GPGPUフレームワークMESI-CUDAにおける マルチGPUへのスレッドマッピング機構

丸山 剛寛[1]　田中 宏明[1]　水谷 洋輔[1,†1]　神谷 智晴[1]　大野 和彦[1]

概要：近年，GPU上で汎用計算を実行するGPGPUが注目されている．現在主流な開発環境であるCUDAでは，高級言語で記述することが可能だが，GPUの複雑なメモリ構造を意識してプログラミングする必要がある．これに対し，我々は単純なメモリ構造モデルでプログラミング可能なMESI-CUDAを提案している．しかし，現在のMESI-CUDAは単一のGPU環境を想定しておりGPU1基を使用するコードしか生成できないため，複数のGPUを搭載していても性能を発揮できない．一方，CUDAでは複数のGPUを利用できるが，個々のGPUをユーザーが直接制御する必要がある．そこで，我々は複数のGPUへ自動的に処理を振り分けるスレッドマッピング機構を提案する．処理を振り分ける方法として静的／動的な方法が考えられるが，前者は静的解析のみで最適な振り分けをすることが困難である．一方，後者の場合，負荷の均等化のためには振り分けの粒度を細かくすることが望ましいが，その結果として並列度が下がりGPUの利用効率が低下する可能性がある．そこで本稿では，静的解析の結果を用いてある程度の処理を静的に振り分けた後，動的振り分けに切り替えるハイブリッド型の手法を提案する．

キーワード：並列コンピューティング，GPGPU，CUDA

Thread Mapping Scheme for Multi GPU on Framework MESI-CUDA

Takanori Maruyama[1]　Hiroaki Tanaka[1]　Yousuke Mizutani[1,†1]　Tomoharu Kamiya[1]
Kazuhiko Ohno[1]

Abstract: The performance of Graphics Processing Units (GPU) is improving rapidly. Thus, General Purpose computation on Graphics Processing Units (GPGPU) is expected as an important method for high-performance computing. CUDA, one of the major developing environment, enables GPU programming using C/C++, but the user must handle the complicated memory architecture. Therefore, we are developing a new programming framework named *MESI-CUDA*, which provides a simple memory architecture model automatically generating low-level CUDA code. However, the current implementation of MESI-CUDA only supports single GPU and cannot utilize multiple GPUs. Although CUDA supports multi-GPU, the user must directly control each device. Thus, we propose a scheme which automatically maps GPU threads to multiple GPUs. Although static mapping causes small runtime overhead, the result may not be optimal. On the other hand, the efficiency of dynamic mapping is largely influenced by the granularity of thread grouping; large grouping disturbs optimal load-balancing, while small grouping cannot utilize large number of GPU cores. Therefore, we propose a hybrid scheme combining static and dynamic mapping methods.

Keywords: Parallel Computing，GPGPU，CUDA

1. はじめに

近年，GPUはCPUに比べて性能向上がめざましく，ムーアの法則をしのぐ演算性能の向上を見せている．そ

[1] 三重大学大学院 工学研究科
Mie University
[†1] 現在，朋和産業株式会社
Presently with HOWA SANGYO CO.,LTD.

© 2014 Information Processing Society of Japan

の演算性能に注目して，GPU に汎用的な計算を行わせる GPGPU（General Purpose computation on Graphics Processing Units）[1] への関心が高まっている．また，CUDA[2] や OpenCL [3] といった GPGPU プログラミング開発環境が提供されている．しかし，これらの開発環境は GPU アーキテクチャに合わせた低レベルなコーディングを必要とする．そのため，ユーザは GPU のアーキテクチャを意識しなければならずプログラミングは困難である．特に，メモリがホスト側（CPU）とデバイス側（GPU）に分かれており，プログラマは両メモリ間のデータ転送コードを記述する必要がある．さらに，デバイス側が複雑なメモリ階層を持ち，用途に応じて使い分けなければ性能を発揮できない．そこで我々はデータ転送を自動化するフレームワーク MESI-CUDA（Mie Experimental Shared-memory Interface for CUDA）[4][5][6] を開発している．本フレームワークは共有メモリ型の GPGPU プログラミングのモデルを提供する．そのため，自動的にホストメモリ・デバイスメモリ間のデータ転送コードを生成する．また，デバイスに応じた最適化を自動的に行う．これによりデバイスに依存しないプログラムを容易に作成することが可能となる．さらに，データ転送と GPU 上での計算のオーバーラップを行うことでプログラムの実行性能も向上させる．

しかし，現状の MESI-CUDA はシングル GPU 環境を想定しており，GPU1 基のみを使用するコードしか生成できない．そのため，複数の GPU を搭載している場合，性能を完全に発揮できない．そこで本研究では，MESI-CUDA 上に，複数の GPU へ自動的に処理を振り分けるスレッドマッピング機構を提案する．

以下，2 章では背景として GPU アーキテクチャと CUDA について解説する．3 章では関連研究を紹介し，4 章で MESI-CUDA の機能とプログラミングモデルについて説明する．5 章ではプログラム解析やコード生成などのスレッドマッピング機構の手法を示す．6 章で，スレッドマッピング機構の有無による CUDA プログラムの実行時間を比較し，その評価結果を示す．最後に，7 章でまとめを行う．

2. 背景

2.1 GPU アーキテクチャ

図 1 に GPU のアーキテクチャモデルを示す．GPU の基本的なアーキテクチャは，多数のコアがグローバルメモリを共有している構造である．しかし，メモリは複雑に階層化されており，それぞれの用途ごとに使い分ける必要がある．各コアはレジスタやローカルメモリを持っている．また，コアは一定数毎にストリーミングマルチプロセッサ（以降 SM と記述）を形成しており，各 SM 毎にシェアードメモリを持つ．

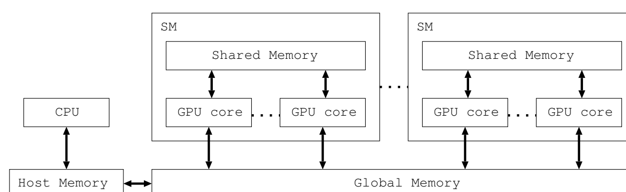

図 1 GPU のアーキテクチャモデル
Fig. 1 GPU Architecture Model

2.2 CUDA

CUDA は nVIDIA 社より提供されている GPGPU 用の SDK であり，C 言語を拡張した文法とライブラリ関数を用いて GPU プログラムを開発することができる．CUDA では，CPU をホスト，GPU をデバイスと呼ぶ．CUDA を用いた行列積を求めるプログラムを図 2 に示す．

カーネル

デバイス上で実行される関数はカーネル関数と呼ばれ，その関数には修飾子`__device__`か`__global__`が付与される（図 2：4 行）．修飾子のついていない関数や`__host__`の修飾子のついた関数はホスト側で実行される．ホスト側のコードから`__global__`の修飾子のついた関数を呼び出すことで，デバイス上でカーネル関数を実行することができる（図 2：29 行）．このときに作成するスレッド数を指定する．

データ転送

CUDA におけるデータ転送は関数の呼び出しで行う．データ転送の種類は 2 種類あり，ホストからデバイスへのデータ転送をする download 転送（図 2：27-28 行）と，デバイスからホストへのデータ転送をする readback 転送（図 2：30 行）である．

グリッド・ブロック

CUDA の仕様では，最高で 2147483647 × 65535 × 65535 × 1024 個のスレッドを実行できる．しかし，このような多数のスレッドに対して 1 つの整理番号で管理するのは困難である．そのため，CUDA ではグリッドとブロックという概念を導入し，その中で階層的にスレッドを管理している．グリッドは 1 つだけ存在し，グリッドの中はブロックで構成されている．ブロックは x 方向，y 方向，z 方向の 3 次元で構成されている．スレッドはブロック内で同様に 3 次元的に管理されている（図 3）．また，同一ブロック内のスレッドは同一 SM 内のコアで実行される．

ビルトイン変数

CUDA にはビルトイン変数が存在し，宣言なしにカーネル関数内で使用できる．各ブロック・スレッドにはそれぞれ番号が割り振られており，`gridDim.x` でブロックの個数を，`blockIdx.x` でブロック番号（0–gridDim.x-1）を，`blockDim.x` でスレッドの個数を，`threadIdx.x` でスレッド番号（0–`blockDim.x-1`）をそれぞれ得ることができる．上で示した変数では x 方向についての値を得ているが，`.x` の部分を`.y`，`.z` とすることでそれぞれ y 方向と z 方向の

ⓒ 2014 Information Processing Society of Japan

```
 1 #define N 4096
 2 #define TPB 1024
 3 #define S (N*N*sizeof(int))
 4 __global__
   void transpose(int *a, int *b, int *c){
 5   int id = blockIdx.x*blockDim.x+threadIdx.x;
 6   int iy = id/N;
 7   int ix = id%N;
 8   int scan;
 9   c[id] = 0;
10   for(scan=0;scan<N;scan++){
11    c[id] += a[N*iy+scan] * b[N*scan+ix];
12   }
13 }
14 void init_array(int d[N][N]){...}
15 void output_array(int d[N][N]){...}
16 int main(int argc, char *argv[]){
17   int *host_a, *host_b, *host_c;
18   int *dev_a, *dev_b, *dev_c;
19   cudaMallocHost(&host_a, S);
20   cudaMallocHost(&host_b, S);
21   cudaMallocHost(&host_c, S);
22   cudaMalloc(&dev_a, S);
23   cudaMalloc(&dev_b, S);
24   cudaMalloc(&dev_c, S);
25   init_array((int(*)[N])host_a);
26   init_array((int(*)[N])host_b);
27   cudaMemcpy(dev_a, host_a, S, cudaMemcpyHostToDevice);
28   cudaMemcpy(dev_b, host_b, S, cudaMemcpyHostToDevice);
29   transpose<<<N*N/TPB,TPB>>>(dev_a, dev_b, dev_c );
30   cudaMemcpy(host_c, dev_c, S, cudaMemcpyDeviceToHost);
31   output_array((int(*)[N])host_c);
32   cudaFree(dev_a);
33   cudaFree(dev_b);
34   cudaFree(dev_c);
35   cudaFreeHost(host_a);
36   cudaFreeHost(host_b);
37   cudaFreeHost(host_c);
38 }
```

図 2　行列積を求める CUDA コード
Fig. 2　Matrix Multiplication Program using CUDA

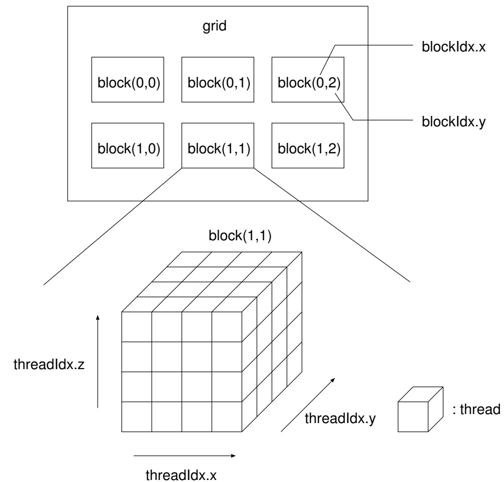

図 3　グリッド-ブロック-スレッド
Fig. 3　grid-block-thread

値を得ることができる．ブロックの番号はユニークであるがスレッド番号はブロックごとに割り振られているため，カーネル関数を起動したとき全スレッドで見るとブロックの数だけスレッド番号が重複してしまう．式 `blockDim.x` × `blockIdx.x` ＋ `theadIdx.x` の値は各スレッドごとにユニークであり，0 から始まる連続した値となる．よってここではこの式の値をスレッドの ID として用いることとし，以下 id と記述する．

メモリ確保・解放

デバイス上で使用する変数はホスト側で `cudaMalloc`，`cudaFree` 関数を用いてメモリ確保・解放を行う必要がある（図 2：22-24，32-34 行）．

2.3　マルチ GPU

GPU1 基を搭載して動作させるシングル GPU に対し，GPU を複数搭載して並列動作させるマルチ GPU は，各 GPU に処理を割り振ることで CUDA プログラムの実行時間短縮が可能となる．CUDA でマルチ GPU を利用する場合は，明示的に `cudaSetDevice` 関数を用いて使用する GPU を指定してから処理を行わせる必要がある．同一の GPU を用いたマルチ GPU 環境ではそれぞれの GPU が全く同一の性能であるため，処理は均等に振り分ければよい．しかし，性能の異なる GPU を用いるときは，均等な処理の振り分けでは性能の劣る GPU の処理がボトルネックとなり，性能の優れた GPU 単体で処理させた場合よりも劣る場合がある．そのため，性能に応じた処理の振り分けを行う必要がある．

3.　関連研究

ユーザに GPU プログラミングを意識させないものとして openACC[10] が挙げられる．これは CUDA のような GPU プログラム用の独自言語を使用せず，並列化を行い

たい逐次処理プログラムに簡単な指示文を挿入することでGPU プログラミングを可能としている．並列化が可能な構文に合わせた指示文を指定することで自動的に GPU で計算できるようコードを変換している．そのため，ユーザは CUDA などの言語を覚える必要は無く，低レベルな最適化コードの記述方法を学ぶ必要もない．一方，すべてコンパイラに任せることになるためユーザが低レベルな並列化処理を記述して最適化したコードと比べると計算速度は劣る．MESI-CUDA フレームワークは，記述の容易さでは openACC に劣るものの，並列処理部分をユーザが記述するため高速なコードを生成しやすい．

マルチ GPU を使用する方法としては，OpenMP や MPI と OpenACC を組み合わせて用いる方法 [11] がある．これは OpenMP や MPI を使って複数のスレッドに処理を分散し、各スレッドでは OpenACC で 1 個の GPU を使うものである．また，OpenACC だけを用いる方法もある．これは使用する GPU を切り替えてデータ転送や演算処理を非同期的に実行することで複数の GPU で並列に処理をさせるものである．どちらの方法においても OpenACC を用いるため，GPU の処理は容易に記述することができるが，スレッド管理や非同期処理はユーザー自身が行わなければならないためプログラミングの難易度は高い．

4. MESI-CUDA の機能

4.1 MESI-CUDA 概要

MESI-CUDA フレームワークは，データ転送コードやメモリ確保・解放，ストリーム処理のコードを自動的に生成することで，ユーザの負担を軽減させる．ホストとデバイスへの処理の振り分けやカーネルの記述はユーザ自身が従来の CUDA に準じる形でコーディングを行う．図 4 に図 2 の CUDA プログラムと等価な MESI-CUDA プログラムを示す．

MESI-CUDA では，データ転送やカーネル処理のスケジューリングを自動的に行う．そのため，仮想的な共有メモリ環境のモデルを採用し，ホスト・デバイス両方よりアクセス可能な共有変数を提供する．共有変数の宣言方法は，図 4 : 3 行目のように変数宣言の修飾子として，`__global__` を付与する．CUDA では図 1 の GPU アーキテクチャをそのままプログラミングモデルとして用いる．これに対し，MESI-CUDA では図 5 に示すプログラミングモデルを用いている．CUDA ではホストメモリ・デバイスメモリを意識してプログラミングする必要があったが，MESI-CUDA では 1 つの共有メモリに見せかけている．よって，ホスト関数・カーネル関数の違いによる変数の使い分けや，データ転送の記述が不要になる．また，フレームワークで自動的に転送のタイミングやカーネル処理の順序を決定し，最適化を行う．この処理の中で，カーネル処理とデータ転送とのオーバーラップが可能なようにストリームの割り当てを行う．

```
 1 #define N 4096
 2 #define TPB 1024
 3 __global__ int a[N*N], b[N*N], c[N*N];
 4 __global__
   void transpose(int *a, int *b, int *c){
 5   int id = blockIdx.x*blockDim.x+threadIdx.x;
 6   int iy = id/N;
 7   int ix = id%N;
 8   int scan;
 9   c[id] = 0;
10   for(scan=0;scan<N;scan++){
11     c[id] += a[N*iy+scan] * b[N*scan+ix];
12   }
13 }
14 void init_array(int d[N][N]){...}
15 void output_array(int d[N][N]){...}
16 int main(int argc, char *argv[]){
17   init_array(a);
18   init_array(b);
19   transpose<<<N*N/TPB,TPB>>>(a, b, c);
20   output_array(c);
21 }
```

図 4　CUDA コードと等価な MESI-CUDA コード
Fig. 4　Equivalent Program using MESI-CUDA

図 4 から分かるようにカーネル関数に関する記述や，ホスト側での処理は CUDA と同様に行っている．その一方で共有変数を用いることにより，メモリ確保・解放，データ転送，ストリームの生成・破棄・指定が不要になっている．

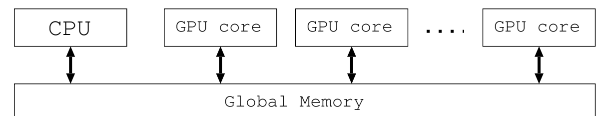

図 5　MESI-CUDA のプログラミングモデル
Fig. 5　Programming Model for MESI-CUDA

4.1.1 本プログラミングモデルの利点・欠点

前述のようにデータ転送やストリーム処理などの記述が不要であり，簡潔なコーディングが可能である．C 言語に比べて大きく異なる点は，カーネル関数の記述のみで，カーネル関数の記述を特殊な関数と見なせば C 言語ライクなコーディングが可能である．しかし，低レベルな記述をフレームワークで隠蔽しているため，メモリ階層の有効活用をユーザが行うことはできず実行性能が処理系の最適化能力に大きく依存する．

4.1.2 現在の処理系の問題点

MESI-CUDA はシングル GPU 環境を想定しており，GPU1 基のみを使用するコードしか生成できない．そのため，計算機に複数の GPU を搭載している場合，性能を完全に発揮できない．

ⓒ 2014 Information Processing Society of Japan

5. スレッドマッピング機構

5.1 概要

前述したように MESI-CUDA はマルチ GPU に対応していない．そこで，我々は複数のデバイスへ自動的に処理を振り分けるスレッドマッピング機構を提案する．複数のデバイスを使用することによって，CUDA プログラムの高速化が可能となる．また，マルチ GPU を使用する煩雑なコードが不要なため，ユーザーはシングル GPU と同様にプログラムを記述することができる．

CUDA および MESI-CUDA ではブロック間の排他制御を行う手段が無いため，各ブロックの実行は独立している．そのため，基本的に各ブロックの実行は任意のデバイス上で可能であると考えられる．したがって、ブロックを処理の割り振りの最小単位とする．マルチ GPU を利用するにはどのデバイスで実行するかを決めたり，割り振りに対応するデータの転送を行ったりする必要がある．MESI-CUDA ではメモリ転送などのコードを自動生成しており，上記のようなコードを生成するように拡張することで実現する．

5.1.1 問題点

性能の異なるデバイスに対し均等に処理を振り分けてしまうと、性能の劣るデバイスがボトルネックとなり、かえって性能が低下する可能性がある．そのため各デバイスの性能に見合った処理の割り振りを行う必要がある．

処理を割り振る方法として、静的・動的の 2 通りが挙げられる．静的な方法では、コンパイル時にプログラムを解析し，割り振る量を事前に決めておく．これは実行時に余分な動作をしないためオーバーヘッドが少ない．しかし，完全にプログラムを解析することは不可能であるため，最適に割り振ることは困難である．

動的な方法では、デバイスの状況に応じて処理を割り振る．これは処理を細分化させ，処理の終わったデバイスへ順番に割り振るため，最適に近い割り振りをすることができる．しかし，実行時にデバイスの状態を監視しなければならないためオーバーヘッドが生じる．また，負荷の均等化のためには振り分けの粒度を細かくすることが望ましいが，その結果として一回のカーネル実行あたりの並列度が下がりデバイスの利用効率が低下する可能性がある．

5.1.2 提案手法

そこで，静的解析の結果を用いてある程度の処理を静的に振り分けた後，動的振り分けに切り替えるハイブリッド型の手法を提案する．本機構では，ユーザーの書いたプログラムを静的解析し，各デバイスの性能により静的に割り振る処理の比率を決定し，その比率に基づいて各デバイスに処理を行わせる CUDA コードを自動生成する．また，一定量の処理を静的に割り振った後に残りを細分化し，各デバイスの状態を監視しながら動的に割り振るコードを自動生成する．これにより，並列度を維持しつつ負荷の均等化を実現し，実行時間の短縮ができる．

前提条件として今回対象とするプログラムのメモリ使用量はシングル GPU でも十分動作可能なものとし，複数のデバイスで実行しなければならないような巨大なデータを扱うプログラムは今後の課題とする．また，コンパイル時に実行環境が分かっているものとする．これにより，実行環境が異なると再コンパイルしなければならないが，実行環境に応じた最適化が可能である．また，今回行う解析は配列を対象とし，アクセス範囲を求める．

5.1.3 実現方法

シングル GPU の場合，処理に必要なデータをデバイスへ一度に転送し，スレッド数を指定してカーネルを呼び出し，処理の結果をホストへ一度に転送するだけで良い．これに対しマルチ GPU の場合，各デバイスで並列に処理を行わせるために，それぞれに必要なデータを転送し，スレッド数を指定しカーネルを呼び出し，ホストへそれぞれの処理の結果を転送する必要がある．そこで，各デバイス毎にデータ転送とカーネル起動を行うようなコードを生成する．しかし，処理をブロック単位で分けた場合，カーネル実行に全てのデータが必要とは限らず，シングル GPU と同じものを転送するとメモリや転送時間が無駄になってしまう．そこで，各デバイスが必要なデータだけを転送するために，カーネル関数内で単一のブロックがアクセスする配列の範囲を解析する必要がある．また，静的振り分けを行う際，各デバイスの性能に合わせて処理を振り分けるために，デバイスの性能を評価する必要がある．そして，動的振り分けを行う際，振り分ける粒度を調整するために，計算量を解析する必要がある．

5.2 解析

5.2.1 範囲解析

各デバイスに転送する配列の要素数を決定するために，カーネル関数内で単一のブロックがアクセスする配列の範囲を解析する．ループ文中で，アクセスされる各配列のインデックスの式中のループ変数や `threadIdx.x` を最小値から最大値まで変化させることで求める．

5.2.2 GPU の計算性能

静的に各デバイスへ処理を割り振る場合，その比率を決定するために，各デバイスの性能を評価する．コンパイル実行時の環境，あるいはコンパイル時に指定されたデバイスについて評価を行う．現在はデバイスの総コア数 × パイプライン数 × クロックレートで推定値を求めている．しかし，メモリの性能や GPU の世代差などを考慮していないので，今後改善する必要がある．

5.2.3 計算量解析

動的に各デバイスへ処理を割り振る際の粒度を決定するために，カーネル関数の計算量を解析する．計算量をソースコードから完全に解析するのは困難であるため，近似値

を求める．現在はループ回数 × ループ文中の演算数で推定値を求めている．今後，演算の種類によって重み付けするなどしてより正確に求めるように改善する必要がある．

5.3 コード生成の方針

5.3.1 データ転送

解析して得たアクセス範囲を用いて転送する配列の範囲を決定する．静的割り振りの場合は，デバイスの性能解析の結果を用いて各デバイスの処理量を決定するので，それに必要なデータを転送する．動的割り振りの場合は，処理を細分化して実行するので，一回の処理に必要なデータを転送する．

5.3.2 ブロック割り振り

元のプログラムでは 1 つのデバイスで生成し実行させていたスレッドを，複数のデバイスに分けて実行させるようにする．静的割り振りの場合は，デバイスの性能評価の結果を用いて各デバイスの処理量を決定し，各デバイス毎に 1 回ずつカーネルを起動する．動的割り振りの場合は，処理を細分化して実行するので，各デバイス毎に複数回に分けてカーネルを起動する．しかし，あまりに細かく起動していては，デバイスの性能を発揮できないという問題がある．これは CUDA のブロックが GPU の SM と対応しており，ブロック数が SM 数より少ない場合は全ての SM に処理を行わせることができないためである．また，SM あたりのブロック数が少ないと SM 上で複数ブロックの並行実行が行われずメモリアクセスのレイテンシが隠蔽できない，最後に端数のブロックを実行する際に他の SM がアイドルになる，といった問題もある．そのため，カーネルの起動で一定数以上のブロックが生成されるように細分化の粒度を調整する．

5.4 コード生成

元のプログラムの総ブロック数を $OBlock$，静的／動的割り振りの比率をそれぞれ $SRate$, $DRate$ ($SRate+DRate=1$)，配列 a の 1 ブロックあたりのアクセス要素数を $AcsElem(a)$，n 番目のデバイスの性能比率を $GPURate(n)$，動的分割数を Div とする．デバイスの性能比率は総性能に対しそのデバイスの性能が占める比率になるように正規化されているものする．

静的に割り振るブロック数は $OBlock \times SRate$ となるので，n 番目のデバイスが実行するブロック数は $OBlock \times SRate \times GPURate(n)$ となり，以下 $SBlock(n)$ と記述する．動的に割り振るブロック数は $OBlock \times DRate$ となるので，1 回で起動するブロック数は $OBlock \times DRate / Div$ となり，以下 $DBlock$ と記述する．

図 4 のプログラムを今回の提案手法でマルチ GPU 化したコードの一部を図 6〜図 8 に示す．以下，これらを使ってコード生成の方法について説明する．

5.4.1 ストリーム生成／メモリ確保

動的振り分けのために，各デバイス毎にストリームを生成する（図 6 : 1-7 行）．これは，動的振り分け時に `cudaStreamQuery` 関数を用いてデバイスの状態を確認し，動的に処理やデータ転送を行わせるためである．また，静的割り振りと異なり同一デバイスに複数回処理を行わせるので，データ転送とカーネル実行をオーバーラップさせ転送時間の隠蔽を行うためでもある．

ホスト用の変数は `cudaHostAlloc` でメモリ確保を行う（図 6 : 8-10 行）．これはストリームを用いて非同期でデータ転送を並列実行するためである．

デバイス用の変数は各デバイス毎にメモリ確保を行う（図 6 : 11-18 行）．カーネル内で 1 つのブロックが全ての要素をアクセスする配列は，全ての要素分確保する（図 6 : 15 行）．カーネル内で 1 つのブロックが全ての要素をアクセスしない場合は，いくつかのパターンが考えられる．ブロックの必要範囲が一定かつ排他的であった場合，静的割り振り用と動的割り振り用の 2 種類の変数を用意する．静的割り振り用は各デバイスに振り分ける処理で必要な分を確保する（図 6 : 13,16 行）．動的割り振り用は一回のカーネル実行に必要な要素数 × ストリーム数分を確保する（図 6 : 14,17 行）．他にも，ブロックの必要範囲が重複している場合やアクセスが規則的／不規則的，連続／不連続な場合が考えられる．このような場合は，各デバイスで全ての要素分確保し転送するようにする．ただし，デバイスで書き込みを行いホストへ転送される配列は，結果を正しくマージする必要がある．重複／不規則なアクセスには今回は対応していない．しかし，このようなアクセスは GPU プログラムには不向きであるため，該当するものは少ないと思われる．

5.4.2 静的振り分け

`cudaSetDevice` を使いデバイスを切り替えながら，データ転送を行う（図 7 : 3 行）．カーネル内で 1 つのブロックが全ての要素をアクセスする配列は，そのまま全て転送する（図 7 : 5 行）．一部しかアクセスしない配列は，$SBlock(n) \times AcsElem(a)$ 個の要素を転送する（図 7 : 4 行）．最初のデバイスには 0 番目の要素から転送し，次のデバイスにはその続きから転送する（図 7 : 1,10 行）．これを搭載されているデバイスの個数だけ繰り返し，全てのデバイスにデータを転送する（図 7 : 2 行）．ブロック数 $SBlock(n)$ でカーネル実行を行う（図 7 : 6 行）．

5.4.3 動的振り分け

静的に割り振った処理が終了したデバイスから動的に処理の振り分けを行う（図 8 : 6 行）．`cudaStreamQuery` でストリームの状態を確認し実行可能状態 (`cudaSuccess`) だったら，データ転送を行う（図 8 : 7 行）．カーネル内で 1 つのブロックが全ての要素をアクセスする配列は，静的割り振り時に既に転送してあるので転送しない．一部しかアク

ⓒ 2014 Information Processing Society of Japan

```
1   cudaStream_t _st[_deviceCount][_BUF];
2   for(_i=0;_i<_deviceCount;_i++){
3     cudaSetDevice(_i);
4     for(_j=0;_j<_BUF;_j++){
5       cudaStreamCreate(&_st[_i][_j]);
6     }
7   }
8   cudaHostAlloc((void**)&_h_a,S,cudaHostAllocDefault);
9   cudaHostAlloc((void**)&_h_b,S,cudaHostAllocDefault);
10  cudaHostAlloc((void**)&_h_c,S,cudaHostAllocDefault);
11  for(_i=0;_i<_deviceCount;_i++){
12    cudaSetDevice(_i);
13    cudaMalloc((void**)&_d_as[_i],
        _SBlock(_i)*_AcsElem(a)*sizeof(int));
14    cudaMalloc((void**)&_d_ad[_i],
        _DBlock*_AcsElem(a)*_BUF*sizeof(int));
15    cudaMalloc((void**)&_d_b[_i],S;
16    cudaMalloc((void**)&_d_cs[_i]
        _SBlock(_i)*_AcsElem(c)*sizeof(int));
17    cudaMalloc((void**)&_d_cd[_i],
        _DBlock*_AcsElem(c)*_BUF*sizeof(int));
18  }
```

図 6 図 4 のマルチ GPU 化コード (初期化)

Fig. 6 Applying Proposed Scheme (Initialize)

```
1   _a_n=0,_c_n=0;
2   for(_i=0;_i<_deviceCount;_i++){
3     cudaSetDevice(_i);
4     cudaMemcpyAsync(_d_as[_i],&_h_a[_a_n],
        _SBlock(_i)*_AcsElem(a)*sizeof(int),
        cudaMemcpyHostToDevice, 0);
5     cudaMemcpyAsync(_d_b[_i],_h_b,S,
        cudaMemcpyHostToDevice, 0);
6     transpose<<<_SBlock(_i),TPB,0,0>>>
        (_d_as[_i],_d_b[_i],_d_cs[_i]);
8     cudaMemcpyAsync(&_h_c[_c_n],_d_cs[_i],
        _SBlock(_i)*_AcsElem(c)*sizeof(int),
9       cudaMemcpyDeviceToHost, 0);
10    _a_n+=_SBlock(_i)*_AcsElem(a);
11    _c_n+=_SBlock(_i)*_AcsElem(c);
12  }
```

図 7 図 4 のマルチ GPU 化コード (静的振り分け)

Fig. 7 Applying Proposed Scheme (Static Scheduling)

セスしない配列は，*DBlock*×*AcsElem(a)* 個の要素を転送する（図 8：8,10 行）．最初は静的に割り振った続きの要素から転送し，以降はその続きから転送する（図 8：12 行）．これを *Div* 回繰り返す（図 8：3 行）．また，1 回のループ毎に異なるストリームを用い，データ転送とカーネルをオーバーラップさせる．ブロック数 *DBlock* でカーネル実行を行う（図 8：9 行）．またその際に引数のインデックスを調整する．

```
1   _step = 0;
2   for(_i=0;_i<_deviceCount;_i++) _d_step[_i]=0;
3   while(_step<_Div){
4     for(_i=0;_i<_deviceCount;_i++){
5       cudaSetDevice(_i);
6       if(cudaStreamQuery(0) == cudaSuccess){
7         if(cudaStreamQuery(_st[_i][_d_step[_i]%_BUF]){
            ==cudaSuccess){
8           cudaMemcpyAsync(
              &_d_ad[_i][_d_step[_i]%_BUF*_DBlock*_AcsElem(a)],
              &_h_a[_step*_DBlock*_AcsElem(a)+_a_n],
              _DBlock*_AcsElem(a)*sizeof(int),
              cudaMemcpyHostToDevice,_st[_i][_d_step[_i]%_BUF]);
9           transpose<<<_DBlock,TPB,0,
              _st[_i][_d_step[_i]%_BUF]>>>(
              &_d_ad[_i][_d_step[_i]%_BUF*_DBlock*_AcsElem(a)],
              _d_b[_i],
              &_d_cd[_i][_d_step[_i]%_BUF*_DBlock*_AcsElem(c)]);
10          cudaMemcpyAsync(
              &_h_c[_step*_DBlock*_AcsElem(c)+_c_n],
             &_d_cd[_i][_d_step[_i]%_BUF*_DBlock*_AcsElem(c)],
             _DBlock*_AcsElem(c)*sizeof(int),
11            cudaMemcpyDeviceToHost,_st[_i][_d_step[_i]%_BUF]);
12          _step++;
13          _d_step[_i]++;
14          if(_step==_Div) break;
15        }
16      }
17    }
18  }
```

図 8 図 4 のマルチ GPU 化コード (動的振り分け)

Fig. 8 Applying Proposed Scheme (Dynamic Scheduling)

6. 評価

提案したスレッドマッピング機構の有用性を示すために，本機構を用いた最適化の有無による MESI-CUDA プログラムの実行時間の比較を行った．評価環境は Core i7-3820 3.60GHz，メモリ 16GB，GeForce GTX TITAN，GeForce GTX 680 を搭載した計算機を使用した．評価には行列積の計算と 2 次元画像の平滑化を行う 2 本のプログラムを用いた．データの大きさが 512×512，1024×1024，2048×2048 の場合において，シングル GPU(GTX 680，GTX TITAN)／マルチ GPU(動的手法，静的手法，ハイブリッド手法) での実行時間を測定した．動的手法／静的手法の分割数／割合は実験的に求めた最適パラメータを用いた．ハイブリッド手法も同様に最適化し，静的割り振りと動的割り振りの比率を 3:1 で固定してある．結果を表 1，表 2 にそれぞれ示す．

行列積ではハイブリッド手法が動的手法よりも速い．これは静的割り振りである程度処理を割り振るため，5.3.2 章で述べた動的手法の問題点を解消できたためであると考え

られる．平滑化では問題サイズが大きくなるとハイブリッド手法は動的手法に劣っている．これは，動的と静的の最適な割合がプログラムにより異なり，今回の 3:1 の比率が適切ではなかったためと考える．これについては自動で決定する機構が必要である．静的手法での割り振りのパラメータが適切でない場合も動的手法の併用で補正できる．総ブロック数の少ないプログラムの場合，動的手法では高い並列性の維持が困難であるが，ハイブリッド手法なら影響が少ない．

表 1 行列積の実行時間 (秒)
Table 1 Execution Time of Matrix Multiplication

	GTX 680	GTX TITAN	動的	静的	ハイブリッド
512	0.0420	0.0344	0.0260	0.0219	0.0223
1024	0.327	0.244	0.180	0.152	0.154
2048	2.59	1.82	1.20	1.12	1.11

表 2 平滑化の実行時間 (秒)
Table 2 Execution Time of Smoothing

	GTX 680	GTX TITAN	動的	静的	ハイブリッド
512	0.0664	0.0970	0.0374	0.0273	0.0348
1024	0.182	0.302	0.111	0.0993	0.113
2048	0.634	1.12	0.415	0.385	0.425

7. おわりに

本研究では複数の GPU へ自動的に処理を振り分けるスレッドマッピング機構を提案した．性能評価の結果，本機構を用いることで複数の GPU へハイブリッド型の手法で処理を振り分け，実行時間を短縮できることが示された．今後の課題として，今回は処理の振り分けの静的／動的の割合を固定しているが，自動的に最適な割合を求められるようにする必要がある．また，提案した解析手法は簡単なものであり、より精度の高い方法を導入する必要がある．

謝辞 本研究の一部は日本学術振興会科研費・基盤研究（C）（課題番号 24500060）による．

参考文献

[1] *GPGPU.org: General-Purpose computation on Graphics Processing Units*, 入手先 ⟨http://www.gpgpu.org/⟩, (2013.06.22).
[2] *NVIDIA Developer CUDA Zone*, 入手先 ⟨http://developer.nvidia.com/category/zone/cuda-zone⟩, (2013.04.27).
[3] *OpenCL - The open standard for parallel programming of heterogeneous systems*, 入手先 ⟨http://www.khronos.org/opencl/⟩, (2013.06.20).
[4] 道浦 悌，大野 和彦，佐々木 敬泰 and 近藤 利夫: *GPGPU* におけるデータ自動転送化コンパイラの提案, 先進的計算基盤システムシンポジウム SACSIS2011,221-222,(2011).
[5] 道浦 悌，大野 和彦，佐々木 敬泰 and 近藤 利夫: *GPGPU* におけるデータ転送自動化コンパイラの設計, 情報処理学会研究報告 2011-HPC-130(17),1-9,(2011).
[6] Kazuhiko Ohno，Dai Michiura，Masaki Matsumoto，Takahiro Sasaki and Toshio Kondo: *A GPGPU Programming Framework based on a Shared-Memory Model*,Parallel and Distributed Computing and Systems - 2011,(2011).
[7] 中村 晃一，林崎 弘成，稲葉 真理 and 平木 敬: *SIMD* 型計算機向けループ自動並列化手法, 情報処理学会研究報告 2010-HPC-126(10),1-8,(2010).
[8] Muthu Baskaran，J.Ramanujam and P.Sadayappan: *Automatic C-to-CUDA Code Generation for Affine Programs*,Springer Berlin / Heidelberg,(2010).
[9] Yi Yang，Ping Xiang，Jingfei Kong and Huiyang Zhou: *A GPGPU compiler for memory optimization and parallelism management*,SIGPLAN Not.,86-97,(2010). (2013.06.20).
[10] *OpenACC*, 入手先 ⟨http://www.openacc-standard.org/⟩,(2013.06.7).
[11] Michael Wolfe: *Scaling OpenACC Across Multiple GPUs*,GPU Technology Conference 2014,(2014).

ⓒ 2014 Information Processing Society of Japan