社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2008－ARC－177 (19)
2008－HPC－114 (19)
2008／3／5

メッセージ通信型GPGPUプログラミング

大島聡史† 平澤将一† 本多弘樹†

GPUの性能向上に伴い，GPUの性能を様々な用途に活用するGPGPUが注目されている．GPGPUは特に並列処理に適しているためCPUを超える演算性能が期待される一方，独自のプログラミング手法を用いる必要があるためソフトウェアの作成が容易ではない．そこで我々は既存の並列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．本稿ではメッセージ通信をはじめとした既存の並列プログラミング手法をいくつかとりあげ，それらがGPGPUプログラミングにどのように適用できるかを検討した．

Message Passing GPGPU Programming

SATOSHI OHSHIMA,† SHOICHI HIRASAWA† and HIROKI HONDA†

As GPU's performance increases, general-purpose computation using GPU (GPGPU) is watched with keen interest more and more. GPGPU is expected to overtake CPU's performance by its parallel processing tendency, but, programming for GPGPU is not easy because of its special programming style. In this paper, we propose GPGPU programming style using existing parallel programming style. We take up several existing parallel programming styles such as message passing, and examined how they can be applied to GPGPU programming.

1. はじめに

近年，高度な画像処理の要求に伴いGPU(Graphics Processing Unit)の性能が著しく向上している[1]．GPUはCPU(Central Processing Unit)と比べて並列処理やベクトル処理に適したハードウェア構成であることから，GPUを画像処理以外の汎用演算に利用するGPGPU(General-Purpose computation using GPUs)[2]への注目が高まっている．

今日ではコンシューマPCの多くにGPUが搭載されているのに対し，GPGPUの活用は進んでいない．その大きな理由の1つにGPGPUプログラミングの難しさが挙げられる．従来のGPGPUプログラミングにおいては，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の技術が必要とされてきた．こうしたプログラム作成手法はグラフィックスプログラミングに慣れていないユーザにとって容易ではなく，GPU活用の妨げとなっている．そのため，現在ではGPGPU向けのプログラミング言語やライブラリなども作られている．しかしながらこれらを利用するには新たに言語仕様などの習得が必要となるため，GPGPUプログラミングの難しさを解消するには至っていない．

我々は既存の並列プログラミング手法を用いたGPGPUプログラミングについての研究を行っている[4]．GPUは並列処理に適したハードウェアであるが，GPGPUプログラミングにおいては既存の並列プログラミング手法とは異なる手法が利用されている．既存の並列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有効に活用することが可能となる．また，既存の並列プログラムを高性能なGPU上で実行するためのプログラム変更を少なくすることができる可能性もある．

本稿ではメッセージ通信をはじめとした既存の並列プログラミング手法をいくつかとりあげ，それらがGPGPUプログラミングにおいてどのように利用できるかを検討する．

本稿の構成を以下に示す．2章ではGPGPUプログラミングの現状と我々が提案している既存の並列プログラミング手法を用いたGPGPUプログラミングの概要を述べる．3章では一般的なGPUにおける既存の並列プログラミング手法の利用方法を検討する．4章ではCUDAを対象としてより具体的な実装方法を検討する．5章はまとめの章とする．

† 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

2. GPGPU プログラミング

2.1 既存のプログラミング手法

GPGPU は，並列処理やベクトル処理に適した GPU を利用し，汎用計算（General-Purpose computation）の高速化を目指すものである．

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，また処理ユニットの行う処理を記述するために HLSL，GLSL，Cg といった専用の言語（シェーダ言語）が用いられている．

GPGPU プログラミングにおいてもこれらの言語や API が用いられてきたが，画像処理以外の GPGPU プログラムを記述するには適していない．GPU で演算を行うには，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が必要である．そのためグラフィックスプログラミングと GPU アーキテクチャについての知識が必要である．

一方，近年ではグラフィックスプログラミングの知識を必要とせずに 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．RapidMind[5] や PeakStream[6] は C/C++を元に独自の拡張を行った言語であり，GPU のみならずマルチコアプロセッサなどにも対応している．また GPU ベンダーによる GPGPU 開発環境の強化への取り組みも進められており，GeForce シリーズを扱う NVIDIA は CUDA[3] を，Radeon シリーズを扱う AMD は CTM や Brook+[7] を推進している．

これらの言語や開発環境はいずれもグラフィックスプログラミングと比べて容易に GPGPU プログラミングを行うことができる．しかしながらこれらを使うためにも新たな言語仕様や実行モデル，プログラミング手法等を習得する必要がある．また GPGPU 専用の言語は作成したプログラムを実行可能な環境（ハードウェア）が限られる．

以上のとおり，GPGPU のための既存のプログラミング手法の特殊性が GPGPU の普及の足かせとなっている．

2.2 既存の並列プログラミング手法を用いた GPGPU プログラミング

我々は GPGPU プログラミングを容易にするための手法として，既存の並列プログラミング手法を利用した GPGPU プログラミングを提案している[4]．

CPU での並列化では，SIMD，SMP，PC クラスタ，Grid など様々な環境に向けて数多くの研究が行われている．また並列プログラミングの実装においては，SIMD 命令，OpenMP，Pthread，MPI や GridRPC などが用いられている．

GPGPU においてもこれら並列プログラミングのための言語やライブラリ，プログラミング手法が利用できれば，GPU の持つ高い計算性能を容易に活用できるようになると期待できる．

3. GPU と既存の並列プログラミング手法

本章では GPGPU に用いる計算環境について確認し，既存の並列プログラミング手法と対応付ける．

GPU は内部の演算器にハードウェアレベルでの高い並列性を備えており，またメインメモリと独立した階層性のあるメモリを搭載している．複数の GPU を搭載した PC 環境や，CPU と GPU を搭載した PC を複数台利用した PC クラスタを構築することもできる．そのため，GPGPU プログラミングを考える上ではこれら多くの CPU，GPU とメモリの活用を視野に入れる必要がある．

一方で，複数 CPU 間での並列処理や PC 間での並列処理には既存の並列プログラミング手法が利用できる．以上から，本稿では以下の 2 つの並列処理を考える．

CPU-GPU 間での並列処理　GPGPU における典型的な GPU の活用方法は，対象となる処理を CPU タスクと GPU タスクに分割し，CPU と GPU で並列処理を行うものである．この場合における CPU-GPU 間での通信と演算の流れは，メインメモリ上のデータを GPU 上のメモリへ転送し，CPU と GPU がそれぞれ演算を行い，GPU 上のメモリからメインメモリへ演算結果などのデータを転送するというものになる．そのためメインメモリと GPU 上のメモリの間における通信や GPU の実行制御には，分散メモリ型並列計算機向けの並列プログラミング手法が相性が良いと考えられる．(図 1-a)

GPU 内部の並列処理　GPU の持つ高い並列実行性能を活用するには，GPU 内部の動作を並列プログラムとして記述することとなる．現在の GPU には GPU 内の演算器全てもしくは一部により読み書き可能な共有メモリが搭載されている．そのため共有メモリ型並列計算機向けの並列プログラミング手法が相性が良いと考えられる．ただし，どのようなメモリが搭載されているか，どのようにデータ参照ができるかなどは，対象とする GPU のアーキテクチャやドライバなどに依存する．そ

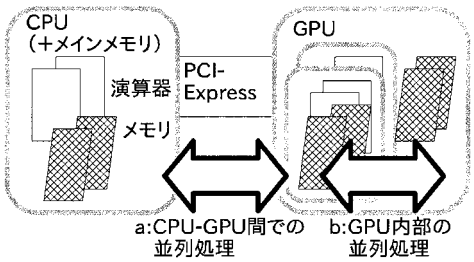

図 1 CPU-GPU 間での並列処理と GPU 内部の並列処理
Fig. 1 Parallel processing between CPU and GPU, and inner GPU

のため，実行環境や対象アプリケーションに応じて，分散メモリ型並列計算機向けの並列プログラミング手法と共有メモリ型並列計算機向けの並列プログラミング手法の両方を検討すべきである．(図 1-b)

4. CUDA における既存の並列プログラミング手法の利用の検討

本章では GPGPU 環境の 1 つである CUDA をとりあげ，既存の並列プログラミング手法の適用方法について検討する．

4.1 NVIDIA CUDA

CUDA は GPU をデータ並列計算機として活用するためのハードウェアとソフトウェアのアーキテクチャであり，C/C++を拡張した記法で GPGPU プログラムを作成することができる．バイナリは専用のコンパイラ (nvcc) を用いて生成する．CPU と GPU が実行するプログラムは別々のソースコードとして書くことも同一のソースコード中に収めることも可能であり，また GPU 向けのバイナリは CPU 向けのバイナリと個別に出力して実行時に読み込むことも，まとめて 1 つのファイルとして出力し実行することも可能である．

本節では次節以降での検討に向けて，CUDA のモデルおよび標準的な C/C++との記法の違いを確認する．

CUDA の実行モデルは，以下の処理を繰り返すモデルである．

(1) メインメモリ上のデータを GPU 上のメモリへと転送する
(2) GPU に対して関数実行を指示し，GPU が演算を開始する
(3) GPU の演算が終了した後に GPU 上のメモリからメインメモリへ演算結果などのデータを転送する

上記 (1)〜(3) の処理は CPU 側のプログラムによって起動される．メインメモリと GPU 上のメモリの間におけるデータの送受信については，CPU 上のプログラムから GPU 上のメモリのアドレスを直接参照することはできないため，メモリの確保や転送を行う専用の関数を用いる必要がある．

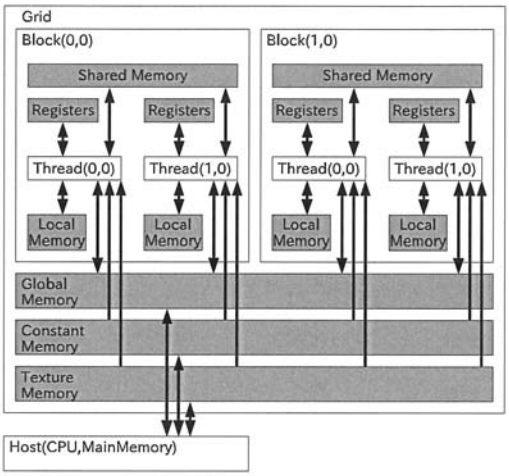

図 2 CUDA のメモリモデル
Fig. 2 Memory model of CUDA

GPU のハードウェアモデルを図 3 に示す．

GPU は Multiprocessor と呼ばれる演算ユニットを複数個（GPU によって異なるが，現在は GPU1 基あたり 1 から 16 ユニット）搭載しており，更に Multiprocessor 内には Processor と呼ばれる演算器を 8 個搭載している．同一 Multiprocessor 内の Processor は SIMD の 1 演算器として動作する．

これに対して並列実行の単位としては Grid，Block，Thread がある．

Grid は CPU が GPU に関数を実行させる際の単位となる．Grid は複数の Block から構成され，更に Block は複数の Thread から構成される．

Block は Multiprocessor に，Thread は Processor に割り当てられて実行される．

Multiprocessor や Processor に対して物理的に並列実行可能な数以上の Block や Thread が割り当てられた場合は，時分割実行される．

各 Multiprocessor は独立に処理を行えるため，Block は既存の CPU プログラミングにおけるスレッドやプロセスに相当する．同一 Multiprocessor 内の各 Processor は SIMD の 1 演算器として動作するため，Thread は既存の CPU プログラミングにおける SIMD 命令中の 1 演算に相当する．

GPU 上のメモリは以下のように分類される．

(1) Global メモリ：全 Block および全 Thread か

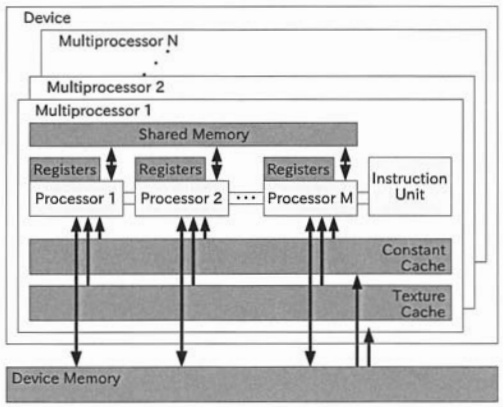

図 3 CUDA のハードウェアモデル
Fig. 3 Hardware model of CUDA

ら共有メモリとして読み書き可能.(高レイテンシ・低速・大容量)

(2) Constant メモリ：全 Block および全 Thread から読み取り専用のメモリとして扱うことができる.(高レイテンシ・キャッシュされる・64KB)

(3) Texture メモリ：全 Block および全 Thread から読み取り専用のメモリとして扱うことができる.(高レイテンシ・キャッシュされる・大容量)

(4) Shared メモリ：各 Block が独立に持ち同一 Block 内の Thread からは共有メモリとして読み書き可能.(オンチップ・低レイテンシ・高速・16KB)

(5) 各 Thread が独立に持つ Register.(低レイテンシ・高速・Multiprocessor ごとに 8192 本)

(6) 各 Thread が独立に持つ Local メモリ.(Global メモリと同等)

Global メモリ，Constant メモリ，Texture メモリについては専用のメモリ操作関数を用いることで CPU から読み書きすることができる.

CUDA では変数宣言時に接頭語を付加することで，その変数がどのメモリとして扱われるかを指定する.__device__をつけることで Global メモリ，__constant__で Constant メモリ，__shared__で Shared メモリとして扱われる．Texture メモリは専用の関数と構造体を用いて利用する．接頭語をつけずに宣言された変数は Register として扱われ，Register の数を超えた分は自動的に Local メモリとして扱われる.

変数と同様に，関数についても接頭語を利用して振る舞いを指定する.

__global__をつけた関数（Global 関数）は CPU から呼び出されて GPU 上で実行される関数，__device__をつけた関数（Device 関数）は GPU 上で実行される関数から呼び出されて GPU 上で実行される関数として扱われる．これらの関数は GPU 上で実行可能なバイナリのみが生成される.

__host__をつけた関数（Host 関数）は CPU 上で実行される関数から呼び出されて CPU 上で実行される関数として扱われ，接頭語のない関数もこれと同じ扱いとなる．__host__と__device__を同時につけた場合は，__device__をつけた関数と同様に GPU 上で実行可能なバイナリが生成されるとともに，__host__をつけた関数と同様に CPU 上でも実行可能となる.

CPU から GPU に対して Global 関数の実行を指示するには，funcname<<<Dg, Db>>>(args) といった形式の記述を利用する．funcname には GPU 上で実行される Global 関数名，Dg と Db には GPU 上での実行における並列数（Block と Thread をいくつ使うか），args には実行時引数列を与える．これにより，指定した数の Block,Thread が funcname 関数を同時に実行する．funcname 関数内では Block や Thread ごとに割り当てられた ID を利用して並列処理を行う．なお CPU が検出できるのは全ての Block および Thread が処理を終えた場合とエラーが発生した場合であり，特定の Block や Thread の実行中の状態を知ることはできない.

Global 関数の呼び出しは関数の終了を待たずに制御が戻る非同期呼び出しである．そのためマルチスレッド化などを伴わずに CPU と GPU による並列処理が可能である．CPU 上で専用の関数を実行することで，Global 関数の終了を確認する（待つ）ことができる.

GPU 上で実行される関数については，再帰呼び出しや static 変数が使えないといった制限がある.

一方で基本的な文法などについては C/C++を利用できるため，既存アルゴリズムの移植等は比較的容易に行うことができる.

ただし，GPU を最大限に活用して高い性能を得るためには，データを GPU 上のどのメモリに配置するかなどを細かく制御する必要があり，容易ではない.

4.2 CPU-GPU 間での並列処理への適用

CUDA を用いた GPGPU プログラミングにおいて，CPU-GPU 間での並列処理に既存の並列プログラミング手法を用いる方法を検討する.

CPU と GPU の間に共有メモリなどの機構は存在しないため，分散メモリ型並列計算機向けのプログラミング手法を用いることを考える.

4.1 節で述べたように，CUDA における GPU の利用は

(1) メインメモリ上のデータを GPU 上のメモリへ

```
// GPU上で実行される関数
__global__ void gpufunc( float *in, float *out ){
  for(int i=0; i<size; i++)out[i] = in[i] * 2.0f;
}

int main(int argc, char **argv){
  CUT_DEVICE_INIT();
  float data[size];
  float *d_in, *d_out;
  for(int i=0; i<size; i++)data[i] = 1.0f;
  // メインメモリからGPU上のメモリへの転送          -(1)
  cudaMalloc((void**)&d_in,sizeof(float)*size);
  cudaMalloc((void**)&d_out,sizeof(float)*size);
  cudaMemcpy(d_in, data, sizeof(float)*size,
             cudaMemcpyHostToDevice);
  // 関数実行の指示と終了待ち                      -(2)
  dim3 threads(1,1,1), grid(1,1,1);
  gpufunc<<<grid, threads>>>(d_in, d_out);
  cudaThreadSynchronize();
  // GPU上のメモリからメインメモリへの転送          -(3)
  cudaMemcpy(data, d_out, sizeof(float)*size,
             cudaMemcpyDeviceToHost);
  for(int i=0; i<size; i++)printf("%f\n", data[i]);
  cudaFree(d_in);
  cudaFree(d_out);
  CUT_EXIT(argc, argv);
  return 0;
}
```

図 4 簡単な CUDA プログラムの例
Fig. 4 Example program of CUDA

と転送する

(2) GPU に対して関数実行を指示し，GPU が演算を開始する

(3) GPU の演算が終了した後に GPU 上のメモリからメインメモリへ演算結果などのデータを転送する

を 1 単位として行われる (図 4).

この実行モデルに対応するものとして，RPC や，コプロセッサやアクセラレータを用いる際の関数呼び出しが挙げられる．例えば送受信に用いるデータ（メモリアドレス）の指定や Global 関数の実行制御をラップした関数を作成することで，GPU を RPC の実行モデルで容易に利用することができると考えられる．

また，CUDA には GPU 上で実行する関数をオブジェクトファイルとして外部に持ち，プログラム実行開始後に読み込み登録・実行する機構がある．これを利用することで，ライブラリの登録・リモート関数の実行と結果の取得という GridRPC[8] サーバ等で用いられる実行モデルも利用可能となると考えられる．

分散メモリ型並列計算機環境で多く用いられているプログラミング手法として，MPI に代表されるようなメッセージ通信が挙げられる．しかし CUDA の実行モデルでは，Global 関数内においてメインメモリと GPU 上のメモリの間でデータの送受信を行うことが想定されていない．メッセージ通信を利用可能にするためには，Global 関数内で自由に送受信を行えるようにする機構等を用意する必要がある．

4.3 GPU 内の並列処理への適用

続いて，GPU 内の並列処理に既存の並列プログラミング手法を用いる方法を検討する．

3 章および 4.1 節で述べたように，GPU 内には特性の異なる複数種類のメモリが存在する．特に CPU-GPU 間での並列処理とは異なり，CUDA における GPU 内の並列処理では全 Block および全 Thread から共有メモリとして利用可能な Global メモリを利用することができる．そのため，共有メモリ型並列計算機向けのプログラミング手法と分散メモリ型並列計算機向けのプログラミング手法の両方が利用できると考えられる．

はじめに，CUDA を直接利用した場合のプログラミング手法を確認する．CUDA における典型的な GPU 内の並列プログラミング手法は以下の通りである．各 Block は Global メモリから Shared メモリにデータをコピーし，演算を行い，結果を Global メモリに書き戻す．データのコピーや Block 内の演算は複数の Thread を用いて並列に実行し，更にこれら一連の処理を複数の Block が並列に実行する．このプログラミング手法は，演算器とメモリに階層性があることを意識してプログラムを作成する必要があるという点で，既存の CPU で用いられているプログラミング手法と比べると煩雑である (図 5-a)．

そこで，既存の並列プログラミング手法を用いて演算器やメモリの階層性を隠蔽する手法を考える．なお，単純化のために Constant メモリと Texture メモリについては扱わないことにする．

手 法 1

Global メモリを介したメッセージ通信を実装する．Block 間のデータ授受では Global メモリを直接読み書きする代わりにメッセージ通信を利用し，Block 内での処理には Thread，Shared メモリ，Register および Local メモリを利用する．1 つの Block が Global メモリを直接操作して他の Block に演算用のデータとして送信し，他の Block は演算用データの送受信と演算を行うというサーバクライアントモデルを考えることもできる．本手法はメモリ階層を 1 階層意識する必要がなくなるため，プログラムの作成が容易になる可能性がある．

一方で，メッセージ通信と共有メモリではプログラミング方法やデバッグ方法等に違いがあり，どちらの手法が良いかについてはプログラマの好みにも左右される．また，本手法ではメッセージ通信と共有メモリを併用することになるため，新たなプログラム作成の難しさが生じてしまう可能性もある (図 5-b)．

手 法 2

Thread による並列処理が SIMD 演算に対応付けられることを利用する．複数 Thread が Shared メモリを利用して SIMD 演算を行う関数を作成し，プロ

```
a: CUDAを直接利用したプログラムの例
__device__ float g_in[PROBLEM_SIZE];
__device__ float g_out[PROBLEM_SIZE];
__global__ void gpufunc()
{
 __shared__ float my[MYDATA_SIZE];
 float tmpdata = 0.0f;
 int bid = blockIdx.x;  // Block判別用
 int tid = threadIdx.x; // Thread判別用

 for(int i=begin; i<end; i+=step){
  // GlobalメモリからSharedメモリにデータをコピー
  my[hoge*tid + i] = g_in[foo*bid + bar*tid + i];
  __syncthreads();
  // sharedメモリにコピー済みのデータを用いて演算
  for(int k=0; k<mysize; k+=1){
   tmpdata = my[fuga * k + tid] * ...;
  }
  __syncthreads();
  // SharedメモリからGlobalメモリへデータをコピー
  g_out[Foo*bid + Bar*tid + i] = tmpdata;
 }
}


b: メッセージ通信によって
   Globalメモリの利用を一部隠蔽した例
__device__ float g_in[PROBLEM_SIZE];
__device__ float g_out[PROBLEM_SIZE];
__global__ void gpufunc()
{
 __shared__ float my[MYDATA_SIZE];
 float tmpdata = 0.0f;
 int bid = blockIdx.x;  // Block判別用
 int tid = threadIdx.x; // Thread判別用

 if(bid==0){
  // block0はサーバ
  for(int i=begin; i<end; i+=step){
   for(int m=1; m<BLOCK_SIZE; m++){     // 送信
    _send(&g_in[foo*bid + bar*tid + i],size,m,i);
   }
   for(int m=1; m<BLOCK_SIZE; m++){     // 受信
    _recv(&g_out[foo*bid + bar*tid + i],1,m,i);
   }
  }

 }else{
  // block0以外は計算コア
  // __device__変数を意識しなくて良い
  for(int i=begin; i<end; i+=step){
   // 担当データを取得
   _recv(&my[hoge*tid + i], size, 0, i);
   for(int k=0; k<mysize; k+=1){
    // 取得したデータを利用して演算
    tmpdata = my[fuga * k + tid] * ...;
   }
   // 計算結果を送信
   _send(&tmpdata, 1, 0, i);
  }
 }
}
```

図 5 書き換えられたソースコードの例
Fig. 5 Example of modified source code

グラム作成時には Block と Global メモリを用いた共有メモリ型並列計算機向けのプログラミング手法と，Register および Local メモリ，そして作成した SIMD 関数を利用する．これにより Thread と Shared メモリの存在を意識する必要がなくなり，プログラムの作成が容易になる可能性がある．

同じように，Thread と Shared メモリを用いて OpenMP のような機構を作ることでも同様の成果が得られるものと考えられる．

手 法 3

1 と 2 の手法を組み合わせる．Global メモリをメッセージ通信，Thread と Shared メモリを SIMD 演算に利用することで，演算器とメモリの階層性が隠蔽される．プログラム作成時には Block 間でのメッセージ通信と SIMD 関数，そして Register および Local メモリのみを意識すればよいため，プログラムの作成が更に容易になると考えられる．

5. おわりに

本稿では GPGPU プログラミングにおける既存の並列プログラミング手法の活用についての検討を行った．また，CUDA を対象としてより具体的な実装を検討した．

検討の結果は以下の通りである．

CPU-GPU 間での並列処理については，RPC として扱うのが CUDA の実行モデルに適していると考えられる．

GPU 内の並列処理については，メッセージ通信や SIMD 演算を利用することで CUDA プログラミングの煩雑さが軽減できると考えられる．

一方でメッセージ通信と共有メモリを同時に利用する場合などは，新たな煩雑さが生じる可能性もあると考えられる．

今後は本稿での検討を元にアプリケーションの作成やライブラリの実装などを行う．それらを用いて，プログラム作成の容易性や，GPU の性能が十分に発揮できるかについて評価を行う．また，CUDA 以外の GPGPU 環境に対しても同様の検討を行うことも視野に入れる．これらを通して，容易に利用可能で高性能な GPGPU プログラミング手法の達成を目指す．

参 考 文 献

1) gpgpu.org: SIGGRAPH 2007 GPGPU COURSE, `http://www.gpgpu.org/s2007/`.
2) gpgpu.org: General-Purpose computation on GPUs(GPGPU), http://gpgpu.org/.
3) NVIDIA: CUDA Programming Guide 1.0 (CUDA NVIDIA Homepage), `http://developer.nvidia.com/cuda/`.
4) 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).
5) RapidMind Inc.: RapidMind, `http://www.rapidmind.net/`.
6) Peakstream: The Peakstream API for GPUs, http://www.peakstreaminc.com/.
7) AMD: AMD Stream Computing, `http://ati.amd.com/technology/streamcomputing/`.
8) K.Seimour, H.Nakada, S.Matsuoka, J.Dongarra, C.Lee and H.Casanova: Overview of GridRPC: A Remote Procedure Call API for Grid Computing, *Proceedings of Grid Computing - Grid 2002*, pp. 274–278 (2002).