# ようこそGPGPUの世界へ

二村幸孝　出口大輔

## Ⅰ．はじめに

GPU（Graphics Processing Units）を使ってHPC（High Performance Computing）をやってみよう，というのが本稿の趣旨である。近年，GPUに汎用計算をさせる試みとしてGPGPU（General-Purpose computation on GPU）に関する技術が非常に注目を集めている。というのも，CPUに対するGPUの性能が非常に高くなってきたこと，また，高性能なGPUを手軽[*1]に入手できるようになってきたこと，が大きな要因である。例えば，Intel Quad-Core Xeon E5472（3.0 GHz，2×6 MB L2 cache，1600 MHz FSB）の性能が約80 GFlops［1］であるのに対し，nVidia Geforce 8800GTXの性能は300 GFlops以上と言われている。これらの結果は同じプログラムを用いて評価したものではないため，一概にどちらの性能が高いかを論じることはできないが，筆者がCPUとGPUの両者を利用した経験から言わせてもらえば，GPUの性能の高さには目を見張るものがある。また，2008年度中にはこの2倍以上の性能を持つGPUが市場に投入され，その性能は約1 TFlopsに達する予定である[*2]。ちなみに，TOP 500プロジェクト［2］の2005年6月のランキングでは500位の性能が約1.2 TFLOPSであることを考えると，最新のGPUは3～4年前のスーパーコンピュータ並の性能を秘めているとも考えられる[*3]。このように，非常に高性能なGPUを手軽に入手できるようになってきたことから，GPUをグラフィックス処理以外の目的に利用するGPGPUに対する期待が高まってきている。

GPGPUの最も初期の研究成果として，1978年に発表されたIkonas Systemが挙げられる［3］。そして，1990年代にはGPUをグラフィックス以外の用途に利用しようという試みもなされている。その後，2000年頃からGPGPUに関する研究が数多く行われるようになり，プログラマブルシェーダ対応のグラフィックスカードが登場して以降，GPGPUの応用範囲は多岐に亘るようになった［4,5］。現在では，GPGPUを行う際にシェーダ言語（HLSL，GLSL，Cgなど）と呼ばれる高級言語の利用が可能であり，通常のプログラムを書く感覚でGPUを利用することも可能になっている。

本稿で紹介するCUDA（Compute unified device architecture）は，nVidia社が提供しているGPUを利用するためのC/C++言語の統合開発環境である。従来，HLSLやGLSLといったシェー

*1　nVidia Geforce 8800GTXは約7～8万円で購入可能（2007年末の時点）。

*2　2008年6月16日に，nVidia社からGeForce GTX 280，AMD社からAMD FireStream 9250が発表された。これらの性能は約1 TFlopsであり，倍精度浮動小数点演算もサポートされている。

*3　最新のランキングではBlueGene/Lが478 TFLOPSを記録している（2007年末の時点）。

ダ言語を用いる場合，DirectX や OpenGL といったグラフィックス処理 API に関する知識が必要不可欠であった。また，これらのシェーダ言語はグラフィックス処理向けに設計されているため，実装するアルゴリズムをグラフィックス処理に適した形に設計しなおす必要があった。これに対し，CUDA では GPU を複数のスレッドを同時に実行できる並列計算機のように扱うことが可能であり，また，C/C++ 言語を用いてプログラムを書くことができる。そのため，これまでに開発してきたアルゴリズムを容易に移植して実行することが可能である。そこで本稿では，CUDA を用いて GPGPU を行うための具体的な手順を示すとともに，GPGPU へ取り組む際に注意すべき点を述べる。

以下，II. で CUDA を使用するための環境の構築方法を示し，III. で CUDA を使う上で注意すべき点と有用なツール群の説明を行う。そして, IV. で CUDA を使ったプログラム例を示し, V. でその他の応用例を示す。最後に，VI. でまとめる。

## Ⅱ. 環境構築

CUDA を使用するためには，CUDA に対応したハードウェア機構を持つ GPU を用意する必要がある。CUDA 公式サイト［6］のドキュメントによると，GeForce 8 以降は CUDA に対応したハードウェア機構が継続的にサポートされていくようである。表 1 に，現在販売されている GPU のうち，CUDA に対応したハードウェア機構を持つものを示す。nVidia の GPU には 3 種類のシリーズが存在しているが，GeForce と Quadro シリーズは，通常のグラフィックスカードとして販売されている製品である。特に，コンシューマ向けの GeForce シリーズは非常に安価に購入することができる。Tesla シリーズは HPC に特化した製品であり，通常のグラフィックスカードとして利用することはできない。

現在 CUDA は Windows XP，Windows Vista，Linux で使用することができる。また CUDA を利用するために必要なソフトウェアとして，Windows では統合開発環境である Visual Studio

表 1　CUDA に対応する nVidia 社製 GPU の一覧

| Series | Products |
|---|---|
| GeForce | 9800 GX2, 9800 GTX, 9800 GT, 8800 Ultra, 8800 GTX, 8800 GTS, 8800 GT, 8800 GS, 8600 GTS, 8600 GT, 8500 GT, 8400 GS, 8800M GTX, 8800M GTS, 8700M GT, 8600M GT, 8600M GS, 8400M GT, 8400M GS, 8400MG |
| Quadro | FX5600, FX4600, FX3700, FX1700, FX570, FX370, NVS290, FX3600M, FX1600M, FX570M, FX360M, Quadro Plex 1000Model IV, Quadro Plex 1000Model S4, NVS320M, NVS140M, NVS135M, NVS130M |
| Tesla | C870, D870, S870 |

2003または2005，Linuxではgccやg++をはじめとする開発環境を必要とする。これはCUDAに付属するコンパイラが，それぞれの開発環境に含まれる機能を利用するためである。以降，本稿ではWindowsを対象に解説を進めていく。読者の使用している計算機がこれらのハードウェアとソフトウェアの必要条件を満たしていない場合，残念ながらCUDAの恩恵を受けることはできない。しかし，これらの環境を新たに整えたとしても，非常に低コストでHPC環境を手に入れることができる。この機会に是非購入を検討して欲しい。

CUDAや関連するさまざまなドキュメントは，公式サイトから誰でも自由に入手することができる。早速，最新バージョンである2.0 Betaをダウンロードしてインストールしよう。CUDAの開発環境を構築するには，ドライバ，ツールキット，SDKの3つのパッケージが必要である。OSが異なる読者は，対応するOS向けのパッケージをインストールして欲しい。

- NVIDIA Driver for Microsoft Windows XP with CUDA Support (174.55)
- CUDA Toolkit version 2.0 for Windows XP
- CUDA SDK version 2.0 for Windows XP

デフォルトの設定では，CUDAツールキット（CUDA Toolkit）は`C:\CUDA`に，CUDA開発者SDK（CUDA SDK）は`C:\Program Files\NVIDIA Corporation\NVIDIA CUDA SDK`にインストールされる。

インストールが完了したら，正しくCUDA環境が構築できているかどうかを確認しよう。CUDA SDKに付属のサンプルプログラムを実行してもよいが，ここではCUDA環境でのプログラミングを理解するために非常に簡単なプログラムを作成する。まず，読者の好みのエディタを使用してプログラム1を打ち込み，`main.cu`というファイル名で保存しよう。“`.cu`”はCUDAに付属のコンパイラ`nvcc`でコンパイルされるソースコードを示す拡張子である。プログラムを保存したら，スタートメニューから“Visual Studio 2005コマンドプロンプト”を起動する。なお，通常のコマンドプロンプトでは，CUDAのプログラムのコンパイルに必要な環境変数が設定されていないためコンパイルすることができないことを覚えておこう。コマンドプロンプトが起動したら，`main.cu`が保存されているディレクトリで，

```
C:\Your\Source\Path> nvcc main.cu
```

のコマンドを実行する。ここで，`a.exe`が得られればコンパイル成功である。それぞれの環境で，プログラムが正しく動作することを確認して欲しい。

このプログラムは非常に単純ではあるが，CUDAにおける並列処理の基本が詰まっている。プログラム1を見られた読者は，次のような見慣れないコードに気付くだろう。

```
kernel <<< nBlocks, nThreads >>> ( dData );
```

これは，GPU上で実行される関数をCPUから呼び出すために，nVidiaがC/C++の構文を拡張した部分である。CUDAではGPUを複数のスレッドを並列に実行できる計算機のように扱うため，何らかの方法でスレッド数等をCPU側から指定する必要がある。この機能に対応するものが，上述の“<<< ... >>>”の部分である。“<<< ... >>>”で指定されたパラメータを用いて，GPU上で **__global__void** kernel（ **int** *data ）が並列に実行される。これらの機能に関する具体的説

プログラム 1　はじめの一歩

```
1   #include <stdio.h>
2
3   __global__ void kernel( int *data )
4   {
5       data[ threadIdx.x ] = threadIdx.x;
6   }
7
8   int main( int argc, char *argv[] )
9   {
10      int *dData, hData[ 5 ];
11      cudaMalloc( ( void ** )&dData, sizeof( int ) * 5 );
12
13      dim3 nThreads( 5, 1 );
14      dim3 nBlocks( 1, 1 );
15      kernel<<< nBlocks, nThreads >>>( dData );
16
17      cudaMemcpy( hData, dData, sizeof( int ) * 5, cudaMemcpyDeviceToHost );
18
19      for( int i = 0 ; i < 5 ; i++ )
20      {
21          printf( "%d", hData[ i ] );
22      }
23      printf( "\n" );
24
25      return( 0 );
26  }
```

明は，次の III. で述べる。

これで読者も GPGPU への第一歩を踏み出すことができた。次章は，CUDA のプログラムを開発していく上で必要な知識とツール群を紹介する。

## Ⅲ．入門編

CUDA 環境でプログラムを開発していくために，CUDA におけるプログラミングモデルとメモリモデル，言語拡張などを理解しておこう。これらを理解することで，より GPU の特性を生かしたプログラムを作成することが可能となる。

GPU は多数のスレッドが高い並列性をもって処理を実行することが可能なプロセッサであるが，GPU のみでプログラムを実行することはできない。そのため，CUDA 環境では GPU は並列演算可能なデバイスとして扱われる。図 1 は CUDA におけるスレッド管理を表している。図に示すように，CUDA ではスレッドのまとまりをブロック，ブロックのまとまりをグリッドと呼び，階層的に全スレッドを管理している。なお，CUDA では CPU における並列実行のように異なるカーネル ( プログラム ) を実行することはできず，グリッド内の全スレッドで同じカーネルが実行される。

CUDA のメモリモデルを図 2 に示す。各スレッドはレジスタとローカルメモリを持ち，また

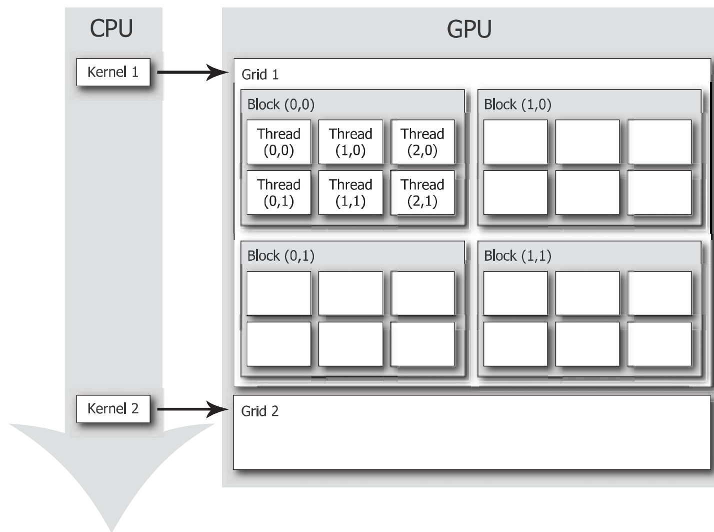


図１　CUDA のプログラミングモデル

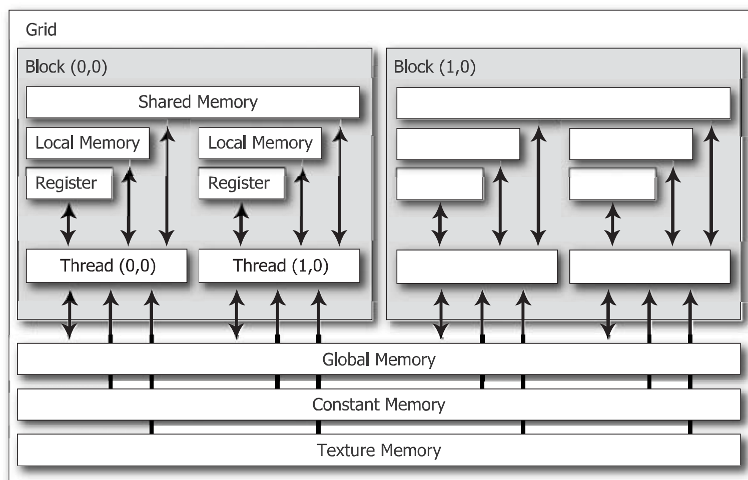


図２　CUDA のメモリモデル

各ブロックでは同じブロック内のスレッド間で共有される高速アクセス可能な共有メモリを持つ。さらに，各グリッドは，同じグリッド内の全スレッドで利用可能なグローバルメモリ，コンスタントメモリ，テクスチャメモリを持つ。このように CUDA ではさまざまなメモリが存在するが，アクセス速度やアクセス可能範囲，キャッシュの有無，などいくつか異なる点が存在するため，目的に応じて使い分ける必要がある。

また，CUDA での開発には拡張された C/C++ 言語を使用する。この拡張には，表２に示す修飾子の追加や，カーネル実行時の並列数を制御するための構文拡張，スレッドを一意に決定するための組み込み変数が含まれる。追加される修飾子には，関数型修飾子と変数型修飾子の２種類が存在し，CPU に対するコードと GPU に対するコードを判別するために利用される。カーネル

表２　言語拡張により追加される修飾子

| 関数型 | **__device__** | GPU から呼び出され，GPU で実行される関数。 |
|---|---|---|
| | **__global__** | CPU から呼び出され，GPU で実行される関数。 |
| | **__host__** | CPU から呼び出され，CPU で実行される関数。 |
| 変数型 | **__device__** | GPU 上のメモリに存在する変数。 |
| | **__constant__** | GPU 上のコンスタントメモリに存在する変数。 |
| | **__shared__** | GPU 上の共有メモリに存在する変数。 |

実行時の並列数の制御は，プログラム２のように記述することで行われる。プログラムにおける **__global__void** kernel ( **int** *parameter ) 関数は，表２に示す **__global__** 修飾子が付加されており，CPU から呼び出され GPU で並列に実行される関数（カーネル）である。そのため，**__global__ void** kernel ( **int** *parameter ) を呼び出す際には，

kernel<<< nBlocks, nThreads, nBytes >>>( parameter );

のように，どれだけの並列数でカーネルを実行するのかを指定する必要がある。なお，nBlocks はグリッド次数（ブロック分割数），nThreads はブロック次数（スレッド分割数），nBytes はブロックごとに割り当てる共有メモリのバイト数を表す。共有メモリを使用しない場合は nBytes を省略することが可能である。すべてのスレッドは並列に実行されるが，同じブロック内のスレッドに限り **__syncthreads** を使用することで同期させることも可能である。また，カーネルのコード内では，各スレッドがデータのどの部分を処理するかを判別するために，表３に示す４つの組み込み変数を利用することができる。

公式サイトでは，CUDA のプログラミングを容易にするためのツールやドキュメントが提供されている。特に，カーネルによるプロセッサの占有率を計算することができる“CUDA Occupancy Calculator”は非常に有用で，このツールを利用することで GPU の性能を最大限に生かしたコードを作成することができる。興味のある読者は，公式サイトからツール及びドキュ

プログラム２　カーネルの並列実行

```
1   __global__ void kernel( int *parameter )
2   {
3       // カーネルの実装
4   }
5
6   int main( int argc, char *argv[] )
7   {
8       // ...
9
10      // カーネルの実行
11      kernel<<< nBlocks, nThreads, nBytes >>>( parameter );
12
13      // ...
14  }
```

表３　組み込み変数

| | |
|---|---|
| gridDim | グリッドの次数。 |
| blockIdx | スレッドが属するブロックのインデックス。 |
| blockDim | スレッドが属するブロックの次数。 |
| threadIdx | ブロック内のスレッドのインデックス。 |

メントをダウンロードし，使用方法を学んで欲しい。

## Ⅳ．実践編

それでは，CUDA を使って実践的な GPGPU プログラミングに挑戦してみよう。本節では，$\mathbf{C}=\mathbf{A}\times\mathbf{B}$ の形で記述される行列積を例に挙げて，CUDA の詳細なプログラミング方法を紹介する。ただし，問題の簡単化のために行と列の大きさは 16 の倍数に限定して説明を行う。汎用的な行列積に関しては，読者への課題としたい。また，以下の説明では行列 **A** の $r$ 行 $c$ 列目の要素を $a_{rc}$ と表し，行列内の各要素は列優先の順序でメモリ内に配置している。それでは，さっそく行列積を実現するプログラムを見ていこう。

前節で説明したように，CUDA では CPU 側で実行されるコードと GPU 側で実行されるコードを明示的に区別して記述する必要がある。例えば，CPU 側から呼び出され GPU で実行される関数の先頭には“**__global__**”というキーワードを付加し，GPU 側から呼び出され GPU で実行される関数には“**__device__**”というキーワードを付加する。また，CUDA では CPU 側と GPU 側でメモリを共有することはできないため，プログラマが明示的にメモリの転送を行う必要がある[*4]。CUDA では，スレッド内で使用するレジスタやローカルメモリに加え，ブロック内のスレッド間で共有可能な共有メモリ，GPU 内の全スレッドで共有されるグローバルメモリ，テクスチャメモリ，コンスタントメモリが存在する。CUDA のプログラムでは，GPU 上で実行される関数（カーネル）内のローカル変数はレジスタ（場合によってはローカルメモリ）に割り当てられる。そして，“**__shared__**”を変数宣言の先頭に付加した場合のみ共有メモリとして利用することが可能となる。また，グローバルメモリを GPU 上で実行される関数内で利用したい場合には，それらを指すポインタを関数の引数として渡す必要がある。これらの点に注意してプログラム３に目を通していただきたい。

プログラム３は行列 **C** の各要素 $c_{rc}$ を計算するプログラムであり，

$$c_{rc}=\sum_{k=1}^{cA} a_{rk}\times b_{kc} \tag{1}$$

を，GPU の各スレッドで求める非常に単純なものである。ここで，$rA$ は行列 **A** の行数，$cA$ は行列 **A** の列数を表している。また，プログラム３には，GPU 側のメモリ上に存在する行列 **A**,

*4　GPU 側のメモリへ CPU から直接アクセスすることはできない。また，CPU 側のメモリへ GPU から直接アクセスすることもできない。プログラムを書く際に落とし穴になる可能性があるため，十分注意が必要である。

プログラム３　行列積を行う GPU 関数

```
1   __global__ void multiply( float *A, float *B, float *C, int rA, int cA )
2   {
3       int c = threadIdx.x + blockIdx.x * blockDim.x;
4       int r = threadIdx.y + blockIdx.y * blockDim.y;
5
6       float sum = 0.0f;
7       for( int k = 0 ; k < cA ; k++ )
8       {
9           sum += A[r + k * rA] * B[k + c * cA];
10      }
11
12      C[c * rA + r] = sum;
13  }
```

**B**，**C** へのポインタを入力する必要がある点に注意していただきたい。各スレッドが計算する範囲は，スレッドを識別するための変数の“threadIdx”と“blockDim”を用いて決定している。“blockDim”は，GPU 関数“multiply”を呼び出す際に設定したスレッド数によって変化し，“threadIdx”は GPU 内のスレッドごとに値が自動設定される。“threadIdx”と“blockDim”の詳しい説明に関しては，文献［8］の 2.2 節を参考にしていただきたい。

それでは，プログラム３を使って実際に行列積を計算してみよう。プログラム３を呼び出すための CPU 側の処理をプログラム４に示す。先で述べたように，CUDA では CPU 側と GPU 側でメモリを共有することはできない。そのため，CPU と GPU それぞれで行列を保持するためのメモリ領域を確保している（11 ～ 13 行目が CPU 側のメモリ確保，16 ～ 18 行目が GPU 側のメモリ確保）。そして，23 ～ 24 行目で CPU 側のメモリを GPU 側のメモリへ転送している。ここで，“cudaMemcpy”関数の最後の引数により，CPU と GPU のどちら向きにメモリを転送するかを指定している。生成する GPU のスレッド数は 27 ～ 28 行目で設定し，31 行目でプログラム３を実行する。CUDA ではスレッド数とブロック数を適切に設定することで，問題に合わせて計算範囲を動的に変更することが可能である。ここでは，III. で紹介した“CUDA Occupancy Calculator”を利用して，行列の大きさに合わせて適切なスレッド数とブロック数を設定している。そして，GPU での計算結果を CPU 側のメモリへ転送し（34 行目），最後にすべてのメモリの解放（39 ～ 44 行目）を行う。

プログラム３とプログラム４をコンパイルして実行してみると，CPU と比較して大きな速度改善が得られないことに気付くだろう。プログラム３の問題点を考えてみると，プログラム３では複数のスレッドが同じメモリ領域（同じ行列の要素）を利用するにもかかわらず，スレッドごとに独立してメモリアクセスを行っていることに気付く。CUDA ではグローバルメモリへのアクセスが非常に遅いため，このメモリアクセスがボトルネックになっていると考えられる。そこで，GPU 内のスレッド間でデータを共有しながら行列積を計算するようにプログラム４を改良してみよう。ここで，CUDA にはスレッド間でデータを共有する仕組みとして，共有メモリが

プログラム4　行列積の計算を行うためのCPU側の処理

```
1  int main( int argc, char *argv[] )
2  {
3      int rA = 512;            // 行列Aの行数
4      int cA = 512;            // 行列Aの列数
5      int rB = cA;             // 行列Bの行数
6      int cB = 512;            // 行列Bの列数
7      float *hA, *hB, *hC;     // CPU側で利用するメモリへのポインタ
8      float *dA, *dB, *dC;     // GPU側で利用するメモリへのポインタ
9
10     // CPU側のメモリを確保
11     hA = ( float * )malloc( rA * cA * sizeof( float ) );
12     hB = ( float * )malloc( rB * cB * sizeof( float ) );
13     hC = ( float * )malloc( rA * cB * sizeof( float ) );
14
15     // GPU側のメモリを確保
16     cudaMalloc( ( void ** )&dA, rA * cA * sizeof( float ) );
17     cudaMalloc( ( void ** )&dB, rB * cB * sizeof( float ) );
18     cudaMalloc( ( void ** )&dC, rA * cB * sizeof( float ) );
19
20     /* ここで行列の各要素に値を設定 */
21
22     // CPU側のメモリをGPU側へ転送
23     cudaMemcpy( dA, hA, rA * cA * sizeof( float ), cudaMemcpyHostToDevice );
24     cudaMemcpy( dB, hB, rB * cB * sizeof( float ), cudaMemcpyHostToDevice );
25
26     // 実行するGPUのスレッド数，ブロック数を設定
27     dim3 nThreads( 16, 16 );
28     dim3 nBlocks( rA / nThreads.x, cB / nThreads.y );
29
30     // GPUのカーネルを実行し，C = A × B の結果を dC に格納
31     multiply<<< nBlocks, nThreads >>>( dA, dB, dC, rA, cA );
32
33     // GPUの計算結果をCPU側へ転送
34     cudaMemcpy( hC, dC, rA * cB * sizeof( float ), cudaMemcpyDeviceToHost );
35
36     /* 計算結果 hC の値をここで確認 */
37
38     // CPUとGPUそれぞれのメモリを解放
39     cudaFree( dA );
40     cudaFree( dB );
41     cudaFree( dC );
42     free( hA );
43     free( hB );
44     free( hC );
45
46     return( 0 );
47 }
```

用意されていることを思い出していただきたい。この共有メモリを有効に活用するために，プログラム5では**C**=**A**×**B**の計算を部分行列の積に分解して処理を行う。

プログラム5では，行列**A**と**B**を16×16の部分行列の集合に分解して計算を行う。まず，行列**A**と**B**の各部分行列を12～13行目で共有メモリに読み込む。9～10行目では，"**__shared__**"を変数宣言の先頭に付加することで，tAとtBを共有メモリとして宣言している。ここで，共有メモリはブロック内でのみ共有可能であり，異なるブロック間では共有することができないことに注意が必要である。次に，15行目でブロック内のスレッドの同期をとり，スレッド間で共有するデータの同期をとっている（"**__syncthreads**"はブロック内のスレッドの同期をとる関数であり，ブロック間でスレッドの同期をとることはできない）。そして，共有メモリ内のデータを用い，各スレッドが部分行列の積を計算している（17～20行目）。16×16の部分行列を共有メモリに読み込むことにより，部分行列の積を求めるのに必要なデータをスレッド間で共有することができる。共有メモリへのアクセスは非常に高速（GPU内のレジスタとほぼ同じ速度でアクセス可能）であるため，部分行列の積は非常に高速に計算することができる。ただし，共有メモリを利用する際はBank Conflictに注意が必要であり，Bank Conflictが発生する場合はパフォーマンスが著しく低下する可能性がある。興味をもたれた読者は，"CUDA Programming Guide" [8]のBank Conflictに関する項目を参照していただきたい。

プログラム5　行列積を行うGPU関数（共有メモリ版）

```
1   __global__ void multiply( float *A, float *B, float *C, int rA, int cA )
2   {
3       int c = threadIdx.x + blockIdx.x * blockDim.x;
4       int r = threadIdx.y + blockIdx.y * blockDim.y;
5
6       float sum = 0.0f;
7       for( int k = 0 ; k < cA ; k += 16 )
8       {
9           __shared__ float tA[16][16];
10          __shared__ float tB[16][16];
11
12          tA[threadIdx.y][threadIdx.x] = A[r + ( k + threadIdx.x ) * rA];
13          tB[threadIdx.y][threadIdx.x] = B[( k + threadIdx.y ) + c * cA];
14
15          __syncthreads( );
16
17          for( int t = 0 ; t < 16 ; t++ )
18          {
19              sum += tA[threadIdx.y][t] * tB[t][threadIdx.x];
20          }
21
22          __syncthreads( );
23      }
24
25      C[c * rA + r] = sum;
26  }
```

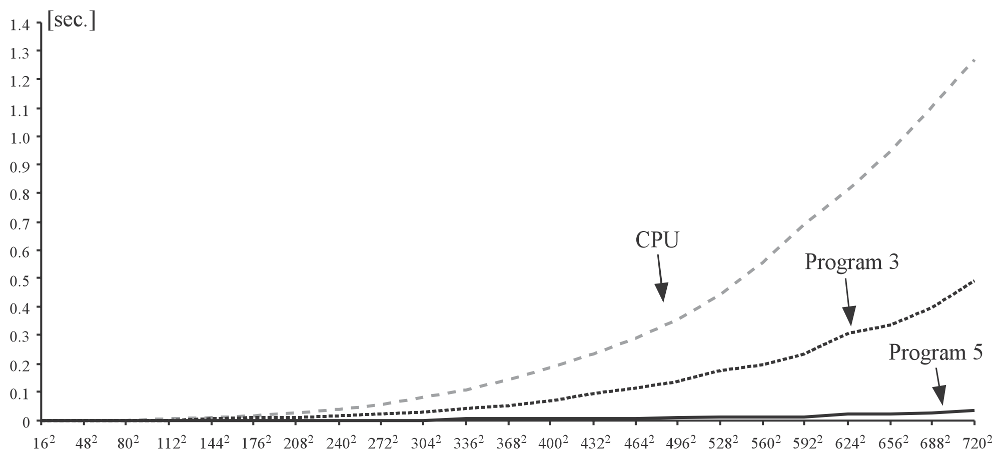


図３　プログラム３，プログラム５，CPU，それぞれで **C=A×B** の計算に要した時間。
行列 **A**，**B**，**C** は正方行列であり，グラフの横軸はその大きさを示している。

プログラム３と５の性能を比較した結果を図３に示す。図３のグラフは，Dell Precision Workstation T7400（CPU: Intel Quad Core Xeon 3.20 GHz×2，nVidia Quadro FX5600, 4.0 GB RAM, Windows XP SP2）の環境で計測した結果である。図３では，プログラム３と５の計算時間に加え，CPU で計算を行った場合の時間も示している。ただし，CPU での計算はシングルスレッドで行っている。この結果から分かるように，プログラム３をプログラム５に変更することで計算性能が大幅に改善することが確認できる。例えば，行列 **A**，**B**，**C** の大きさが 512×512 の場合，CPU は 404.5 ms.，プログラム３は 191.6 ms.，プログラム５は 12.0 ms.，の計算時間を要している。つまり，プログラム５は CPU と比べて約 33 倍，プログラム３と比べて約 16 倍高速に計算できることが分かる。アルゴリズムの工夫次第では，より高速に行列積を計算することも可能である。興味のある読者は，さらなる高速化にチャレンジして欲しい。

## Ⅴ．その他の応用例

医療の現場で利用されている CT 装置や MRI 装置等により得られるボリュームデータの可視化手法として，ボリュームレンダリングと呼ばれる可視化技術が広く利用されている。物体の表面形状のみを可視化するサーフェスレンダリングとは異なり，ボリュームレンダリングは物体表面に加え，物体内部の情報も可視化することが可能な技術である。しかしながら，ボリュームレンダリングでは，非常に多くの画素（サンプル点）に対して色や不透明度を計算する必要があるため，その計算コストは非常に高く，高精細な可視化画像を実時間で生成することは難しい。しかしながら，レイキャスティングを利用したボリュームレンダリングでは，各レイごとに独立して計算を行うことが可能である。そこで，各レイの計算を CUDA を用いて並列化することにより，高速なボリュームレンダリングを実現することができる。例えば，OS: WindowsXP，CPU: Intel Quad-Core Xeon 3.20 GHz，Memory: 3.0 GB，GPU: NVIDIA Quadro FX5600×2 の計算機環境では，図４に示すような画像を実時間で生成することができることを確認している。また，

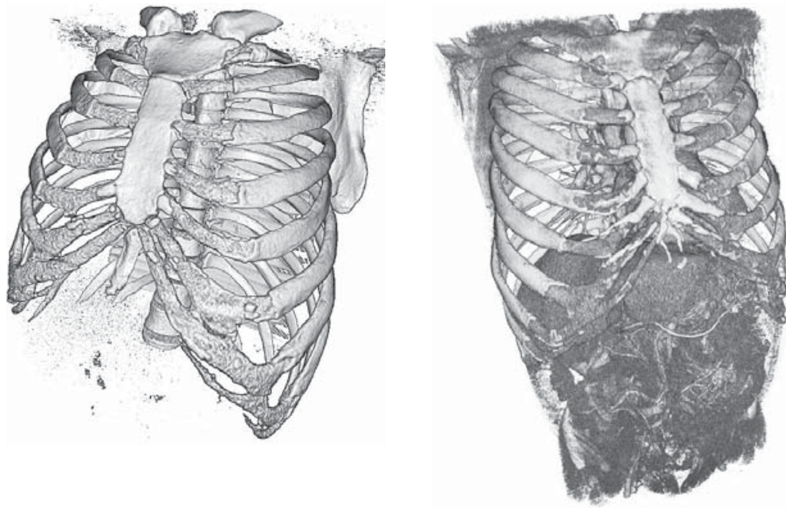

図４　３次元Ｘ線CT像をボリュームレンダリングした結果。

同様のアルゴリズムをCPUを用いて実装した場合と比較した結果，CUDAを利用したボリュームレンダリングはCPUの10倍以上の速度で画像を生成することが可能であった。

上述のボリュームレンダリングに限らず，医療分野，信号処理，数値計算などの様々な分野に対してCUDAを利用しようという試みがなされている。そのいくつかが，CUDAのホームページ[6]にて紹介されている。興味のある読者は,最新のCUDAの動向をチェックしてみて欲しい。また，CUDA以外のGPGPUに関しては文献［5,7］で数多く紹介されている。GPGPUに挑戦する際は一読することをお勧めする。

## Ⅵ．むすび

本稿ではCUDAを使用したGPGPUプログラミングについて解説した。冒頭で述べたようにGPUの処理能力は年々向上してきており，今後もその性能向上は続くと見られている。CPUと比較して何十倍も高速に計算を行うことができ，また，そのような環境が非常に手頃な価格で手に入るという点は，GPGPUの大きなメリットである。特に，スーパーコンピュータのような高性能な計算機環境が必要であったものが，我々が普段利用しているPC上で実行できる可能性があるという点は非常に興味深い。

非常に魅力的なGPGPUではあるが，現段階ではいくつかの制限が存在する。その１つが，現在のGPUは32ビットの単精度浮動小数点演算と整数演算しか扱うことができない点である。倍精度の浮動小数点演算を扱うことができないため，現状では精度の要求される計算にGPUを利用することはできない。しかしながら，2008年6月16日に発表された最新のGPUでは倍精度の浮動小数点演算がサポートされており，2008年度中には我々の手元に届く予定である。この問題を気にされている読者は，最新のGPUが入手できるようになるまで，今しばらくお待ちいただきたい。

本稿では紙面の都合上，CUDAの詳細については深く触れることはできなかった。特に，共有メモリを使用する際に問題となるBank Conflictや，テクスチャメモリなどのキャッシュが有効なメモリの利用方法，複数のGPUを同時に利用する方法，などはCUDAを利用する上で理解しておくべき項目である。興味をもたれた読者は“CUDA Programming Guide”[8]を読み，CUDAに対する理解を深めていただきたい。最後に，本稿がGPGPUへと踏み出す第一歩となれば幸いである。

参考文献

[1] http://www.intel.co.jp/jp/performance/server/xeon/hpcapp.htm

[2] “TOP 500,” http://www.top500.org

[3] J. N. England, “A system for interactive modeling of physical curved surface objects,” Proceedings of SIGGRAPH 78, pp.336−340. 1978

[4] M. J. Harris, G. Coombe, T. Scheuermann, and A. Lastra, “Physically-Based Visual Simulation on Graphics Hardware,” Proceedings of SIGGRAPH 2002 / Eurographics Workshop on Graphics Hardware 2002, pp.1−10, 2002

[5] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, “A Survey of General-Purpose Computation on Graphics Hardware,” Computer Graphics Forum, Vol.26, No.1, pp.80−113, 2007

[6] “CUDA ZONE,” http://www.nvidia.com/object/cuda_home.html

[7] “GPGPU,” http://www.gpgpu.org/

[8] “CUDA Programming Guide,” http://www.nvidia.com/object/cuda_develop.html

（にむら　ゆきたか：名古屋大学大学院情報科学研究科）

（でぐち　だいすけ：名古屋大学大学院工学研究科）