを GPU を用いて実行する． OpenACC を用いる場合には，GPU で実行する処理を，図 2 の 3 行目と 34 行目のようにディレクティブを追記することによりコンパイラに指示する．さらに，各 gang と worker に割り当てるループを 4 行目と 6 行目のようにディレクティブを追記し，それぞれ gang と worker の数を指示する．また，各カーネルの実行に必要なデータと計算結果は 2 行目と 35 行目のように指示することで CPU と GPU 間のデータ転送が行われる．指示しない場合はコンパイラがカーネルに必要なデータを判断し，各データの CPU と GPU 間のデータ転送が行われる．

OpenCL では，work-item ID をそれぞれループ変数である N と K に割り当てることで並列処理を記述

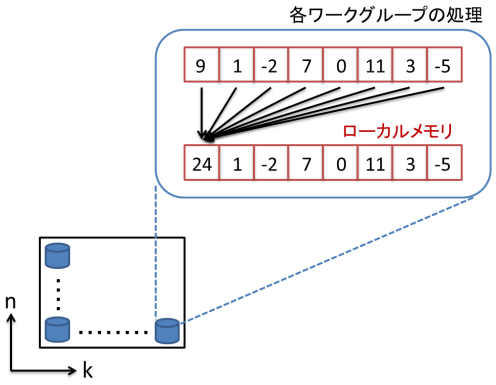

図 3 各点 (K,N) へのリダクション

する．また，OpenACC と同様にカーネルの実行に必要なデータの確保や CPU と GPU 間のデータ転送を OpenCL の API で記述する必要がある．

図 2 の単純な実装を opt0 とする．opt0 における OpenACC と OpenCL のプログラム行数を表 2 に示す．OpenCL による実装では，Fortran のプログラムを C に変換しているため OpenACC に比べ行数が多い．GPU で実行される処理と CPU と GPU 間のデータ転送に関するプログラムの行数だけを比較すると，OpenACC は GPU の初期化等を明示的に記述する必要がないため，OpenCL に比べて非常に少ない行数で同様の実装が可能である．

4.2 データ転送の最適化

図 2 の 14 行目と 24 行目の計算で使用される BETA 配列は，全体のイタレーションループ毎に更新されるため，イタレーションループ毎に CPU と GPU 間の転送が必要である．元のプログラムでは，別のルーチン (coeffs) 内で 40KB のデータに基づいて 42MB の BETA 配列が更新されている．この coeffs ルーチンに関しては，GPU よりも CPU の方が高速に処理を実行可能である．しかし，GPU ではデータ転送よりも演算のコストのほうが低いため，42MB のデータ転送よりも，BETA 配列を更新するコストのほうが低い．このため，GPU で coeffs ルーチンを実行し，BETA 配列を更新することでデータ転送コストを削減する．

また，明示的にカーネルで使用される各配列の要素を指示しない場合，コンパイラの判断により各要素が CPU と GPU 間で転送される．不必要な配列要素を転送しないよう明示的に指示，または変数に代入するすることで CPU と GPU 間のデータ転送を必要最小限とすることが可能となる．この最適化を施した実装を opt1 とする．

表 3 各 work-group サイズでの occupancy

C2070	occupancy	C1060	occupancy
54x1	0.333	54x1	0.375
54x2	0.667	54x2	0.375
54x3	0.750	54x3	0.375
54x4	0.583	54x4	0.375
54x5	0.562	54x5	0.375
54x6	0.688	54x6	0.375
54x7	0.750	54x7	0.375

4.3 メモリアラインメント最適化

GPU にはオフチップメモリへのメモリ転送を効率的に行うコアレッシングという機能が存在する．本論文で想定する現行の NVIDIA 社製 GPU では，コアレッシングにより 32 バイト，64 バイト，128 バイトという単位でメモリアクセスが行われる．

本論文では，各配列に余分な要素を追加し，第一次元方向の要素数が 16 の倍数となるように調整する (パディング)．その結果，配列要素へのアクセスがすべて 16 個単位で行われるようになる．この最適化は OpenACC と OpenCL の両方において同様に実装可能である．この最適化を施した実装を opt2 とする．

4.4 リダクションの最適化

元のプログラムではイタレーション数が短いため，worker 数や work-item 数が少なくなり，性能低下の原因になっている．opt0 では gang 内の worker 数をイタレーション数に合わせ，各イタレーションの処理を worker に分担している．そのため，イタレーション数以上に worker 数を増加するためには，K と N 以外のループを worker に割り当てる必要がある．

図 2 において，K と N 以外のループでは各計算結果を集約計算（リダクション）して各点（K,N）に書き込む．よって，これらのループでは依存関係があり，これらのループのうち一つを worker 毎に並列に行った場合，そのループでの計算結果をリダクションする際に，各 worker の排他的な処理が必要となる．

OpenCL においては，図 3 に示すように work-group 毎に共有可能なローカルメモリを用いて，work-item ID により各 work-item の排他的な処理を細かく記述することで，各 work-group 内の work-item で並列にリダクションを行うよう実装することが可能である．OpenCL において，work-group 内の work-item 毎にリダクションを行うよう最適化を施した実装を opt3 とする．

なお，現在の OpenACC のディレクティブでは，gang や worker のインデックスの取得ができない．そのため OpenCL と異なり，gang 内の worker でリダクションを行い，gang 単位で並列に各点に結果を書き込むよう

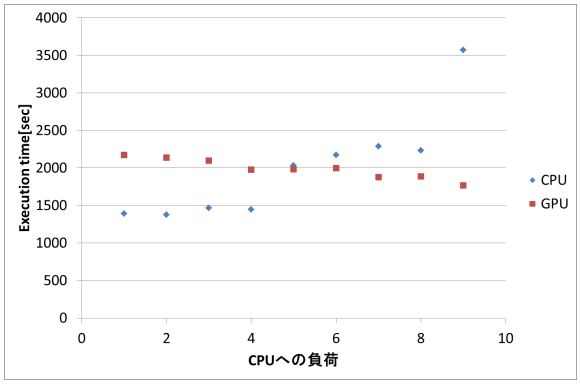

図 4 CPU と GPU の負荷比率と実行時間の関係

な実装を行うことができない．OpenACC では，カーネルによるスカラ変数へのリダクションが実装されている．よって，図 2 のようなループの場合，カーネル毎にリダクションを行い，その結果をループ K と N 毎に各点に書き込むような最適化も可能である．しかしこの実装方針では，カーネルを多数起動する必要があり，カーネル毎に逐次的な処理となるため性能が低下してしまうことが予備実験より明らかになっている．

4.5 work-group サイズの最適化

opt3 において，図 3 のようにリダクションは最内ループ I で行われており，work-group 内の work-item 数である work-group サイズは，ループ I のイタレーション数となっている．そのため，他のループにおいてもリダクションを行うことで work-group サイズを大きくし，並列度を増加させることが可能である．より大きな work-group サイズではメモリアクセスが効率的になるが，プライベートメモリ等ハードウェアリソースの使用量が増加してしまう．よって，GPU のハードウェアリソース量を考慮して最適な大きさに調節する必要がある．

OpenCL のプロファイラでは，最適な work-group サイズを決定する手助けとなる occupancy を計算することができる．occupancy は，カーネル実行時に起動された work-item がどれだけ GPU のリソースを占有しているかを示す指標である．

GPU の世代毎にリソース量は異なるため，NVIDIA の Tesla C2070 と Tesla C1060 それぞれにおける各 work-group サイズ毎の ocuppancy を表 3 に示す．表 3 より，Tesla C2070 では work-group サイズを 54x3, Tesla C1060 では opt3 のままでよいことがわかる．このサイズの最適化を施した実装を opt4 とする．OpenACC でも，worker 数を調整することで並列度を調整することが可能である．しかし，今回のルーチンでは

表 4 各クラスタの緒元

	クラスタ 0	クラスタ 1
CPU	Intel Core i7 930	Intel Xeon 5570
GPU	NVIDIA Tesla C2070	NVIDIA Tesla C1060
OS	CentOS 6.0	RHEL 5.3
GCC	4.4.4	4.1.2
CUDA	4.1.1	4.0.1
OpenCL	1.1	1.0
OpenACC	HMPP 3.1.0	PGI 11.8

opt3 を施さない場合，最大でも 54 個の worker でしか処理を分担できないため，並列度を増加させることができない．

4.6 CPU への負荷分散

GPU でのカーネル実行中 CPU では処理が行われない．coagul ルーチン以外で coagul ルーチンと並列処理可能なルーチンの実行時間はカーネルの実行時間の 1%以下であり，並列に処理しても性能向上が期待できない．したがって，カーネルの計算領域を CPU にも割り当て，並列に処理させることでさらなる高速化が可能である．

計算領域を CPU と GPU とで割り当てる場合，CPU の実行時間が長くなる．各計算システムによって CPU と GPU の性能は異なるため，実行時間が一致するように計算領域を分割する必要がある．したがって，CPU と GPU での実行時間が同じになるように計算領域を分割し，CPU と GPU に割り当てる．図 4 に，Intel Core i7 930 と NVIDIA Tesla C2070 における CPU と GPU の実行時間を示す．横軸はループ N のイタレーション数のうち CPU で分担して処理するイタレーション数を示している．図 4 より，GPU と CPU への負荷の割当配分を 37:4 にしたとき最も高い性能を達成することが可能であることがわかる．各システム毎に負荷が均等になるよう OpenCL で計算領域を分割し，負荷分散した実装を opt5 とする．OpenACC でも，CPU と GPU で処理を並列実行することが可能であるが，並列度が不足するために本論文では OpenACC の opt3 以降を実装しなかった．

5. 性 能 評 価

評価には，二種類の GPU クラスタを用いた．各クラスタの諸元を表 4 に示す．

5.1 各最適化による効果

各クラスタでの評価結果を図 5，図 6 に示す．縦軸は，各クラスタでの CPU の実行時間からの性能向上率を示している．OpenMP によりクラスタ 0 では 4 コア，クラスタ 1 では 8 コアで CPU 実行されている．

図 5 より，Tesla C2070 を搭載したクラスタ 0 で実

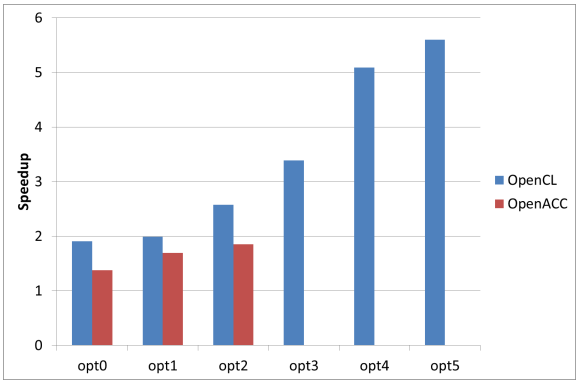

図 5 クラスタ 0 における性能向上率

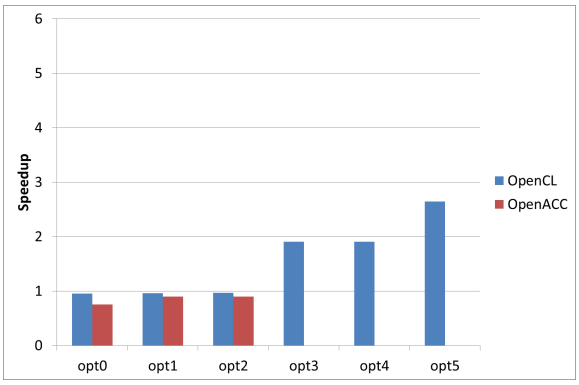

図 6 クラスタ 1 における性能向上率

行した場合は，OpenACC で実装したバージョン (OpenACC 版)，OpenCL で実装したバージョン (OpenCL 版) 両者において最適化毎に性能が向上している．しかし，OpenACC 版と OpenCL 版の性能を比較すると OpenACC 版は OpenCL 版に対して約 25%程度低い性能となっている．opt0 から opt2 において，OpenACC 版と OpenCL 版の処理は同一である．そのため，これらの性能差はコンパイラによって生成されるコードの性能が低いことを示している．現在各ベンダにより OpenACC のコンパイラの改善が進められているため，この性能差は今後低減していくと考えられる．

opt1 のデータ転送の最適化により opt0 と比較して，OpenACC 版では約 22%，OpenCL 版では約 4%の性能向上が得られた．CPU と GPU 間のデータ転送は GPU 実行時のオーバーヘッドであり，今回のように少ないデータ量からより大きなデータを生成する処理を GPU で実行することにより，データ転送量を削減することが可能である．OpenACC では，OpenCL に比べて少ないプログラミング負荷で処理をカーネルとして GPU に実行させることが可能であるため，このような場合には容易にデータ転送を削減可能である．

opt2 のメモリアラインメント最適化により opt1 と比較して，OpenACC 版では約 9%，OpenCL 版では約 29%の性能向上が得られた．この最適化は，元のプログラム上で行うことが可能な最適化であるため，OpenACC でも OpenCL と同様に最適化を行うことが可能である．今回のカーネルは，メモリアクセスのほとんどがグローバルメモリへのアクセスであるため，このような最適化が非常に効果的である．opt2 では，各配列のパディングを行う必要があるため，OpenACC による実装においても元のプログラムを修正する必要がある．この結果から，OpenACC においてもディレクティブによる指示だけではなく，このような元のプログラム上での最適化が必要であることがわかる．

opt3 の work-group 毎のリダクションにより opt2 と比較して，OpenCL 版では約 32%の性能向上が得られた．さらに opt4 の work-group サイズの調整により，約 50%の性能向上が得られた．現在の OpenACC の仕様では，OpenCL で実装可能であった opt3 のような最適化ができず，今回のように依存関係のあるループを並列化することができない．そのため，少ない worker 数での並列処理となり，GPU の演算性能を最大限引き出すことができない．

opt5 の CPU への負荷分散により，opt4 と比較して約 5%の性能向上が得られた．このように CPU と GPU に効率的に処理を割り当てることで，複合型計算システム全体の演算性能を最大限利用し性能向上させることができる．OpenACC においても非同期で GPU により処理を行うディレクティブを用いて，このような最適化を行うことが必要であると考えられる．

以上より CPU4 コアで実行した場合と比較して，OpenACC 版では最大 1.9 倍，OpenCL 版では最大で約 5.6 倍の性能向上が得られた．

図 6 より，NVIDIA Tesla C1060 を搭載したクラスタ 1 では，Tesla C2070 の場合と異なり opt2 で約 1%ほど性能低下している．Fermi 世代の GPU ではメモリアラインメントの調節のみでコアレッシングが効率化される．そのため，Fermi 世代より前の GPU においては opt2 によりメモリアクセスが効率化されない．よって，各世代の GPU 毎に最適化のアプローチが異なることを示している．さらに，opt3 の最適化を施さない場合，OpenACC 版，OpenCL 版ともに CPU で実行した場合よりも低い性能となっている．今回のカーネルは，グローバルメモリのアクセスが容易にボトルネックとなりやすく，キャッシュを搭載しない Tesla C1060 では GPU を利用しただけでは性能が引き出せない．その

ため，opt3 のような最適化が必要なことを示しているが，opt3 の実装に必要な OpenCL と同様の抽象度のディレクティブが存在しない OpenACC では，OpenCL と同様の性能を得ることができない．また，クラスタ 1 においても OpenACC 版と OpenCL 版の性能を比較すると，OpenACC 版は OpenCL 版に対して約 20%から 7%程度低い性能となっている．

以上より，OpenACC は非常に少ないプログラミング労力で GPU を利用可能であり，今回のようなアプリケーションの場合，データ転送の最適化も容易に適用可能であることがわかる．OpenACC においてメモリアラインメント最適化等が，実アプリケーションの複雑なカーネルにおいても効果的であることが示された．しかし，現状ではコンパイラの性能等により，同様の処理を行うよう OpenCL で実装した場合の約 25%から約 7%程度低い性能しか達成できていない．さらには，高い性能向上が得られた work-group 毎のリダクションの最適化も，OpenACC の抽象度では実装困難である．そのため，最大でも OpenCL で最適化を施した場合の約 36%の性能までしか達成することができなかった．よって，OpenACC ではある程度の性能向上を達成することが可能であるが，GPU の演算性能をより引き出すためには OpenCL のような低い抽象度での最適化が必要であることが示された．

6. ま と め

本論文では，実アプリケーションであるナノ粒子群形成アプリケーションを OpenACC と OpenCL を用いて実装した．それぞれの実装について最適化を行い，メモリアラインメント最適化等により，OpenACC を用いた場合には CPU 実行時と比較して最大で約 1.9 倍の性能向上が得られた．しかし，OpenCL で実装可能な work-group 毎のリダクションの最適化を OpenACC では実装することができないため，OpenCL で最適化を施した場合の約 36%の性能までしか達成することができなかった．さらに，GPU のハードウェア世代によるキャッシュ制御の有無等により，CPU 実装より低い性能となる場合があることが示された．以上より，実アプリケーションのような複雑なカーネルでは OpenCL のような低い抽象度での最適化が必要であり，現状の OpenACC では容易に高い性能を達成することが困難である．

OpenACC は少ないプログラミング労力でアプリケーションの GPU コード開発が可能になる．しかしながら，実アプリケーションの複雑なカーネルを現状の OpenACC で実装した場合，その性能は OpenCL で実装した場合より低くなることがわかった．今後の GPU のハードウェアによる支援やコンパイラの最適化により，性能についても改善されることが期待される．

謝　　辞

ナノ粒子群生成アプリケーションのプログラムを提供していただき，計算内容について様々なご助言をいただいた東北大学の茂田正哉助教に感謝します．

本研究の計算結果の一部は，RIKEN Integrated Cluster of Clusters (RICC) システムを利用して得られた．また本研究の一部は，戦略的創造研究推進事業 (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「進化的アプローチによる超並列複合システム向け開発環境の創出」の助成を受けている．

参 考 文 献

1) NVIDIA Corporation. *NVIDIA CUDA Programming Guide 3.0*, 2010.
2) Khronos OpenCL Working Group. The OpenCL Specification version 1.1. http://www.khronos.org/opencl/.
3) The OpenACC pplication Programming Interface 1.0. http://www.openacc-standard.org/, 2011.
4) R.Dolbeau et al. HMPP: A Hybrid Multicore Parallel Programming Environment. *Workshop on GPGPU 2007*, 2007.
5) ThePortland Group. PGI Accelerator Programming Model for Fortran & C. http://www.softek.co.jp/SPG/Pgi/Accel/, 2010.
6) Masaya Shigeta and Takayuki Watanabe. Growth model of binary alloy nanopowders for thermal plasma synthesis. *Journal of Applied Physics*, Vol. 108, No.4, pp. 043306 –043306–15, 2010.