Vol.2012-HPC-133 No.12
2012/3/26

複数 GPU 向けの CUDA コードを生成する OpenMP 処理系の提案

長塚 郁[†1] 大島聡史[†2] 平澤将一[†3]
近藤正章[†1] 本多弘樹[†1]

著者らは OpenMP プログラムから CUDA プログラムへ変換する処理系,“ OMPCUDA ”の開発を行っている．本稿では，OMPCUDA における複数 GPU 向けの CUDA プログラムを生成するための機能の実装を述べ，生成された CUDA コードの評価結果について考察する．

1. はじめに

汎用計算に GPU を用いる GPGPU は，その価格性能比やエネルギー性能比が高いことから，科学技術計算を中心として広く用いられるようになってきている[1)2)]．そして，GPGPU プログラムの開発に際して，現状では CUDA[3)] を用いることが多い．

このような中で，我々は CPU 並列プログラミングの経験者が CUDA の知識を習得せずとも GPGPU を行ったり，CPU 向けの並列プログラムを簡単に GPU で実行できるようにすることを目指して，OpenMP プログラムから単一 GPU で実行される CUDA プログラムへ変換する処理系“ OMPCUDA ”を開発してきた[4)5)]．

本研究では，OMPCUDA を複数 GPU 向けに拡張を行い，OpenMP プログラムから複数 GPU 向けの CUDA プログラムへ変換する処理系“ OMPCUDA/MG ”を開発し，その評価を行う．

2 章では単一 GPU 向けの OMPCUDA について述べ，3 章では OMPCUDA/MG の実装を述べる．4 章では本提案のコンパイラで変換したソースコードの性能評価と考察を行い，5 章で関連研究について触れ，6 章でまとめる．

†1 電気通信大学 大学院情報システム学研究科
Graduate School of Information Systems The University of Electro-Communications
†2 東京大学 情報基盤センター
Information Technology Center, The University of Tokyo
†3 電気通信大学/JST
The University of Electro-Communications/JST

2. 単一 GPU 向け OMPCUDA

初期の OMPCUDA は単一 GPU を対象とし，OpenMP 処理系 Omni OpenMP Compiler[6)](以下 Omni) をベースに開発した，OpenMP プログラムから CUDA コード生成するトランスレータである．OMPCUDA は OpenMP と単一 GPU 向けの CUDA のメモリモデルと並列処理モデルの相違点を対応付け，プログラマが OpenMP を用いて記述したプログラムから GPU 上で実行可能な CUDA プログラムに変換を行う．OMPCUDA はその変換のための“ プログラム変換機構 ”と並列実行時に必要となる機能を提供する“ 実行時ライブラリ ”から成り立っている．

OMPCUDA では，多くの OpenMP による大規模並列アプリケーションプログラムでは forall ループが大部分である点，forall ループ処理の並列化は 1GPU Thread 当たりの処理が同じになりやすく，かつ並列性が高くなりやすいため，GPU 上で並列実行する問題に適しているという点から，GPU での並列実行対象を forall ループのみとしている．

Omni では並列実行部を専用の関数に書き出すことで，逐次処理との区別を行っている．OMPCUDA のプログラム変換機構ではこの関数をさらに GPU カーネル関数へ変換し，逐次処理部分に書かれた並列実行部関数呼び出し部分を Global Memory の確保やデータ転送，GPU カーネル関数呼び出しなどの GPU を制御する一連の処理に置き換える．

Omni における forall ループの並列化は実行時ライブラリが行う．各 CPU スレッドが自らのスレッド ID を元に新たな部分ループを作成する関数を用いて行われている．OMPCUDA においても，GPU Thread ID と GPU Block ID を用いて GPU 上で実行される forall ループのスケジューリングを行う関数が実装されている．その際，OMPCUDA ではループのイタレーションをブロック分割した静的なスケジューリングのみを提供している．これは，動的なスケジューリングの場合では，各 GPU Block 内の GPU Thread 同士および各 GPU Block 間の GPU Thread 同士の排他制御および待ち合わせによる性能低下の可能性があるためである．

また forall ループの並列化とともに利用されることが多いリダクション演算の中で，各 GPU Thread の総和を求める演算の実装されている．

Vol.2012-HPC-133 No.12
2012/3/26

3. 複数 GPU 向け CUDA ソースコードを生成するための実装

3.1 メモリモデルの対応付けと実装

並列実行部のメモリモデルの対応付け

OpenMP におけるメモリモデルはメインメモリを CPU スレッドの共有メモリとした共有メモリ型であり，複数の CPU スレッドは同一の変数に対して同一のアドレスで参照をすることができる．

CUDA では単一 GPU 内においては，各 GPU Thread は Global Memory に対して同一の変数に対して同一のアドレスで参照をすることができる．

一方，複数 GPU 向け CUDA では異なる GPU のお互いの Global Memory を直接参照することはできないため分散メモリ型となる．そのため，GPU 間のデータの授受は CPU のメインメモリを介する必要がある．

並列実行部と逐次実行部間のメモリモデルの対応付け

OpenMP では逐次処理部と並列処理部で同じメモリ空間を共有している．単一 GPU の CUDA では CPU で実行される逐次処理実行部と GPU で実行される並列処理部においては Global Memory と CPU のメインメモリではメモリ空間が独立している．加えて，複数 GPU 向けの CUDA の場合では各 GPU の Global Memory 間ではメモリ空間が独立している．

実　　装

複数 GPU 向け CUDA でも全てのデータをメインメモリに格納しておくことで，CPU 側のメインメモリを各 GPU の共有メモリとした共有メモリ型とみなすことができ，対応付けは可能である．本研究では各 GPU に転送する CPU のデータの分割は行わず，全てのデータをコピーすることで，上記の対応付けに対処することとする．

その際，GPU 上で計算した結果を CPU のメインメモリへ転送する必要があるが，担当した処理の部分だけを転送しなくてはならないため，各 GPU の並列処理後，GPU 上で計算を行った部分のみを転送して CPU のメインメモリと Global Memory および各 GPU 間のデータのコヒーレンシを保つこととする．

3.2 並列処理モデルの対応付けと実装

並列処理モデルの対応付け

OpenMP と複数 GPU 向け CUDA の並列処理モデルでは，両者の並列処理階層モデルおよびイタレーションの割り当て方法に相違点がある．

図 1　OpenMP の並列処理階層モデル

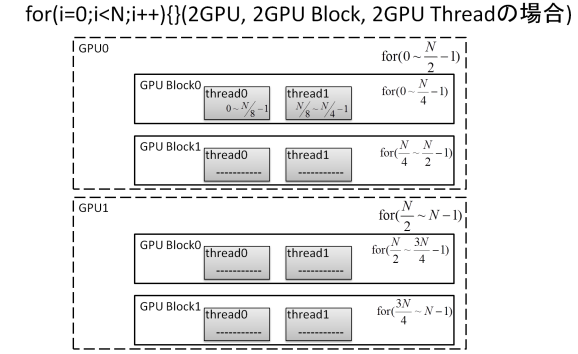

図 2　複数 GPU 向け CUDA の並列処理階層モデル

今回想定する OpenMP の並列処理階層は複数の CPU スレッドによる一階層並列処理である (図 1)．

CPU が forall ループを並列実行する場合，イタレーションを各 CPU スレッドに直接割り当てられる．どのようにイタレーションを分割して CPU スレッドに割り当てるかが条件命令でプログラム中に記述され，それに合わせてコンパイラによって分配が行われる．

一方，複数 GPU 向けの CUDA の並列処理階層は複数の GPU とその GPU 上で用意された GPU Block，その GPU Block に含まれている GPU Thread による三階層となっている (図 2)．

まず，各 GPU へのイタレーションの割り当て方式は，静的なブロック分割のみとした．

これは大きなオーバーヘッドとなる CPU-GPU 間のデータ転送回数とカーネル関数呼び出し回数をできるだけ減らすためである．

次に，各 GPU Block および各 GPU Thread へのイタレーション割り当て方式は，静的なブロック分割およびサイクリック分割のみとした．これは初期の OMPCUDA の単一 GPU における GPU Block および GPU Thread に対するイタレーション割り当て方式を継続して使用するためである．

実　　装

各 GPU Block と GPU Thread への割り当ては実行時ライブラリ別に割り当て方法を変更できるようにすることとする．入力される OpenMP プログラムの指示子でスケジューリング法の指定がない場合，生成された CUDA コードでのイタレーション割り当てはブロック分割で行われる．forall ループ指示子内で schedule(static, 1) が指定された場合はサイクリック分割で行われるように実装した．

各 GPU が GPU カーネル関数を実行する際，実行時ライブラリに用意されているスケジューリング関数によって GPU Block, GPU Thread が担当するイタレーションを決定する．forall ループのスケジューリング (割り当て) のブロック分割の場合の一例を図 4 に，サイクリック分割の場合の一例を図 5 に示す．1 つの GPU Thread に複数のイタレーションを割り当てる際にブロック分割の場合ではコアレスアクセスせず，サイクリック分割の場合ではコアレスアクセスすることができる．

これらのスケジューリングで使われる GPU 数，GPU Block 数，GPU Thread 数の設定は環境変数や実行時のコマンドライン引数で指定できるように実装をした．

また，1CPU スレッドで対象とする GPU の数は一つまでと限定されている．そのため，複数の GPU を動かすためには各 GPU の処理を行う同数の CPU スレッドが必要となり，その CPU スレッドの制御のためにマルチスレッド化が必要となる．そのマルチスレッド化を行うようにプログラムを変換するように実装をした．

3.3 プログラムの変換機構

OMPCUDA が OpenMP プログラムから複数 GPU 用の CUDA プログラムへ変換するための手順は以下のとおりである．手順 1 と 2 が既存の Omni の担当処理で，手順 3 以降は OMPCUDA による処理である．また，プログラムの変換の流れの概略図は図 3 に示す．

(1) OpenMP プログラムを入力として受け取り，各言語に対応した Frontend によって OpenMP 指示子の入った中間コード (Xcode) へと変換する．
(2) OpenMP 指示子の挿入された中間コードに対して，OpenMP モジュールを用いて指示子の解釈を行い，Omni の実行ライブラリを含む中間コードを生成する．この際，OpenMP における並列実行部は関数として分離され，並列実行部関数をスレッドに割り当てる Omni の実行時ライブラリ関数によって呼び出されるように書き換えられる．
(3) 中間表現で書かれたプログラム全体から実行時ライブラリの並列実行開始関数が呼び出されている部分を探し出す．
(4) 並列実行部の内部で利用されている変数を確認し，関数内ローカルではない変数，すなわち CPU-GPU 間で送受信を行う必要がある変数を特定する．また Omni の並列実行開始関数は並列実行部で利用する呼び出し元のローカル変数も引数として受け取る使用のため，引数を確認して送受信を行う必要があるローカル変数を特定しておく．
(5) 並列実行部を別ファイルに書き出し，元のソースコードから削除する．並列実行部に関数呼び出しがある場合はその関数内で利用している変数についても同様に確認し，まとめて書き出す．ただし，並列実行部で呼ばれている関数は逐次実行部でも呼び出されている可能性があるので，元のソースコードにも残したままとしている．
(以下，別ファイルに書き出したソースコードを GPU コード，残されたソースコードを CPU コードと呼ぶことにする．GPU コードは並列実行部の数だけ存在する)
(6) Omni の並列実行開始関数の呼び出しを GPU デバイスメモリの確保，CPU-GPU 間のデータ送受信および GPU カーネル関数の呼び出しを行う各 CPU スレッドの動作処理に書き換える．これを nvcc でコンパイルし，OMPCUDA の実行時ライブラリとリンクして CPU 用の実行ファイルを生成する．
(7) GPU コードには，CUDA の記述に従い，関数や変数に指示子 (__global__ など) を追加する．更に関数の引数を調整し，CPU-GPU 間で変数の送受信が行えるようにした上で，CUDA コンパイラ (nvcc) を利用して CUDA 用の実行ファイルを生成する．このプログラム生成の流れに沿って，OpenMP プログラムより複数 GPU を動作させるための各処理を行う CPU スレッドの動作を含む CUDA プログラムを生成する．この際，生成される CPU スレッドの数は起動させる GPU の数と同数である．

3.4 リダクション演算の実装

単一 GPU 向け OMPCUDA において GPU におけるリダクション演算の方法に Owens らの手法[7] をベースに Shared Memory を用いた並列化ライブラリ関数を提供しており，ハンドコーティングによる CUDA プログラムに近い実行時間を実現した．本稿の OMPCUDA においても 1GPU デバイス内のリダクション演算にはこの並列化ライブラリ関数を用いて

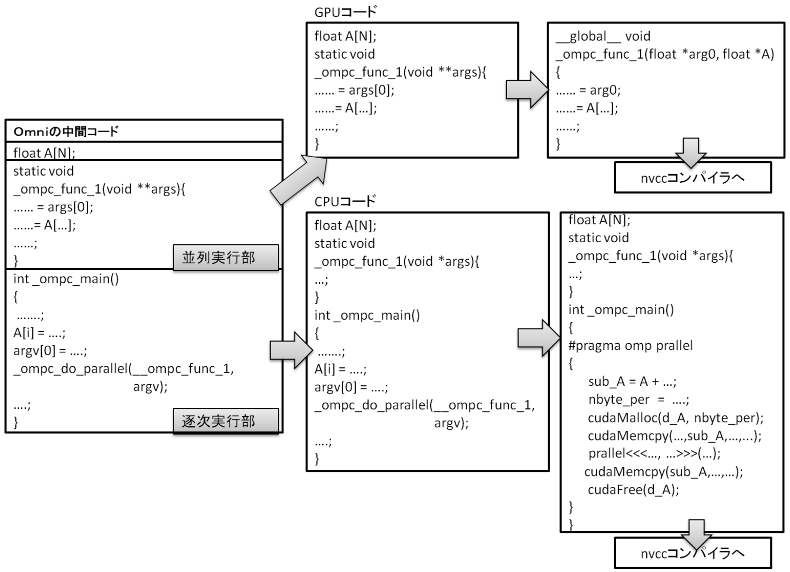

図 3 プログラム変換の流れ

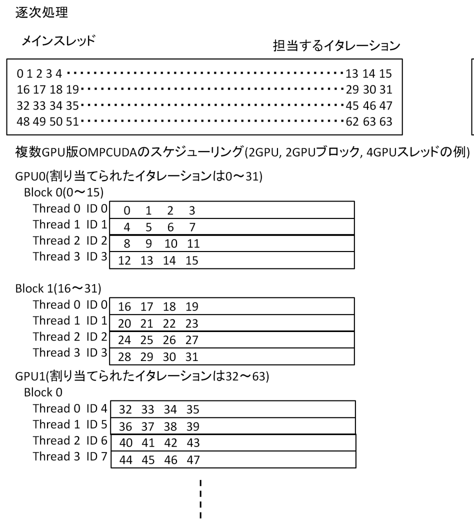

図 4 ブロック分割の例

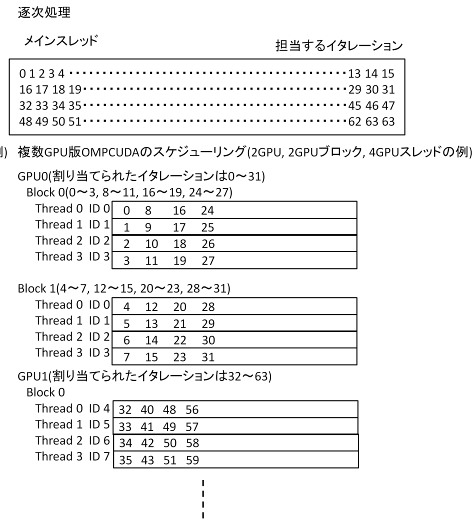

図 5 サイクリック分割の例

行うこととした．

リダクション演算において複数 GPU を用いた場合に問題となるのは，各 GPU で求めた結果をどのように集計を行うかである．これに対しトーナメント方式で GPU 上で足し合わせを行う，もしくは CPU 上で足し合わせるの二つの方法が挙げられる．足し合わせの処理自体はそれほど重いものではなく，GPU 上で実行するほどの並列度もない．GPU へデータを再転送するコストを考えれば，CPU 上で足し合わせる方法のほうが妥当である．

そのため各 GPU デバイスのリダクション演算の実行が終わった後，各 GPU へ割り当てられた一時保存変数に結果を格納し，CPU 上でそれらの足し合わせることにする．現在は加算のみを実装している．

4. 評価と考察

この章では複数 GPU 向けに拡張した OMPCUDA の性能評価実験を行う．評価は OMPCUDA の実行時間 (以下 OMPCUDA) を Omni でコンパイルした並列実行コードを複数 CPU スレッドで実行した場合 (以下 Omni)，ハンドコーティングによる CUDA プログラム (以下 SimpleCUDA) を CPU と GPU で実行した場合の実行時間と比較する．

表 1 評価環境
Table 1 Evaluation environment.

CPU	Intel(R) Xeon(R)CPU X5550(4 コア) 2.67GHz×2
メインメモリ	4.0GB DDR3-1333-4GB ECC Reg
GPU	TeslaT10Processor(240 コア)×4 コアクロック 1.44GHz
ビデオメモリ	GDDR3 4GB×4
GPU 接続バス	PCI-Express Gen2×16 ×2
コンパイラ	nvcc 3.2 V0.2.1221

評価環境は表. 1 のとおりである．同時に使用できる CPU スレッド数は Hyper-Threading による 16 スレッドまでとなり，今回は 4 スレッド，8 スレッド，16 スレッドの場合の結果を取った．GPU は 1 個の PCI-Express Gen2×16 から 2 個の GPU に接続されている．本研究において同時に使用できる GPU 数は 4 個となる．

4.1 行列積プログラム

実験対象プログラムとして使用する行列積計算のプログラムを図 6 に示す．グラフ内の接尾辞の n は，CPU スレッドおよび GPU の数を示している．本研究ではコードチューニ

```
//2重ループによる行列積のソースコード
#define SIZE 8192
float a[SIZE*SIZE],b[SIZE*SIZE],c[SIZE*SIZE];
#pragma omp parallel for private(i,k)
for(j=0; j<SIZE*SIZE ;j++)
{
    float tmp=0.0f;
    for(i=0; i<SIZE ;i++)
    {
       tmp += a[(j/SIZE)*SIZE+i] * b[i*SIZE+(j%SIZE)];
    }
    c[j] = tmp;
}
```

図 6 行列積のソースコード

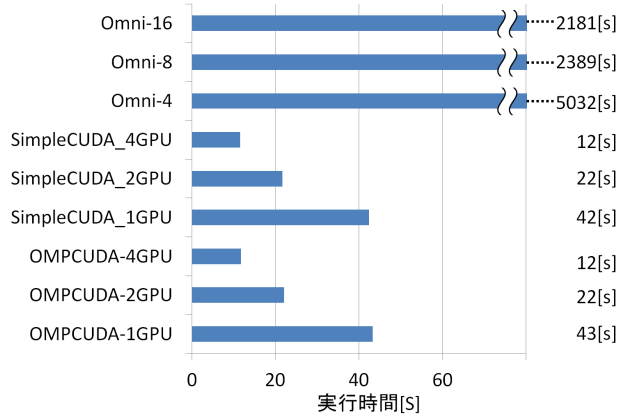

図 7 行列積の結果

ングによって最大性能を得ることよりも単純な実装で容易に高性能を得られることを重視するため，単純なループ処理によるコードを用いて実行時間の比較を行った．行列積は全て単精度浮動小数点の転置無し正方行列による行列積とし，行列積の一辺のサイズを問題サイズとし，8192 に固定した．

実行結果を図 7 に示す．OMPCUDA で変換したソースコードの実行時間は Omni による CPU プログラムの実行時間より大幅に短くなっている．

また OMPCUDA の実行時間は SimpleCUDA の実行時間とほぼ同等となった．

```
//グレゴリ級数による円周率求算プログラム
#define SIZE 1024*1024*1024
float answer = 0.0;
#pragma omp parallel for reduction(+:ansewer)
for(int i=0;i<SIZE;i++)
{
        answer += (4.0 / (4.0 * i + 1.0) – 4.0 / (4.0 * i + 3.0));
}
```

図 8 グレゴリ級数による円周率求算のソースコード

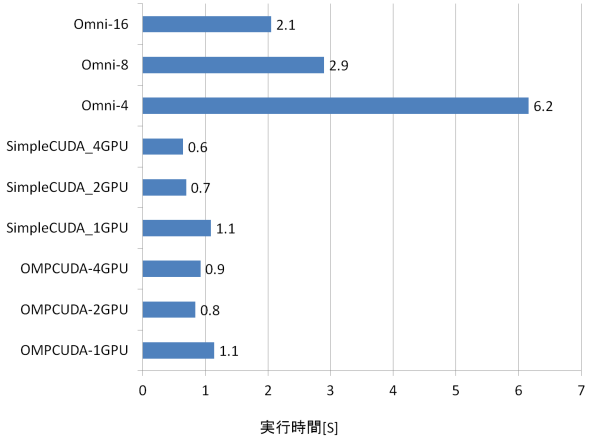

図 9 グレゴリ級数による円周率求算の結果

4.2 グレゴリ級数による円周率求算

グレゴリ級数による円周率求算は計算に必要なデータ量が少なく，リダクション演算以外は理想な並列度を持つ問題として知られている．そのため，並列化による高速化が容易な問題である．変換を行う対象のプログラムは図 8 に示す．

実行結果を図 9 に示す．OMPCUDA の実行時間は Omni の実行時間より短いことが分かる．一方 OMPCUDA の実行時間はハンドコーティングによる CUDA プログラムの実行時間よりは長くなってしまっている． OMPCUDA における各 GPU 数の結果を比較すると，2GPU の場合が最も実行時間が短縮できている．そこで，4GPU の実行時間が長い原因を探るべく，1GPU の場合と 4GPU の場合のそれぞれの OMPCUDA で変換したプログラム中の各処理の実行時間を測定した．4GPU の場合では各 CPU スレッド内でタイマーを

動かして実行時間を測定している．それぞれの結果を図 10 と図 11 に示す． 最も時間が長いのは Global Memory (GPU デバイスメモリ) の確保の処理であり，その処理の最も処理時間の長かった結果を比べると 1GPU に対して 2GPU では 2 倍，4GPU では 4 倍と使用した GPU の数に比例して増加している事がわかった．

4.3 近傍計算の繰り返し処理

前述のグレゴリ級数の円周率求算プログラムによる実行時間評価で Global Memory の確保の処理のオーバヘッドが大きい事が判明した．現状では図 12 のような並列処理の繰り返し処理を変換すると，Global Memory の確保の処理なども繰り返し実行してしまうため，実行時間が長くなっていると考えられる．

実際にはこれらの処理を繰り返し処理の外へと出し，一度実行すればよい．そこで，図 12 のプログラムを変換したのち，ハンドコーティングによってこれらの処理を繰り返し処理の外へと出した場合と比較を行った。 実行結果を以下の図 13 に示す．

メモリ確保をループの外も出した場合の実行時間は GPU が複数個になるにつれて短縮されていることが分かる．更に，ループの外へ出す前では 1GPU の方が実行時間が短かったのに対し，ループの外へ出した後では 4GPU が最も実行時間が短いことが分かる．ただし，どちらの場合においても Omni のほうが実行時間が短いため，更に実行時間を短くできるような仕組みを考える必要がある．

5. 関 連 研 究

本研究の提案する複数 GPU 向け CUDA コード生成を行う OpenMP 処理系 OMPCUDA/MG は，OpenMP を用いてプログラムを行ってきたプログラマが，その環境のまま複数 GPU を用いることができるようにすることに重点を置いている．

OpenMP プログラムから CUDA ソースコードへの変換についての関連研究として OpenMPC が挙げられる[8]．OpenMPC ではコード変換コンパイラである Cetus Comipler を用いて，OpenMP のソースコードの解読と CUDA ソースコードへの変換を行っている．また，OpenMPC 独自の指示子や条件付け命令などを追加することによって，プログラマがコンパイラではできないような詳細な最適化を行うことができるようになっている．それにより，基となる OpenMP と比較してプログラムは複雑となるが，人間の手によって書かれたソースコードにより近い性能を出すことに成功している．

また，OpenMP のように並列処理対象に指示子を置くことで GPU の使用を可能とした OpenACC[10] が Nvidia 社から発表された．OpenACC は，CUDA への実装レベルで実績

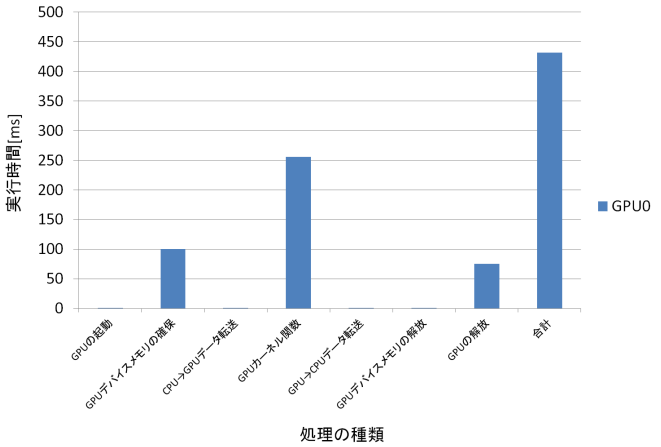

図 10 1GPU における実行の各処理にかかる処理時間測定結果

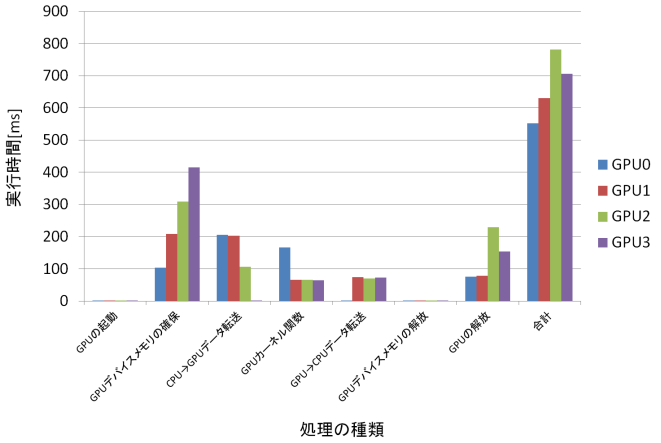

図 11 4GPU における実行の各処理にかかる処理時間測定結果

```
//格子点の近傍計算の繰り返しループ処理
#define SIZE 8192
float a[SIZE*SIZE],b[SIZE*SIZE];
#pragma omp parallel for private(i,k)
 for(j=1; j<SIZE-1; j++)
 {
        for(i=1; i<SIZE-1; i++){
          a[i+j*SIZE] = (b[i-1+j*SIZE]+b[i+1+j*SIZE]+b[i+(j-1)*SIZE]+b[i+(j+1)*SIZE])/4.0;
        }
 }
```

図 12 近傍計算の並列ループのソースコード

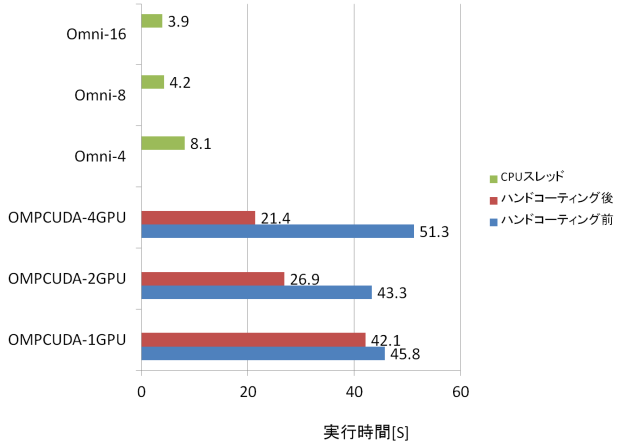

図 13 近傍計算の並列ループの結果

のある PGI 社の PGI Accelerator Model[9] をモデルとしており，OpenACC 独自の指示子を用いているため，CUDA で必要となる GPU-CPU 間のデータ転送などの命令をプログラマが書く必要がなくなり，従来の CUDA を扱うよりも簡単なプログラミングで GPU を扱うことができる．

それに対して，本研究では OpenMP の新たな指示子などは追加をせず，従来の OpenMP のプログラミングを行って GPU を取り扱えることを重点としている．

OpenMP プログラムから分散メモリ型並列プログラミングフレームへの変換を行う研究の一例として，OpenMPD[11] があげられる．OpenMPD は Omni OpenMP Compiler をベースとし，分散メモリ並列システムである MPI プログラムを OpenMP のように指示子による並列プログラミングで行うことができる手法となっている．OpenMPD では，共有メモリ型である OpenMP と分散メモリ型である MPI とのメモリモデルの対応付けを行えるように実装がされている．

また，複数 GPU に対応している事例として，三好らの MPI を埋め込み可能な GPU プログラミングフレームワーク[12] が挙げられる．この研究では複数 GPU 間のデータの直接のやり取りを可能とするために，CUDA プログラム内に MPI が埋め込み可能となっている．更には GPU 上での実行状態の保存や復帰を CPU コード上で行うことができる．

6. おわりに

本研究では OpenMP プログラムによって複数 GPU 環境を使用可能とすることを目的とし，その達成に複数 GPU 向け CUDA コードを生成する OpenMP 処理系を提案した．その実装には著者らが開発を行ってきた OMPCUDA を複数 GPU 向けに拡張することで行い，OpenMP と複数 GPU 向けの CUDA のメモリモデルと並列処理モデルの相違点の対応付けを行った．

OMPCUDA の性能評価実験を行ったところ，行列積計算プログラムではハンドコーティングによる CUDA プログラムにほとんど近い実行時間を得られた

グレゴリ級数による円周率求算や近傍計算の繰り返し処理プログラムの実験から，逐次ループ内の並列ループに対しては Global Memory の確保などの処理をループ外に出すなどの対応が必要であることがわかった．これは今後の課題としたい．

謝　　辞

本研究の一部は科学技術振興機構（JST）の戦略的創造研究推進事業（CREST）『ULP-

HPC:次世代テクノロジのモデル化・最適化による超低消費電力ハイパフォーマンスコンピューティング』によるものである。

参 考 文 献

1) Takashi Shimokawabe,et al., An 80-Fold Speedup, 15.0 TFlops, Full GPU Acceleration of Non-Hydrostatic Weather Model ASUCA Production Code, Proceeding of the 2009 ACM/IEEE conference on SuperComputing 2010 PP.1-11(2010).
2) 小川慧, 青木尊之，山中晃徳, GPU クラスタを用いた Phase Field モデルに基づく相変態計算のスケーラビリティ, 計算工学講演会論文集, PP.145-148(2010).
3) NVIDIA, NVIDIA CUDA C Programing Guide 3.2,nVIDIA,2010.
4) 大島聡史, 平澤将一，本多弘樹, OMPCUDA:GPU 向け OpenMP の実装,HPCS2009, PP.131-138(2009).
5) S.Ohshima, S.Shoichi，H.Honda,OMPCUDA : OpenMP Execution Framework for CUDA Based on Omni OpenMP Compiler ,In: EWOMP ’10 , PP.161-173(2010).
6) M.Sato, S.Satoh, K.Kusano and Y.Tanaka, Design of OpenMP Compiler for an SMP Cluster, EWOMP’99, PP.32-39(1999).
7) John Owens and UC Davis. Data-parallel algorithms and data structures.In SUPERCOMPUTING 2007 Tutorial: Hight Performance Computing withCUDA, 2007.
8) Seyong Lee, Rudolf Eigenmann, OpenMPC: Extended OpenMP Programming and Tuning for GPUs, Proceeding of the 2010 ACM/IEEE conference on SuperComputing2010, PP.1-11(2010).
9) PGI, PGI Compiler and Tools, http://www.softek.co.jp/SPG/Pgi/, 2012
10) NVIDIA, Cray Inc., Portland Group, CAPS enterprise, OpenACC DIRECTIVE FOR ACCELERATOR, http://www.openacc-standard.org/, November 2011
11) 李 珍泌, 佐藤 三久, 朴 泰祐, 分散メモリ向けデータ並列言語 OpenMPD の設計と実装, HOKKE2007，PP. 49-54，(2007).
12) 三好 健文，近藤 正章，入江 英嗣，吉永 努，本多 弘樹，MPI を埋め込み可能な GPU プログラミングフレームワークの検討, SACSIS2011, PP.298-305(2011).