OMPCUDA：GPU向けOpenMPの実装

大島聡史[†]　平澤将一[†]　本多弘樹[†]

GPUを用いた汎用演算GPGPUは高い演算性能が注目されている一方で，プログラム作成の難しさが問題となっている．そこで我々は，既存の並列化プログラミング手法を用いたGPGPUプログラミングを提案している．本論文では共有メモリ型並列計算機向けの並列化プログラミングにおいて広く用いられているOpenMPを用いたGPGPUの可能性を探るため，CUDA対応GPU向けのOpenMP処理系OMPCUDAを実装した．並列性の高いテストプログラムを用いて評価を行った結果，OMPCUDAは既存のOpenMPと同様の記述でGPU向けの並列プログラムを容易に作成できることおよびGPUの持つ高い並列性を活かしてプログラムの高速化が行えることが確認できた．

OMPCUDA : Implementation of OpenMP for GPU

Satoshi OHSHIMA,[†] Shoichi HIRASAWA[†] and Hiroki HONDA[†]

GPGPU (General-Purpose computation using GPUs) attracts attention because of its performance. However, the difficulty of the programming poses a problem. So we have proposed GPGPU programming techniques used the existing parallel programming techniques. In this paper, we implemented an OpenMP processor OMPCUDA for the NVIDIA CUDA, in order to illustrate the possibility of GPGPU using OpenMP which is widely used in parallel programming on the shared memory parallel computers. As a result of evaluation using test programs which has highly parallelism, we confirmed that OMPCUDA can create GPGPU parallel programs easy using existing OpenMP's notation and can make programs speed-up by utilizing advantage of highly parallelism of GPU.

1. はじめに

GPU (Graphics Processing Unit) はCPU (Central Processing Unit) と比べて理論演算性能や電力あたり性能が高いため，数値計算や動画像処理などを高速化するアクセラレータとして注目されている．今日では多くのPCに高性能なGPUが搭載されており，GPUを汎用演算に用いるGPGPU (General-Purpose computation using GPUs)[1)] を利用可能な計算機システムが普及しつつある．

一方で，GPGPUプログラミングには専用の言語やライブラリを利用する必要がある．またこれらを利用するためにはGPUのハードウェアアーキテクチャや実行モデル，性能最適化のための方針等を新たに習得する必要がある．そのうえGPUはCPUと比べて短期間でアーキテクチャが更新されるため，習得した知識や技術は長く使うことができない．そのためGPGPUプログラミングの習得はアプリケーションプログラマにとって負担が大きく，GPGPUの普及の妨げとなっている．

我々はより多くのアプリケーションプログラマにとってGPGPUを簡便に利用できるようにすることを目指している．GPUは並列処理方式によるハードウェアであり，GPGPUによる高速化が特に期待されているのは並列性の高いアプリケーションである．一方でCPU向けの並列化プログラミング手法についてはこれまでに多くの研究が行われており，対象ハードウェアや対象問題に応じて様々な手法が用いられている．そこで我々は，既存の並列化プログラミング手法がGPGPUプログラミングに利用できれば，これまでに蓄積された並列化の知識や技術を利用できるため多くのアプリケーションプログラマがより簡便にGPGPUを利用できるようになると考え，既存の並列化手法を用いたGPGPUプログラミングの提案を行ってきた[2)]．

本論文では，既存の共有メモリ型並列計算機で広く用いられているOpenMPを用いたGPGPUの可能性を明らかにするため，GPGPU環境の1つである

† 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
独立行政法人科学技術振興機構，CREST
Japan Science and Technology Agency, CREST

CUDA 向けの OpenMP 処理系 “OMPCUDA” を実装し，OMPCUDA を用いることでアプリケーションプログラマが GPU の性能を簡便に利用できるかを確認する．

本論文の構成を以下に示す．2 章では現在 GPGPU に用いられているプログラミング手法や言語についてまとめ，GPGPU プログラミングの課題を確認する．3 章では，OpenMP を用いた GPGPU プログラミングの可能性を検討する．4 章では OMPCUDA の実装について述べ，5 章ではサンプルプログラムを用いて OMPCUDA の記述の容易性と性能を確認する．6 章はまとめの章とする．

2. GPGPU プログラミングの課題

GPU を用いたグラフィックスプログラミングにおいては，GPU の動作タイミング制御や CPU-GPU 間の通信などを行うために DirectX や OpenGL といったグラフィックス API が，また GPU 内部の処理ユニットが行う処理を記述するために GPU 向けのアセンブリ言語や専用の高級言語 (シェーダ言語) が用いられている．これらの API や言語は標準化が進んでおり対応 GPU も多いため，現在でも画像表示を伴う GPGPU プログラミングに用いられている．

しかしながら，グラフィックスプログラミングの手法を用いてより汎用的な演算を記述するためには，演算データの配置をテクスチャに対するデータの設定に割り当て，演算内容をシェーダ言語の記述およびシェーダを用いた画像描画の手続きに置き換え，演算結果の取得にテクスチャ間描画を利用するなど，対象問題をグラフィックスプログラミングに対応させる独自のプログラミング手法を利用する必要がある．そのためには，対象アプリケーションのアルゴリズムに加えてグラフィックスプログラミングや GPU アーキテクチャについての知識が必要であり，アプリケーションプログラマにとって大きな負担となってきた．

そこで近年では，グラフィックスプログラミングを必要とせずに GPGPU プログラミングを行える新しい言語やライブラリの研究が進められている．

CUDA[3] や CTM[4] は，GPU ベンダが提供する GPGPU 向けのプログラミング言語やライブラリであり，グラフィックス API よりも低いレイヤで GPU にアクセスすることができる．そのため，GPU のアーキテクチャに適した実装を行うことでより高い性能を得ることが可能である．しかしながら，これらの言語やライブラリは GPU アーキテクチャへの依存度が高いため，利用するためには GPU のアーキテクチャを理解する必要がある．そのため，アプリケーションプログラマへの負担が軽減されているとは言い難い．

RapidMind[5] や PeakStream[6]，SPRAT[7] など，GPU による処理を「ストリームと呼ばれる入力データ」に対して「カーネルと呼ばれる演算操作」を作用させるものとみなすストリーミング言語の研究も進められている．ストリーミング言語は GPU のアーキテクチャをストリーム処理という概念で抽象化しているため，GPU に関する知識がないアプリケーションプログラマでもストリーミング言語を使うことで GPU 向けの並列プログラムを作成することができる．しかしながら，ストリーミング処理という概念や言語仕様の習得がプログラマにとっての新たな手間となっている．

3. OpenMP を用いた GPGPU プログラミング

本章では OpenMP と GPGPU の対応付けについて述べ，さらに次章では我々が実装を行った CUDA 対応 GPU 向けの OpenMP 処理系 OMPCUDA について述べる．

CPU 向けの並列化については，SIMD，SMP，マルチコア CPU，PC クラスタ，Grid など既に様々な環境を対象に数多くの研究が行われている．特にプログラマが明示的に並列性を記述する際には，pthread などのスレッドライブラリ，MPI などの通信ライブラリ，そして OpenMP が広く利用されている．

本研究では，以下の 2 点から OpenMP に着目し，OpenMP を用いることで GPGPU プログラミングを容易に行えるのではないかと考えた．これらの考えに基づく OpenMP と GPGPU の基本的な対応づけの方法を図 1 に示す．

(1) OpenMP の実行モデルでは，同時に実行可能な複数のスレッドと，スレッド間で共有されるメモリを持つ実行環境が想定されている．
一方現在の GPU には，同時に実行可能な多数の演算器と，演算器間で共有されるメモリが搭載されている．
そのため GPU 上の多数の演算器を OpenMP におけるスレッドに，GPU 上の共有メモリを OpenMP におけるスレッド間で共有されるメモリに割り当てることで，OpenMP を GPU に対応づけられると考えられる．

(2) OpenMP を用いた並列化プログラミングでは，プログラマが指示子を用いて明示的に指定した部分のみを複数のスレッドで共有メモリを用いて並列実行し，その他の部分は逐次実行する．

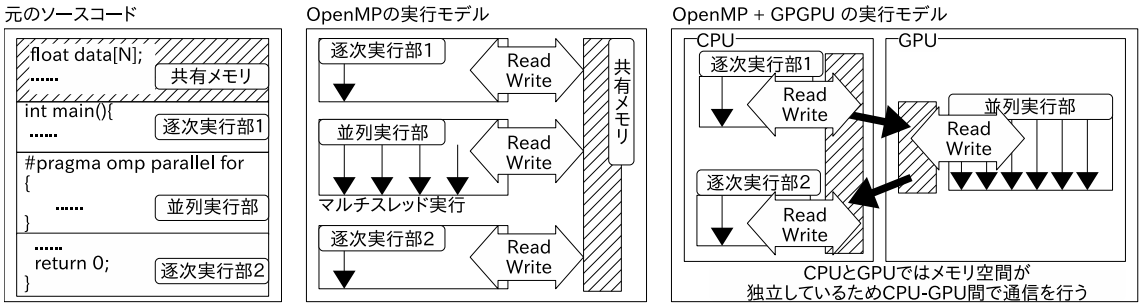

図 1 OpenMP と GPGPU の基本的な対応づけの方法
Fig. 1 Basic assignment of OpenMP and GPGPU

GPU は並列実行方式によるハードウェアであるため，GPGPU においては並列性の高い部分のみを GPU 上で並列実行し，その他の部分は CPU で逐次実行するのが一般的である．

プログラム全体の中から並列化による高速化に適した部分のみを選択的に並列実行する実行モデルは GPGPU と OpenMP で共通している．

一方で，既存の OpenMP では逐次実行部と並列実行部とで同じメモリ空間を利用するのに対して，CPU と GPU は互いに独立したメモリを持ちメモリ空間も異なるため，CPU-GPU 間で必要なデータを送受信する必要がある．しかしながら OpenMP の実行モデルでは逐次実行部と並列実行部が同時に実行されることはないため，逐次処理部と並列処理部の境界で必要なデータを全て送受信すれば正しい実行結果が得られると考えられる．またより高い性能を得るためには，更新されたデータのみを送受信するなど転送量を減らす機構が有効と考えられる．

さらに，並列実行部を CPU と GPU で分割実行することができればより高い性能が得られる可能性があるものの，CPU-GPU 間での適切なデータの送受信が必要になるため対象とする並列実行部によっては逆に性能が低下してしまうことが予想される．そのため常に高い性能を得るためには，並列実行部を CPU のみで実行するか，GPU のみで並列実行するか，(可能であれば)CPU と GPU で分割並列実行するかを選択する必要が生じる．この選択は並列実行部の計算量や CPU-GPU 間の通信時間に左右されるため，アプリケーションプログラマが選択するための新たな指示子やの導入や，最も高速な方法の選択を支援するチューニング機構などが必要になると考えられる．

4. OMPCUDA の提案と実装

OpenMP を用いて記述されたプログラムの並列実行部を GPU 上で実行するには，アプリケーションプログラマが記述したプログラムを GPU 上で実行可能な形式，例えば CUDA やシェーダ言語等，既存の GPGPU 向けプログラミングで用いられている形式に変換する必要がある．そこで我々は，プログラマが OpenMP を用いて記述したプログラムを GPU 上で実行可能な形式に変換する “プログラム変換機構” と，並列実行のために必要ないくつかの機能を提供する “実行時ライブラリ” からなる処理系 “OMPCUDA” を実装した．本章では OMPCUDA 全体の構成と，プログラム変換機構および実行時ライブラリの構成について述べる．

OMPCUDA は，OpenMP を用いた典型的な並列プログラムである**#pragma omp parallel for** 節を用いたループ並列プログラムに対して，GPU の特徴である高い並列性を有効に活用し効果的に並列化・高速実行することを主眼に置いて実装を行った．

4.1 OMPCUDA 全体の構成

2 章で述べたように，グラフィックス API とシェーダ言語を用いたグラフィックスプログラミングを用いれば様々な GPU で実行可能なプログラムを作成することができる．また CUDA や CTM を使えば，実行環境が限定される代わりに低いレイヤで GPU にアクセス可能となる．

我々は OpenMP プログラムを GPU 上で実行可能な形式に変換する機構が作りやすいことを重視して実行環境を CUDA 対応 GPU に絞り，OpenMP を用いて記述されたプログラムの並列実行部を CUDA 対応 GPU 上で実行する処理系 OMPCUDA を実装した．OMPCUDA は既存の OpenMP 処理系 Omni OpenMP Compiler version 1.6[8] (以下 Omni) をベースに実装した処理系である (図 2)．

Omni では，OpenMP 指示子の挿入されたソースコードを Omni 独自の中間表現 (X 形式) によって書かれた中間コードに変換したうえで，実行時ライブラリ (バックエンドコンパイラにより静的リンクされる

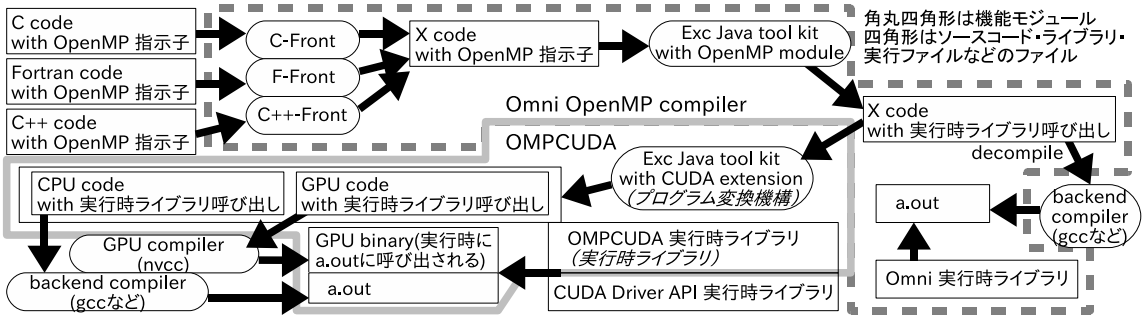

図 2 Omni と OMPCUDA の関係および OMPCUDA の全体像
Fig. 2 Relationship between Omni and OMPCUDA, and whole picture of OMPCUDA

ライブラリ) を用いたマルチスレッドプログラムへと変換している．OpenMP における並列実行部の指定はブロック単位で行われるのに対して，Omni の中間コードにおける並列実行部は関数として分離させられている．分離させられた関数は，実行時ライブラリの並列実行開始関数によってスレッドに割り当てられ，並列実行される．

一方で，CUDA における GPU 上での並列処理も関数単位であり，CPU からの指示に従い GPU 上の多数の演算器で同時に同じプログラムを実行する．ただし，演算の前に CPU から GPU へ必要なデータを送信し，また演算の後には GPU から CPU へ結果を送信する必要がある．そこで OMPCUDA も処理の対象を OpenMP 指示子の挿入されたソースコードではなく，Omni によって OpenMP 指示子に対する処理 (実行時ライブラリ呼び出しへの書き換え) が行われた後の中間コードとすることで，OpenMP 指示子に対する処理の再実装を省いている．

OMPCUDA は，プログラム変換機構と実行時ライブラリによって構成されている．

プログラム変換機構は Omni によって OpenMP 指示子に対する処理が行われた後の中間コードを調査し，“Omni の並列実行関数呼び出し” を “CPU から GPU へのデータ送信，GPU への演算実行開始指示，GPU から CPU へのデータ書き戻し” の一連の処理に置き換えて CUDA 向けのプログラムに変換する機構である．CPU-GPU 間で送受信する必要があるデータを探し出す機能も持ち合わせている．

実行時ライブラリは，Omni の実行時ライブラリが持つ機能を CUDA 向けに再実装したものである．

4.2 プログラム変換機構

OMPCUDA のプログラム変換機構が行う，Omni の中間コードを CUDA に対応したプログラムへと変換する手順を以下に示す．この手順は図 2 および図 3 と対応している．

(1) 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 (図 3-①).

(2) 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仕様のため，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 (図 3-②).

(3) 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 (図 3-③).
並列実行部に関数呼び出しが存在する場合は，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から呼び出される可能性もあるため，元のソースコードにも残したままとする．
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こととなる．

(4) GPU コードには，CUDA の記法に従い関数や変数に指示子を付加する．さらに関数の引数を調整し CPU-GPU 間で変数の送受信を行えるようにする．これを GPU コンパイラでコンパイルし，CUDA の実行時 (動的リンク) ライブラリおよび OMPCUDA の実行時 (静的リンク) ライブラリとリンクして GPU 用の実行ファイルを生成する (図 3-④).

(5) CPU コードについては既に述べた通り，Omni の並列実行開始関数呼び出しを CPU-GPU 間の

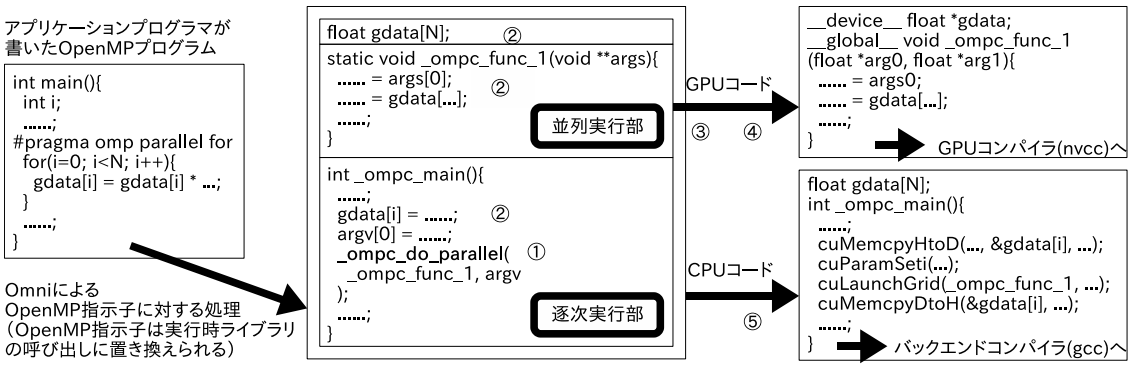

図 3 プログラム変換機構の処理手順
Fig. 3 Sequence of program converter

データ送受信および GPU への演算実行開始指示に書き換える．これをバックエンドコンパイラでコンパイルし，CUDA および OMPCUDA の実行時ライブラリとリンクして実行ファイルを生成する (図 3-⑤).

現在の実装では並列実行部で参照するデータは全て送受信の対象としているが，GPU 上で変更されないデータは CPU へ書き戻さないようにするなど送受信するデータサイズを削減することで性能が向上すると考えられる．なお CPU 向けの OpenMP ではスレッド間でメモリ空間を共有するため動的に確保される配列やポインタを含んでいても実行できるが，OMPCUDA では対象となるデータを特定できなくては転送が行えない．現在は動的に確保される配列やポインタを含むプログラムを正しく実行できないが，今後は自動並列化等の研究を参考に改良を行う予定である．

4.3 実行時ライブラリ

Omni の実行時ライブラリは，スレッドの生成および割り当てやループのスケジューリング，スレッド間の同期処理，リダクション演算などの機能を提供する．

OMPCUDA は，Omni において逐次実行部で行っている処理は CPU で行い，並列実行部で行っている処理は GPU で行う，という方針で実装している．処理によっては CPU から GPU ないし GPU から CPU へ移動した方が性能が向上する可能性もあるが，今後の課題とし，本論文では扱わないものとする．

本論文では単純な for ループの並列化に最低限必要な関数として，GPU 上で実行される for ループのスケジューリングを行う関数を実装した．さらに，for 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も特に典型的な，各スレッドの持つデータの総和を求める演算を，GPU 上で実行されるリダクション関数として実装した．Omni では CPU 上で実行されるスレッド割り当ておよび並列実行開始関数も提供しているが，これについてはプログラム変換機構による CUDA 実行時ライブラリ関数の呼び出しへの書き換えが役割を果たしているため，OMPCUDA の実行時ライブラリでは特に処理を行わない．

for ループのスケジューリング

Omni における for ループの並列化は，実行時ライブラリが提供する，各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．CUDA の実行時ライブラリにもスレッド ID と同様に一意の ID を得る関数が備わっているため，OMPCUDA はこれを用いて Omni と同様に部分ループを作成するための関数を提供することにした．

Omni はループ並列化におけるスケジューリング方法を複数種類提供しており，アプリケーションプログラマは提供されている方法から 1 つを選択することができる．OMPCUDA では最も単純なスケジューリングであるループのイタレーションを等分割したスタティックスケジューリングのみを提供する．

OMPCUDA における for ループのスケジューリング (割り当て) 例を図 4 に示す．

CUDA における並列実行は既存の CPU におけるスレッド実行と異なり，Block と Thraed の 2 階層からなっている．CPU からの並列実行開始指示を受けると，多数の Thread が同時に同じ関数の実行を開始する．Block は Thread が複数まとめられたものであり，各 Block を構成する Thread の数は全 Block で同一である．同一 Block 内の Thread は同時に異なる演算を行うことができないが，Block ごと・Thread ごとに固有の ID を取得できるため，各 Thread は ID を用いて共有メモリ上の異なるデータに対して同時に演算を行うことができる．

そのため CUDA における並列処理については，

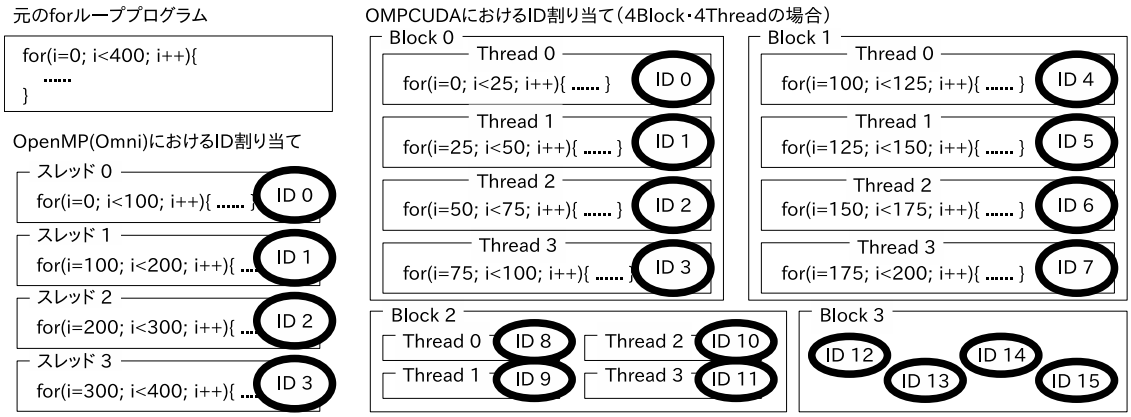

図 4 OMPCUDA における for ループのスケジューリング方法
Fig. 4 A for-loop scheduling technique of OMPCUDA

1Block 内の複数 Thread を用いた並列処理，複数 Block を用いるが各 Block 内では 1Thread のみを利用する並列処理，複数 Block・複数 Thread を用いた並列処理，以上の 3 種類の並列処理が考えられる．複数 Block・複数 Thread を用いた並列処理が最も多くの演算を同時実行できるものの，同一 Block 内の各 Thread は同時に異なる演算を行えないため，対象問題によっては複数 Thread を用いた並列処理の効果が得られない可能性がある．

これに対して，単純な for ループの並列化では各スレッドが同時に同じ演算を行うことが多い．そのため，CPU 向けの OpenMP におけるスレッド並列処理は，上記 3 種類の並列処理のいずれにも対応づけることができると考えられる．そこで OMPCUDA では，GPU の持つ高い並列性を有効に活用できるようにすることを目指し，for ループを複数 Block・複数 Thread を用いた並列処理に割り当てている．

ところで，CUDA では物理的に搭載されたプロセッサの数を超える多くの Block や Thread を利用するプログラムを作成し動作させることが可能である (多い分は時分割実行される) が，最適な Block 使用数・Thread 使用数は対象問題と対象 GPU により左右される．そのため現在の OMPCUDA では，Thread 使用数の初期値を 256(プログラミングガイドでは 256 が適切な数の目安とされている)，Block 使用数の初期値をループ回数を 256 で割った数としている．またアプリケーションプログラマに対して CUDA のアーキテクチャが見えてしまうため好ましくないものの，Block 使用数・Thread 使用数を実行時に関数や環境変数を用いて指定可能にしている．

リダクション演算

GPU や CUDA を用いたリダクション演算については，既に様々な実装が知られている．

OMPCUDA では Owens らの手法[9] をベースとしたリダクション関数を実装した．

5. 評　　価

本章では OMPCUDA について，並列プログラムが容易に作成できるかと，並列実行の効果により高い性能が得られるかの 2 点について評価を行う．

評価対象プログラムには，理想的な並列性を持つプログラムを実際に並列高速化できるかを確認するため，円周率の計算 (グレゴリ級数) と行列積を用いた．

評価環境は表 1 に示す通りである．

表 1 評価環境
Table 1 Evaluation environment

CPU	Intel Xeon E5345 2.33GHz (4 コア)
メインメモリ	4.0GB (32bitOS につき 3.2GB 使用可)
GPU	NVIDIA GeForce GTX 280 演算器 (SP) 数 240, SP クロック 1296MHz
ビデオメモリ	1.0GB
CPU-GPU 接続	PCI-Express Gen2 x16
OS	CentOS 5.0 (kernel 2.6.18)
コンパイラ	GCC 3.4.6, nvcc 2.0 v0.2.1221 Omni OpenMP Compiler version 1.6

円周率の計算 (グレゴリ級数)

グレゴリ級数を用いた円周率の計算は，最後にリダクション演算が必要な以外は理想的な並列性を持ち，計算に必要なデータ量も少ないため，並列化による高速化が容易なプログラムである (図 5)．

実行結果を図 6 および図 7 に示す．OMPCUDA の実行時間はいずれも CPU-GPU 間でのデータ転送時間を含むが，GPU を初期化する時間は含んでいない．

図 6 は，並列度の指定に対する CPU と GPU (OMPCUDA) それぞれの実行時間を示している．Omni に

```
  float answer = 0.0f;
  int i, n = 1000000000;
#pragma omp parallel for reduction(+:answer)
  for (i = 0;i < n;i++){
    answer += (4.0 / (4 * i + 1)
             - 4.0 / (4 * i + 3));
  }
```

図 5 円周率計算のソースコード
Fig. 5 Source code of pi

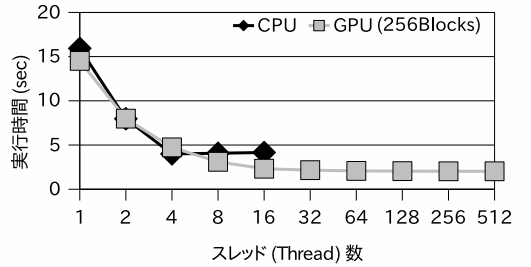

図 6 pi の実行時間 1
Fig. 6 Execution time of pi 1

おけるスレッドと OMPCUDA における Thread は同一の概念ではないが，ここでは同じ軸上に示している．CPU は，同時実行可能なスレッド数に達するまでは実行時間が短くなるものの，搭載されているコア数以上のスレッドを用いても実行時間は短くならない．一方で GPU は CPU と比べてより多くの演算を同時に実行可能であり搭載されている演算器数以上の並列度でも性能が向上するため，CPU より高い並列度まで実行時間が短縮されている．

また，GPU に搭載されている計算コアは単体での演算性能が CPU に搭載されている計算コア単体での演算性能と比較して低いため，使用する Block 数・Thread 数が少ない場合には CPU と比べて低い性能であるものの，使用する Block 数・Thread 数が多い場合には CPU と比べて高い性能が得られていることも確認できた．

図 7 は，GPU について Block 数と Thread 数を広く変化させた際の実行時間を示す対数グラフである．図 6 と同様に GPU に搭載されている演算器数を超えて性能が向上していることがわかる．

行 列 積

続いて，行列積を用いて性能比較を行った．

行列積はグレゴリ級数と同様に高い並列性を持つ問題である．GPU を用いて高速実行できる問題としても知られており，描画機能やシェーダを用いた実装，GPU の性能モデル化，CPU と GPU を用いた並列計算など様々なテーマで取り上げられている[10)11)12)]．

行列積には CPU・GPU ともに様々な最適化手法があるものの，本論文では最大性能を得ることよりも単

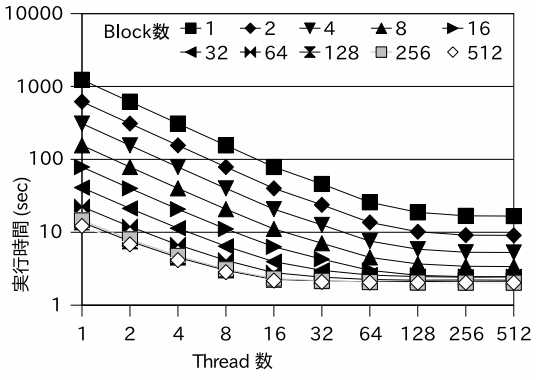

図 7 pi の実行時間 2
Fig. 7 Execution time of pi 2

```
#define N 1024
  float a[N*N], b[N*N], c[N*N];
#pragma omp parallel for private(j,k)
  for(i=0; i<N; i++){
    for(j=0; j<N; j++){
      for(k=0; k<N; k++){
        c[i*N+j] += a[i*N+k] * b[k*N+j];
  } } }
```

図 8 行列積のソースコード
Fig. 8 Source code of matrix multiplication

純な実装で容易に性能を得られることを重視しているため，図 8 に示すような単純な三重ループによる実装を用いて性能を比較した．

実行結果を図 9 および図 10 に示す．グレゴリ級数と同様に OMPCUDA の実行時間はいずれも CPU-GPU 間でのデータ転送時間を含んでいるが，GPU を初期化する時間は含んでいない．

図 9 は図 6 同様に CPU と GPU の実行時間を示している．CPU・GPU ともに，一定のスレッド数までは実行時間が短くなり，それ以上では逆に長くなるという傾向を示している．なお，並列化対象となった for ループのループサイズが 1024 なのに対して Block 数*Thread 数が 1024 を超えても性能が向上している．これはループのスケジューリングが番号の小さい Block に優先的に問題を割り当てる都合によるもので，例えば Block256・Thread32 の点で実際に演算を行っているのは 32Block*32Thread であり，Block256・Thread16 の点で実際に演算を行っている 16Block*64Thread と比べてより適した並列度だったと考えることができる．

図 10 は図 7 同様に Block 数と Thread 数を広く変化させた際の実行時間を示す対数グラフである．円周率の計算と比べてグラフが上下に揺れているものの，ほぼ同様の傾向が見られる．

以上の 2 つの実験では，ソースコードに修正を行

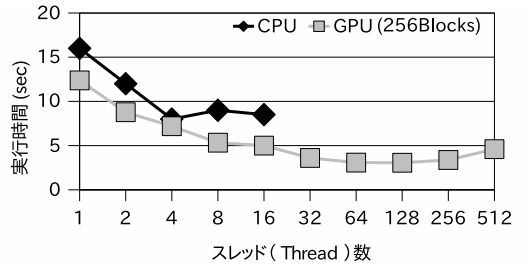

図 9 行列積の実行時間 1
Fig. 9 Execution time of matrix multiplication 1

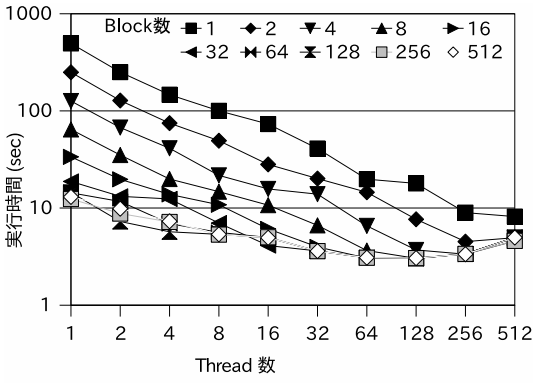

図 10 行列積の実行時間 2
Fig. 10 Execution time of matrix multiplication 2

わずにプログラムのコンパイルを行うことができた．CUDA プログラムの記述には CPU-GPU 間のデータ転送と GPU への演算開始指示を行うために 10 行から 20 行程度のプログラム記述を，さらに CUDA の初期化などの処理を含めると全部で 80 行程度の CPU 上で実行される CUDA プログラムを記述する必要があり，OMPCUDA はこの行数分の実装の手間を削減できたと言える．これらのことから，OMPCUDA を用いることでアプリケーションプログラマは OpenMP の知識を用いて簡便に GPGPU プログラムを作成できることが確認できた．

6. おわりに

本論文では GPU を簡便に利用する方法として，CUDA 対応 GPU 向けの OpenMP 処理系 OMPCUDA を実装した．またテストプログラムを用いて評価を行い，OMPCUDA はアプリケーションプログラマにとって大きな手間となる GPU アーキテクチャに関する知識の習得や CPU-GPU 間の通信の記述等を隠蔽できること，多数の演算器を活用することで CPU よりも高い性能が得られることを確認した．

今後は OMPCUDA を sections 節などの OpenMP 指示子や階層的な並列化にも対応させ，様々なアプリケーションを GPU 上で容易に実行できるようにすることを目指す．また発展的な課題としては，GPU と同様に多数の演算コアを持ちプログラミングの難しさが問題視されている Cell B.E. 向けの OpenMP コンパイラとの比較などが考えられる．

参 考 文 献

1) Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E. and Purcell, T. J.: A Survey of General-Purpose Computation on Graphics Hardware, *Eurographics 2005, State of the Art Reports*, pp. 21–51 (2005).
2) 大島聡史, 平澤将一, 本多弘樹: 既存の並列化手法を用いた GPGPU プログラミングの提案, 情報処理学会研究報告 (ARC-175), pp. 7–10 (2007).
3) NVIDIA Corporation: NVIDIA CUDA Compute Unified Device Archtecture Programming Guide version 2.0 (2008).
4) Advanced Micro Devices: ATI CTM Guide Version 1.01 (2006).
5) McCool, M. D.: Data-Parallel Programming on the Cell BE and the GPU using the RapidMind Development Platform, *GSPx Multicore Applications Conference* (2006).
6) Peakstream: The Peakstream API for GPUs, http://www.peakstreaminc.com/.
7) 滝沢寛之, 白取寛貴, 佐藤功人, 小林広明: SPRAT：実行時自動チューニング機能を備えるストリーム処理記述用言語, 情報処理学会論文誌コンピューティングシステム, Vol. 1, No. 2, pp. 207–220 (2008).
8) M.Sato, S.Satoh, K.Kusano and Y.Tanaka: Design of OpenMP Compiler for an SMP Cluster, *EWOMP '99*, pp. 32–39 (1999).
9) Owens, J. and Davis, U.: Data-Parallel Algorithms and Data Structures, *SUPERCOMPUTING 2007 Tutorial: Hight Performance Computing with CUDA* (2007).
10) K.Fatahalian, J.Sugerman and P.Hanrahan: Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication, *Graphics Hardware 2004* (2004).
11) Jesse D. Hall, Nathan A. Carr, J. C. H.: Cache and Bandwidth Aware Matrix Multiplication on the GPU , Technical report, University of Illinois Dept. of Computer Science (2003).
12) 大島聡史, 吉瀬謙二, 片桐孝洋, 弓場敏嗣: CPU と GPU を用いた並列 GEMM 演算の提案と実装, 情報処理学会論文誌コンピューティングシステム, Vol. 47, No. SIG12(ACS 15), pp. 317–328 (2006).