既存の並列化手法を用いた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は特に並列プログラムにおいてCPUを超える高い性能が期待される一方，GPGPUプログラミング特有の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．本稿ではGPGPUプログラミングを容易にする1つの手法として既存の並列化手法を利用することを提案する．また具体的な実装に向けて，近年利用され始めたGPGPU向けプログラミング言語CUDAを利用し，GPU上の処理を既存の並列化手法であるSIMD命令，OpenMP，MPIを用いて記述する方法を検討する．

GPGPU Programming
Using Existing Parallelizing Method

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

GPGPU utilizing GPU's performance for general-purpose computation is attracting much attention. GPGPU is expected to effect higher performance than CPU. However, creating GPGPU programming is not easy because programming methods peculiar to GPGPU programming are needed. In this paper, we propose to use existing parallellizing method as one of a new method making GPGPU programming easier. Also we consider writing programs running on GPUs with SIMD instruction, OpenMP and MPI based on the new GPGPU programming language of CUDA.

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と比較すると，GPGPUの活用事例は限られている．その大きな理由の1つにGPGPUプログラミングの難しさ，すなわちGPUは並列処理に適したハードウェアでありながら既存の並列化手法を容易に利用できないことが挙げられる．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GPGPUプログラミングに特有のものであるため，他の分野のユーザにとってGPGPUの活用は容易ではない．現在ではCUDA[3]のようにGPGPUプログラミングの特殊性を隠蔽するプログラミング言語なども登場しているが，特定のGPUでのみ利用可能なうえに，GPUアーキテクチャの理解と新しい言語の習得が必要である．

そこで本稿では，既存の並列化手法を用いたGPGPUプログラミングを提案する．GPUは並列性の高いプログラムの実行に適しているため，既存の並列化手法を容易にGPUへ適用する手段があれば，GPUを並列プログラムの実行環境として有効活用可能となることが期待できる．

本稿の構成を以下に示す．2章では本研究の背景にあたるGPUとGPGPUについて簡単に説明し，現状の問題点を明らかにする．3章では解決案として既存の並列化手法を利用したGPGPUプログラミングを提案する．4章では提案に対する実装例として，CUDAを用いてGPU向けに既存の並列化手法を実装する方法を検討する．5章では関連研究に触れ，6章はまとめと今後の課題の章とする．

† 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

2. GPUのアーキテクチャと既存のGPGPUプログラミング手法

GPU は本来，高速に画像描画を行うために発展したハードウェアである．現在では動画再生支援などの処理も統合される傾向にあるため，性能の差異は大きいものの，一般的な PC の多くに搭載されている．

図 1 に伝統的な GPU のハードウェア構成と画像描画のための主な処理の対応を示す．GPU が行う画像描画のための主な処理は，並列計算やベクトル計算による高速化に適している．そのため GPU 上の処理ユニットは並列化が進んでおり，ベクトル計算に適した内部構造となっている．また高度な画像表現には描画内容に応じた複雑な計算が必要なため，処理ユニットのプログラマブル化が進んでいる．GPU を用いたプログラムを作成するには，DirectX や OpenGL といったグラフィックス API を用いて GPU の動作タイミング制御や CPU-GPU 間のデータ通信などを行い，HLSL，GLSL，Cg といったシェーダ言語を用いて計算ユニットの行う処理を記述する必要がある．

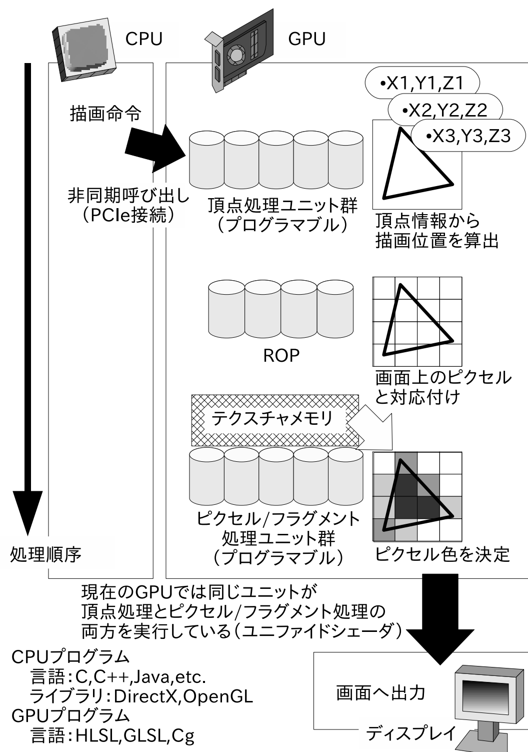

図 1 伝統的な GPU のハードウェア構成と画像描画処理の対応
Fig. 1 Hardware construction of traditional GPU and relationship with graphics processing

GPGPU は，GPU の特徴を利用して様々な処理 (汎用計算，General-Purpose computation) を行うものである．GPU のハードウェア性能を活かせる用途，すなわち並列計算やベクトル計算に適した対象問題に対しては，CPU を圧倒する演算性能を得ることができる．しかしながら，GPGPU アプリケーションの実装は容易ではない．例えば GPU を用いて数値計算を行うためには，数値計算アルゴリズムを画像描画のシステムに対応させて実装する必要がある (図 2)．そのためには数値計算プログラミングとグラフィックスプログラミングの両方に精通している必要があり，高い性能を得るためには並列化手法や GPU アーキテクチャについての知識も重要である．現在では CUDA などグラフィックスプログラミングの知識を必要としない GPGPU プログラミングのための言語やライブラリもあるが，これらを用いてアプリケーションを作成するには GPU アーキテクチャの知識が必要であったり，既存のプログラミング手法とは異なる手法を習得する必要があったりする．

今日では MAC OS X の Aqua/Quartz や Windows Vista の AeroGlass をはじめとして OS やデスクトップ環境のレベルである程度高性能な GPU を要求するケースが増加している[4)~7)]．更に動画再生支援機能を持つ GPU(ビデオカード) も増加しているため，コンシューマ向け PC の多くに GPU が搭載されている．GPU の持つ理論演算性能の高さを考慮すると，GPU の持つ全ての性能を活用することはできなくても，ある程度の性能が活用できれば様々なアプリケーションの高速化が期待できる．しかしながら，現在画像処理以外の分野において GPGPU が活用されているのは一部の数値計算や科学技術計算に限られている．その最たる理由として，GPU を活用するプログラムの作成が容易でないことが挙げられる．GPGPU に詳しくない多くのアプリケーションプログラマにとっては，GPGPU が容易に利用できることが重要である．これはもちろんコンシューマ PC に限った話ではなく，特に高性能な計算環境を求める HPC 用途でも共通の課題である．

3. 提 案 内 容

3.1 既存の並列化手法を利用した GPGPU プログラミングの提案

GPU は CPU と比べて高いハードウェアレベルの並列性を備えているため，GPU による演算性能の向上が期待されているのはもっぱら高い並列性を持つアプリケーションである．一方で並列化は SIMD，SMP，

OpenGL+GLSLを用いた典型的なGPGPUプログラムの流れ
（配列をf_hoge倍して返すプログラムの例）

= × f_hoge

CPU側プログラムの例（処理の流れ）

1. glut/GLEWの初期化
   glutInit*, glewInit
2. FBO（描画先を変更するための機能）の設定
   glGenFramebuffer*
3. テクスチャ設定 = データ送信
   glTex*
4. シェーダ設定 = 演算パラメタの指定
   gl*Object*, gl*Shader*, glUniform*
5. 描画=演算
   glBegin, glTexCoord, glVertex, glEnd
6. 描画結果の取得 = データ(演算結果)受信
   glRead*
7. 終了処理
   glDelete*

非同期呼び出し

GPU側プログラムの例

```
// CPU側に見せる各種演算パラメタの宣言
uniform samplerRECT tex1;
uniform float f_hoge;

// 演算を行う関数
void main()
{
   vec2 pos;            // xy座標を表す二次元ベクトル変数
   vec4 color;          // 色を表す四次元ベクトル変数
   pos = gl_FragCoord;           // 座標の取得
   color = texRECT(pos, tex1);  // テクスチャ色の参照
   gl_FragColor = color * f_hoge;           // 演算
}
```

図 2　描画処理を利用して計算処理を行うプログラムの例
Fig. 2　Example of numerical culculation programming using graphics programming

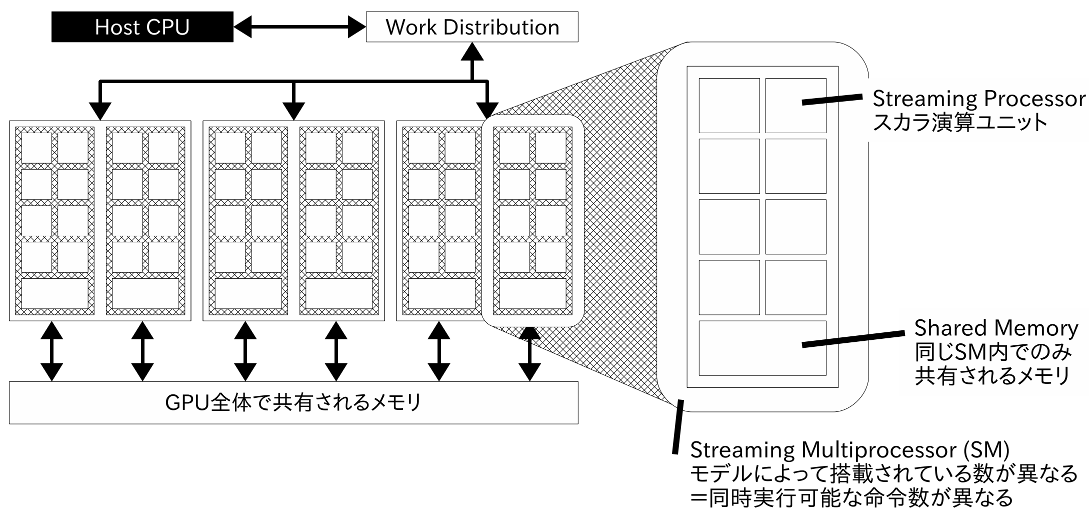

図 3　GeForce8000 シリーズのアーキテクチャ
Fig. 3　Architecture of GeForce8000 series

PC クラスタ，Grid など既に様々な研究が進められているテーマであり，現在もマルチコア CPU の普及などにより注目されている．既存の並列化手法を用いて GPGPU プログラミングを行えるようにすることができれば，GPU を既存の並列環境により近いものとして扱えるようになると考えられる．GPU が身近で使いやすいものとなることで，多くのユーザが GPU の持つ高い計算性能を有効に活用できるようになることが期待できる．

そこで，GPGPU プログラミングにおいて既存の並列化手法を利用することを提案する．本章の残りの部分では，最新 GPU のハードウェア構成を確認したうえで，既存の並列プログラミングにおいては並列化対象の粒度に応じて様々な並列化手法が用いられていることを考慮し，GPU のハードウェア構造と並列化手法との対応付けを検討する．また次章では本提案に対する実装の例として，CUDA 向けに既存の並列化手法を実装することを検討する．

3.2　GPU に対する既存の並列化手法の割り当ての検討

最新 GPU のハードウェア構造を概観し，既存の並列化手法を GPU に適用するにはどうすればよいか，既存の並列化手法で用いる実行モデルと GPU ハードウェアをどのように関連付けるかについての検討を行う．

本稿執筆時点の最新世代 GPU である GeForce8000 シリーズおよび RadeonHD2000/3800 シリーズの内部構造の概要を図 3 および図 4 に示す．これらの GPU ではプログラマブル処理ユニットの並列化が進んでおり，GPU 全体で最大 100 以上の演算を並列実行可能

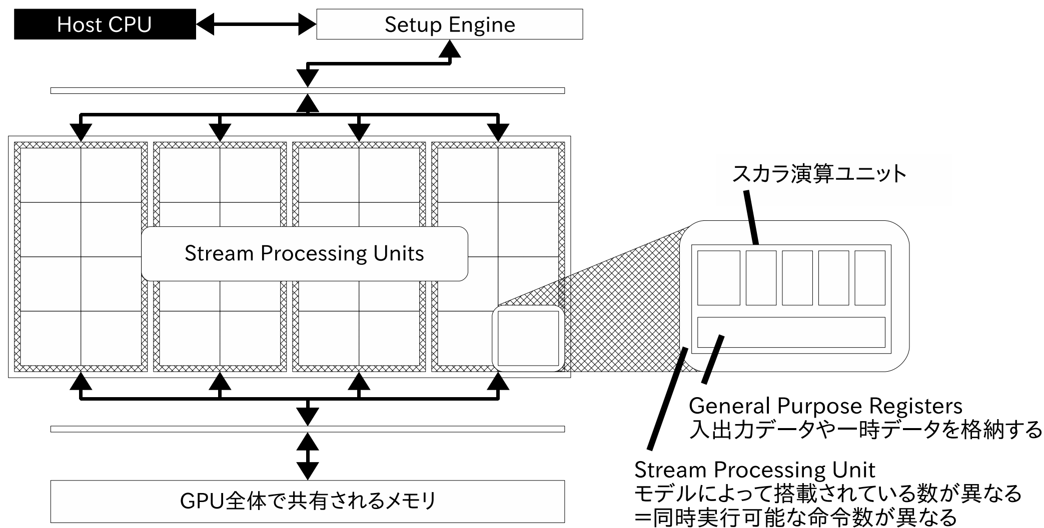

図 4 RadeonHD2000/3800 シリーズのアーキテクチャ
Fig. 4 Architecture of RadeonHD2000/3800 series

となっている．ただし，GPU 上の全ての演算器が同じ機能を持ち均等に配置されているわけではない．いくつかの演算器が集まって処理ユニットを構成しており，更に処理ユニットが集まって GPU を構成している．またメモリについても，GPU 全体から透過的にアクセス可能な (グローバルな) メモリのみを備えているわけではなく，処理ユニットごとにローカルなメモリも備えている．すなわち現在の GPU は，ハードウェアレベルでの高い並列性を備えているのみならず，搭載している演算器とメモリには階層性が備わっていると捉えることができる．

既存の並列プログラミングにおいては，データレベルの並列性，命令レベルの並列性，スレッドレベルの並列性，プロセスレベルの並列性といった並列性が利用されている．また具体的な並列プログラミングの手段としては SIMD 命令，OpenMP および各種スレッドライブラリ，MPI などが用いられている (図 5)．そこで，GPU の階層性と既存の並列化手法の階層性とを対応付け，GPU 向けにこれらの関数・ライブラリを実装することを考える．

3.2.1 GPU 向け SIMD 命令の検討

SIMD 命令は 1 度の演算で複数のデータに対して処理を行ういわゆるベクトル演算命令であり，データ並列向けの細粒度な並列化に利用されている．GPU プログラミングにおいては，画像描画の際に行う頂点や色の演算に 4 次元ベクトル演算が適していることから，ハードウェアレベルでベクトル演算に適しているのみならず，シェーダ言語にベクトル演算向けのインターフェイスが備わっている．ベクトル演算による高速化が行えるようにプログラムを組むことは GPGPU プログラミングの基本である．一方 CPU プログラミングにおいては MMX や SSE，VMX など各 CPU の備える命令セットを利用して SIMD 命令を利用する．そのため CPU の対応する命令セットごとに異なるプログラム記述をする必要がある上に，インラインアセンブラを記述する必要がある場合もあるため，GPU 向けの SIMD 命令は作成しにくい．そこで，中西ら[8)]

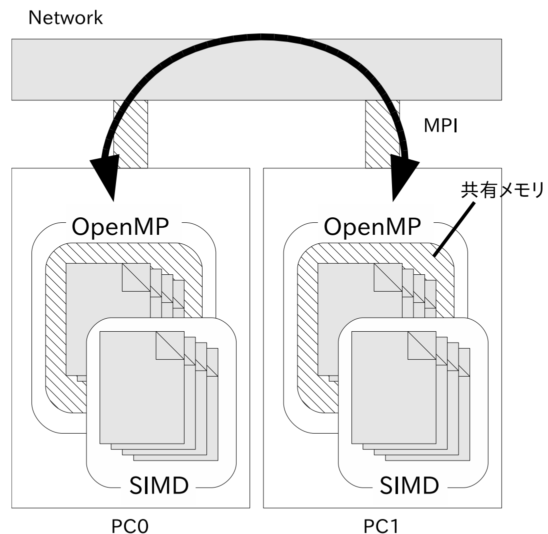

図 5 CPU における階層型並列プログラミング
Fig. 5 Hierarchical parallel programming on CPU

による記述方式を利用することを考える．この記述方式はユーザ (アプリケーションプログラマ) に対してSIMD 命令セットごとの記述の違いを隠蔽し，実行環境を問わず統一的な記述によって SIMD 化を行うことのできる方式である．実装的には提案する記述方式を用いて書かれたプログラムに対して各 CPU の対応する SIMD 命令へのプログラム変換を行うものであるため，記述方式から GPU の対応するベクトル演算への変換機能を実装することで GPU 向け SIMD 命令の実装が可能となる．

3.2.2　GPU 向け OpenMP の検討

OpenMP および各種スレッドライブラリを用いた並列化は，共有メモリを備えた環境に適した細粒度の並列化手法である．OpenMP は各種スレッドライブラリ (主に pthread) の上位に実装されておりかつ記述が容易で使いやすいため，ここでは GPU 向け OpenMP の検討に話を絞る．OpenMP では逐次プログラムのソースコードにプラグマを挿入し，コンパイル時にプラグマを解釈して共有メモリを用いたスレッド並列プログラムに変換する．ループ変数の自動書き換えによるループの並列化といった細粒度の並列化に使われることが多く，逐次プログラムの段階的な並列化にも適している．OpenMP を利用するには OpenMP のプラグマを解釈することが可能なコンパイラが必要である．対応コンパイラの代表的なものとしては Omni OpenMP Compiler[9]，Intel C++ Compiler,GCC などがあり，他にも商用コンパイラのいくつかが対応している．OpenMP コンパイラの基本的な実装は，プラグマが挿入された部分にスレッドの制御を行う記述を挿入することで実現できる．GPU には共有メモリが搭載されているため，OpenMP におけるスレッドの制御に相当する記述を GPU 上で模倣できれば GPU 向け OpenMP の実装が可能となる．

3.2.3　GPU 向け MPI の検討

MPI はメッセージ通信を利用し，複数ノード間におけるデータや制御のやり取りを行うことで並列プログラムを形成するための規格である．ノードごとに取得した rank 情報に基づいて処理を振り分けるといった使い方に適しており，OpenMP と比べて疎粒度の並列化に使われることが多い．MPI は分散メモリ環境向けの並列化手法であるが，共有メモリ環境でも特に問題なく利用することができる．むしろ通信の細かい制御が可能なため，共有メモリ環境でも OpenMP より高い性能が得られることがある．MPI の実装はコンパイラや複雑なソースコード変換機構を必要としない代わりに，(複数の PC に跨って) 複数のプロセスを立ち上げるための機構等を備えている．MPI の実装としては mpich および mpich2,LAM[10),11] などが挙げられる．GPU 向け MPI の実装については，演算ユニット間における共有メモリを介したデータや制御のやり取りに MPI のインターフェイスを利用できるようにするという実装が考えられる．また，CPU-GPU 間の通信に MPI のインターフェイスを利用できるようにすることで，CPU 上のメモリと GPU 上のメモリという分散メモリの管理を既存の分散メモリに近い記述によって扱えるようにすることができる．

以上のように，既存の様々な並列化手法に GPU を対応付けることで，GPU 内部の演算器やメモリが持つ階層性を活用した GPU 向け並列プログラムを容易に作成可能となることが期待できる．また SMP やマルチコア CPU を搭載したノードによって構成される PC クラスタなどにおいては，ノード内の並列化を OpenMP，ノード間の並列化を MPI によって行うハイブリッドな並列化などによって効率的な並列化が行われている．GPU を用いる場合においても，GPU 内の階層的な並列性を利用したハイブリッドな並列化や，GPU と CPU の並列処理など様々なレベルでの並列処理が容易に行えるようになることが期待できる．

4. CUDA を用いた GPU 向け既存の並列化手法の実装に向けた検討

本章では，CUDA を用いて GPU 向けに既存の並列化手法を実装する方法についてより具体的な検討を行う．

これまで GPU の機能を利用するにはグラフィックス API を介する必要があったため，GPGPU においては全ての処理をグラフィックスプログラミングの方式にあわせて実装する必要があった．そのため GPU プログラミングを行うためにはグラフィックスプログラミングを習得する必要があった．これに対して CUDA では多くの処理をより直感的に記述することができる (図 6)．またこれまで GPU の内部情報についてはグラフィックスプログラミングに必要な程度の情報のみが公開されていたが，CUDA とともに多くの情報が提供されるようになった．これにより GPU プログラムのデバッグや最適化に有益な情報が入手しやすくなったと言える．

図 7 に CUDA の並列実行モデルおよびメモリモデルを示す．これを元に，CUDA を用いて既存の並列化手法である SIMD 命令，OpenMP および MPI の実装を検討する．

CUDAを用いた典型的なGPGPUプログラムの流れ
（配列をf_hoge倍して返すプログラムの例）

= × f_hoge

CPU側プログラムの例（処理の流れ）

```
1. cudaの初期化
   CUT_DEVICE_INIT();
2. GPU上で利用するデータの準備
   cudaMalloc(...);
3. データ送信
   cudaMemcpy(...);
4. 演算パラメタ（並列度）の設定
   dim3 threads(x,y,z); dim3 grid(x,y,z);
5. 演算指示
   main<<<grid, threads>>>(data, f_hoge);
   cudaThreadSynchronize();
6. データ(演算結果)受信
   cudaMemcpy(...);
7.終了処理
   cudaFree(...);
   CUT_EXIT();
```

非同期呼び出し

GPU側プログラムの例

```
// 演算を行う関数
__global__ void main(float *data, float f_hoge)
{
  // 担当するデータ範囲を算出
  int id = blockIdx.x * N + threadIdx.x;
  // 演算
  data[id] *= f_hoge;
}
```

図 6　CUDA を用いた GPGPU プログラミング
Fig. 6　GPGPU programming using CUDA

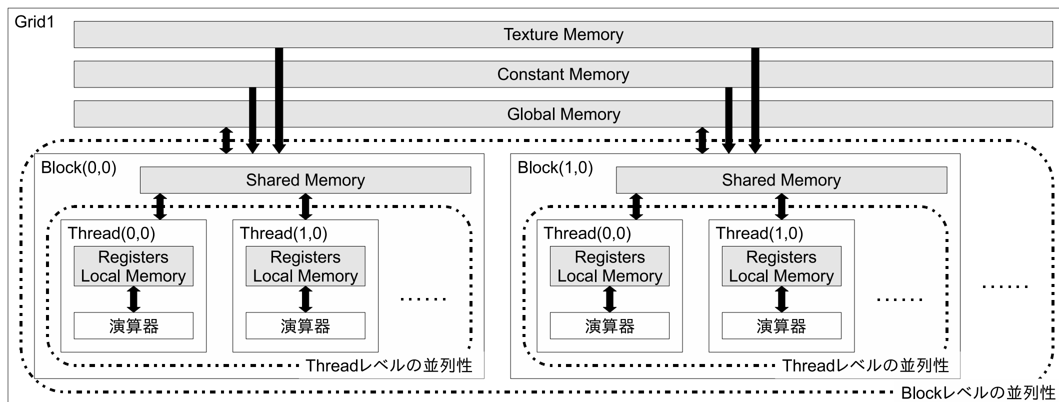

図 7　CUDA の並列実行モデルとメモリモデル
Fig. 7　Parallel execution model and memory model of CUDA

4.1　CUDA 向け SIMD 命令の実装

はじめに CUDA 向け SIMD 命令の実装について考える．CUDA における各 Block 内の Thread は，対象のデータ群に対して同一の処理を同時に実行することが可能である．これは典型的な SIMD 命令の処理と同一であるため，Thread を用いた演算に対して既存の SIMD 命令セット (共通記述方式) に対応したインターフェイスを用意することで CUDA 向け SIMD 命令が実現できると考えられる．

4.2　CUDA 向け OpenMP の実装

続いて CUDA 向け OpenMP について考える．OpenMP を用いた並列処理には，各演算器が自由にアクセスすることのできる共有メモリが必要である．CUDA には共有メモリとして利用可能な複数種類のメモリが存在する．Global Memory, Constant Memory, Texture Memory は全ての Block および全ての Thread から共有メモリとして利用可能であり，Shared Memory は同一 Block 内の全ての Thread から共有メモリとして利用可能となっている．これらのうち，Constant Memory と Texture Memory は GPU から見ると読み込み専用のメモリであるため，今回は利用対象から除外する．残る Global Memory と Shared Memory については，Global Memory を利用すれば Thread レベルと Block レベル双方で，Shared Memory を利用すれば Block レベルで OpenMP の並列処理を置き換えられると考えられる．よって，これらのメモリを利用して OpenMP の機能を実現することで，CUDA 向け OpenMP が実装できると考えられる．

OpenMP は細粒度な並列処理に利用するため，Block レベルの並列処理よりも Thread レベルの並列処理に適していると考えることができる．一方で GPU 内部のメモリ転送速度は CPU-メインメモリ間よりも高速であり，また Block レベルの並列処理は Thread レベルの処理と比べて処理の自由度が高い．

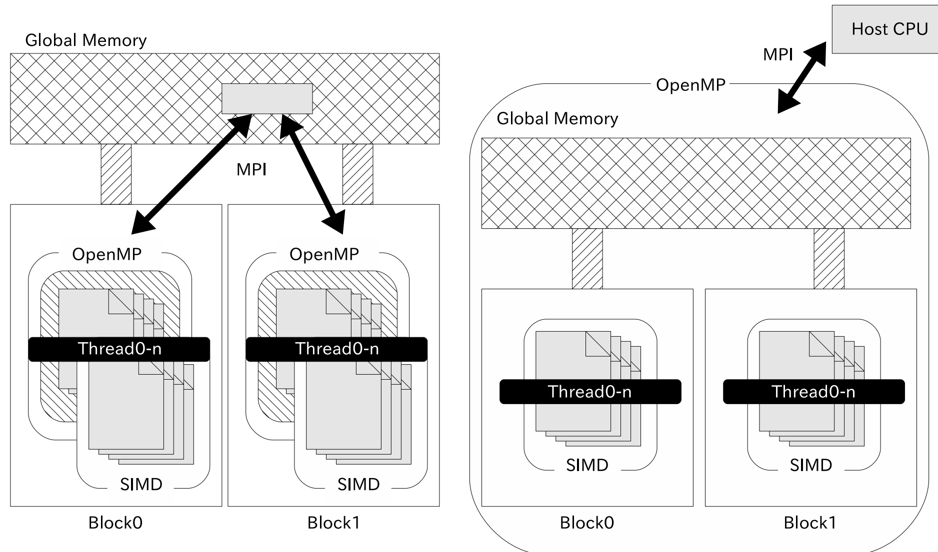

図 8 CUDA を用いたハイブリッド並列プログラムの動作イメージ
Fig. 8 Executive image of hybrid parallel program using CUDA

そのため CUDA 向け OpenMP の実装については，Thread レベルと Block レベルのそれぞれで実装して性能比較を行い，より良い実装を探ることにする．

4.3 CUDA 向け MPI の実装

最後に CUDA 向け MPI の実装について考える．CUDA の並列実行モデルのうち，Thread レベルの並列処理は SIMD に近く，異種の演算を並列実行することができない．これは疎粒度の並列処理を行う MPI において致命的な問題であるため，Block レベルの並列処理を利用することにする．MPI を用いた並列処理にはノード間の通信が不可欠であり，また同期通信関数を模倣するには Block 間の同期機構が必要である．CUDA には Block 間で直接的な通信を行う機構が存在しないが，Global Memory を介してデータをやり取りすることで Block 間の通信を模倣する．更に CUDA には Block 間の同期を取る機構が用意されていないため，Global Memory を利用して同期処理を模倣する．以上によって CUDA 向け MPI が実装できると考えられる．

一方，CPU-GPU 間のデータ転送を行う関数を定義し，CPU と GPU という異なるハードウェア間の分散メモリを MPI のインターフェイスで管理可能にするという適用方法も考えられる．この際の問題としては，CUDA の実行モデルとして Grid(CPU から GPU に対して演算指示を行う際の単位) の途中で CPU-GPU 間のデータ送受信ができないこと，そのため Grid の分割などが必要となる可能性が挙げられる．しかしながら既存のインターフェイスで管理できるメリットは大きいため，この適用方法は実装する価値がある．

以上のように Thread レベルの並列化と Block レベルの並列化を適切に使い分けることで，CUDA 向けの SIMD 命令，OpenMP，MPI を実装することができると考えられる．更に，既存の CPU 向け並列プログラミングにおける階層型の (ハイブリッドな) 並列プログラムを模倣することも可能となる．図 8 に我々が検討を進めている CUDA を用いたハイブリッド並列プログラムの動作イメージを示す．これらを用いることで，GPU の性能をより容易にかつ効率良く発揮できるようになることが期待できる．

5. 関 連 研 究

現在，効率良く GPGPU プログラミングを行うための言語やライブラリの研究がいくつか行われている．中には BrookGPU[12] や RapidMind[13] のように，GPU のみではなく Cell Broadband Engine やマルチコア CPU などへの適用を視野に入れたものも存在する．これらは GPGPU プログラミングを容易にしたいという要求は本研究と同様である．しかしながら，C/C++に対する拡張という形をとり既存のプログラミング言語に近い記述ができるようにしている

とはいえ，新しい言語やライブラリを作成しているため既存のアプリケーションプログラマに対する習得コストの大きさは否めない．一方，対象アーキテクチャに適した言語を作成することで既存の言語を用いるより高い性能を得やすい可能性もあるため，記述や習得のしやすさと性能についての議論・評価を行う価値がある．

本研究で実装に利用している CUDA は，GeForce8000 シリーズのアーキテクチャに強く依存したライブラリである．CUDA は今後 NVIDIA のリリースする新しい GPU でも利用できるとされているが，今のところ他社の GPU で用いることはできないため，GPU プログラムをすべて CUDA で記述するというのは現実的ではない．これに対し，NVIDIA と GPU のシェアを二分している AMD も Close-to-the-Metal(CTM)[14] の提供を表明している．GPU プログラムの記述しやすさや性能の発揮しやすさの面から，今後も GPU ベンダーが GPGPU 向けに開発環境や資料を提供し続ける可能性は高い．一方で，CUDA や CTM が言語仕様の変更なく継続して提供されるかは不明であり，また GPU の種類ごとに個別のプログラムを作成する必要があることは，GPGPU プログラミングを容易にするという観点からは大きな障害である．本提案は GPU の違いを吸収し共通に利用できる環境を提供するものであり，こうした問題を解決できる可能性があると言える．

6. おわりに

本稿では GPGPU を容易に利用するための新しい手法として，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た．また CUDA を用いた実装例についても検討を行った．本提案によって GPGPU プログラミングに既存の並列化手法が利用できるようになれば，様々なアプリケーションに対して容易に GPU を活用できるようになる．更に GPU という特殊なハードウェア向けのプログラミングを既存の CPU 向けプログラミング手法で行う例として考えることで，並列化手法にとどまらず様々な CPU 向けのプログラミング手法を GPU 向けに活用できる可能性を示している．本提案は，今後ますますの性能向上が期待される GPU を容易に活用する新しい可能性を示すものである．

現在は提案内容の実装を進めている．今後も実装を進め，各種アプリケーションに適用して性能評価を行う．またこれらを通して GPGPU プログラミングに既存の並列化手法を用いること自体についても評価を行うことで，より良い GPGPU プログラミング手法についての議論が深まることが期待できる．

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) Apple: Apple Human Interface Guidelines, http://developer.apple.com/documentation/UserExperience/Conceptual/OSXHIGuidelines/.
5) Apple: Graphics & Imaging - Quartz, http://developer.apple.com/graphicsimaging/quartz/.
6) Microsoft: Experience Windows Vista: Windows Aero, http://www.microsoft.com/windows/products/windowsvista/features/experiences/aero.mspx.
7) RenderingProject/aiglx: RenderingProject/aiglx - Fredora Project Wiki, http://fedoraproject.org/wiki/RenderingProject/aiglx.
8) 中西悠, 渡邊啓正, 平澤将一, 本多弘樹: コードの性能可搬性を提供する SIMD 向け共通記述方式, 情報処理学会論文誌 コンピューティングシステム, Vol. 48, pp. 95–105 (2007).
9) M.Sato, S.Satoh, K.Kusano and Y.Tanaka: Design of OpenMP Compiler for an SMP Cluster, *EWOMP '99*, pp. 32–39 (1999).
10) Burns, G., Daoud, R. and Vaigl, J.: LAM: An Open Cluster Environment for MPI, *Proceedings of Supercomputing Symposium*, pp. 379–386 (1994).
11) Squyres, J. M. and Lumsdaine, A.: A Component Architecture for LAM/MPI, *Proceedings, 10th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, No. 2840, Venice, Italy, Springer-Verlag, pp. 379–387 (2003).
12) BrookGPU: BrookGPU, http://graphics.stanford.edu/projects/brookgpu/.
13) RapidMind Inc.: RapidMind, http://www.rapidmind.net/.
14) ATI: ATI CTM Guide, http://ati.amd.com/companyinfo/researcher/documents.html.