Vol.2014-HPC-143 No.6
2014/3/3

GPGPU処理系の自動最適化手法における シェアードメモリへのデータ転送方法の改良

神谷 智晴[1]　丸山 剛寛[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 処理系が生成するコードは最適化が不十分であり，手動最適化を施した CUDA コードと比べて実行時間が長くなることがある．一例として，GPU ではグローバルメモリの他，低容量だがアクセスレイテンシが短いシェアードメモリが複数存在し，手動最適化では両者を明示的に使い分ける．しかし従来の MESI-CUDA 実装ではグローバルメモリしか使用しない．そこで，我々は MESI-CUDA 上でシェアードメモリを用いるコードを自動生成する手法を開発している．本研究では，従来手法に対しシェアードメモリへのデータ転送部分の改良を行った．シェアードメモリへデータを転送する際，実行中のスレッドに合わせて格納するデータを入れ替えることでシェアードメモリの利用効率を向上させた．また，データを単純に分割して各シェアードメモリに格納するだけでなく，境界部分を重複して格納できるようにした．これにより従来手法では対応できなかったプログラムの最適化を可能としている．

キーワード：並列コンピューティング，GPGPU，CUDA

An Improved of Transferming Data of Shared Memory in GPGPU Programming Framework

Tomoharu Kamiya[1]　Takanori Maruyama[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 forhigh-performance computing. Major developing environment, such as CUDA, enables GPU programming using 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 modelautomatically generating low-level CUDA code. The current implementation of MESI-CUDA may generate inefficient code compared with the hand-optimized CUDA program, because the auto-generated code only uses the global memory of GPU. In this research, we improve our conventional method of transferring data to shared memory. Changing storing data in accordance with executing threads improves efficiency of using shared memory. We propose storing not only divided data but also data on the boundary doubly. These make it possible to optimize program which our conventional method cannot optimize.

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

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] 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 はグローバルメモリのみを使用する CUDA コードを生成しており，手動でメモリ階層を最適化した CUDA プログラムと比較すると実行時間が長くなるという問題がある．そこで, 我々は MESI-CUDA 上でシェアードメモリを用いるコードを自動生成する手法を開発している．

本研究では，従来手法に対しシェアードメモリへのデータ転送部分の改良を行った．シェアードメモリへデータを転送する際，実行中のスレッドに合わせて格納するデータを入れ替えることでシェアードメモリの利用効率を向上させた．また，データを単純に分割して各シェアードメモリに格納するだけでなく，境界部分を重複して格納できるようにした．これにより従来手法では対応できなかったプログラムの最適化を可能としている．

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 章ではデータ解析やコード生成などの自動最適化機構の手法を示す．6 章で，自動最適化機構の有無による 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．

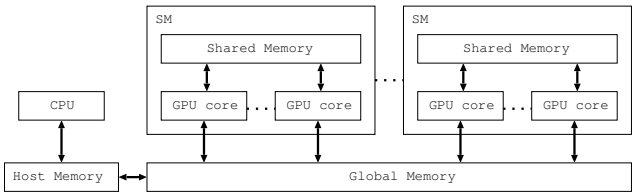

図 1　GPU のアーキテクチャモデル
Fig. 1　GPU Architecture Model

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 プログラムを容易に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 : 5 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 : 31 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 : 28-29 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 : 33 行）である．カーネルを実行するためにはカーネルで使用するデータの download 転送が完了している必要があり，カーネル実行後にホストが参照するデータについては readback 転送が完了している必要がある．

グリッド・ブロック

CUDA の仕様では，最高で 65535 × 65535 × 512 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が現在の CUDA では z 方向は 1 で固定となっ

```
 1 #include <stdio.h>
 2 #define N 2048
 3 #define BLOCKx 512
 4 #define BLOCKy 1
 5 __global__ void transpose(int *a, int *b, int *c){
 6   int k;
 7   int id=blockDim.x*blockIdx.x+threadIdx.x;
 8   c[id] = 0;
 9   for(k = 0;k < N;k++){
10     c[id] += a[k] * b[id+(k*N)];
11   }
12 }
13 void init_array(int d[N][N]){
                     ⋮
14 }
15 void output_array(int d[N][N]){
                     ⋮
16 }
17 int main(int argc, char *argv[]){
18   int ha[N*N], hb[N*N], hc[N*N];
19   int *da, *db, *dc;
20   int i, t;
21   cudaMalloc(&da,N*N*sizeof(int));
22   cudaMalloc(&db,N*N*sizeof(int));
23   cudaMalloc(&dc,N*N*sizeof(int));
24   dim3 grid(N/BLOCKx,N/BLOCKy);
25   dim3 block(BLOCKx,BLOCKy);
26   init_array((int(*)[N])ha);
27   init_array((int(*)[N])hb);
28   cudaMemcpy(da, ha , N*N*sizeof(int),
                        cudaMemcpyHostToDevice);
29   cudaMemcpy(db, hb , N*N*sizeof(int),
                        cudaMemcpyHostToDevice);
30   for (i = 0 ; i < N ; i++){
31     transpose<<<N/BLOCKx,BLOCKx>>>(da+(i*N),
                                    db,dc+(i*N) );
32   }
33    cudaMemcpy(hc, dc, N*N*sizeof(int),
                        cudaMemcpyDeviceToHost);
34   cudaFree(da);
35   cudaFree(db);
36   cudaFree(dc);
37   return 0;
38 }
```

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

ており，実際には 2 次元的に配置され管理されている．スレッドはブロック内で 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．

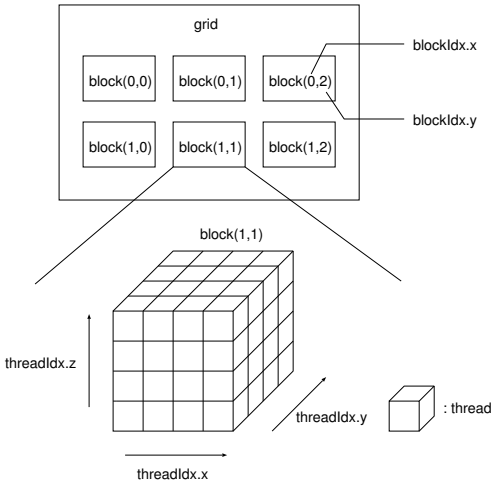

図 3 グリッド-ブロック-スレッド
Fig. 3 grid-block-thread

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同じ番号が重複してしまう．式 `blockDim.x` × `blockIdx.x` ＋ `threadIdx.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 2：21-23，34-36 行）．

シェアードメモリ

シェアードメモリは SM 毎に存在するオンチップメモリであり，同一ブロック内のスレッドが共有して使用できる（図 1）．グローバルメモリに比べて非常に高速なアクセスが可能となっている．また，バンクに分割されており，スレッド間のバンク・コンフリクトが無ければレジスタアクセスと同じ速さで処理することができる．カーネル関数内

では変数の型宣言の前に修飾子__shared__を付けることでシェアードメモリ上に領域が確保される．GPU プログラミングでは演算処理時間に対してデータアクセスレイテンシの割合が非常に大きく，レイテンシをいかに小さくできるかが高速化の鍵になっている．そこでアクセスレイテンシの小さいシェアードメモリにアクセス頻度の高いデータを格納することで実行時間を削減することができる．

3. 関連研究

GPGPU について，低レベルなアーキテクチャモデルを隠蔽し，より抽象的なプログラミングモデルを提供することでプログラミングの難易度を下げる研究が様々な観点から行われている．逐次的な処理を自動的に並列化する研究としては，for 文などのループに対する並列化が多くなされており，簡単なループ処理を含むプログラムについては良い結果を得ることができている [6][7]．しかし，非定型的な構造のプログラムや複雑なループについては，高性能な GPU 用のプログラムを得ることは困難である．また，メモリ階層についての支援ツールとして，自動的に各メモリ階層の特性に応じてデータの配置を自動的に行う研究 [8] がなされているが，GPU プログラムを解析して自動で割り当てるため，従来通りの GPU プログラミングを行う必要がある．

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[9] 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．

2013 年秋に発表された CUDA 6 では新たに実装される Unified Memory という機能を使用することでホスト側とデバイス側両方からアクセス可能なメモリを使用できる．加えて，CPU と GPU との間での通信量をドライバで最低限度の量に最適化することによる高速化も見込める．

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．また，コンパイルレベルで最適化を行うため今後解析性能が向上すればより高度な最適化が可能となる．そのため，CUDA 6 がランタイムレベルで自動に行う最適化よりも高い効果を得られる見込みがある．

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 : 4 行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．

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．

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．将来的にはユーザの必要に応じて低レベルの記述も可能とする予定である．

4.1.2 現在の処理系の問題点

従来手法ではシェアードメモリを使用しているが，効率が良いとは言えない．また，限られたプログラムしかシェアードメモリを使用する最適化を行うことができない．

```
 1 #include <stdio.h>
 2 #define N 1024
 3 #define BLOCKx 512
 4 __global__ int a[N][N], b[N][N], c[N][N];
 5 __global__ void transpose(int *a, int *b, int *c){
 6   int id = blockDim.x*blockIdx.x+threadIdx.x;
 7   int k;
 8   c[id] = 0;
 9   for (k = 0 ; k < N ; k++){
10     c[id] += a[k] * b[id+(k*N)];
11   }
12 }
13 void init_array(int d[N][N]){
                    ⋮
20   }
21 void output_array(int d[N][N]){
                    ⋮
30  }
31 int main(){
32   int i;
33   init_array(a);
34   init_array(b);
35   for(i=0;i<N;i++){
36      transpose<<<N/BLOCKx,BLOCKx>>>(a+(i*N), b, c+(i*N));
37   }
38 }
```

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

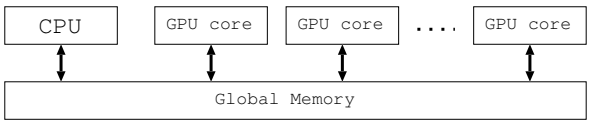

図 5　MESI-CUDA のプログラミングモデル
Fig. 5　Programming Model of MESI-CUDA

5. 自動最適化機構

5.1 概要

前述したように現在の MESI-CUDA 処理系の最適化は十分とはいえない．そこで，シェアードメモリを自動的に使用する CUDA コードを自動生成する機構を提案している [10]．本論文ではこの機構の改良について述べる．使用するデータをシェアードメモリに格納することでカーネル関数の高速化が可能となるが，容量が SM 毎に 64KB と非常に小さく，プログラム中で使用するすべてのデータを格納することは困難である．しかし，SM 毎に存在するため各ブロックごとにアクセスする部分のみを格納することで，64KB よりも大きなデータでも分割して格納することができる．また，効率的に用いるには使用頻度の高いデータを選択して格納する必要がある．

以下，既提案手法についての解析・コード生成の説明をした後，今回改良した境界部分の格納・シェアードメモリのデータ入れ替えについて述べる．その後，実際のプログラムを用いてコード生成の例を示す．

本機構では，配列アクセスのインデックスを解析して，ブロック内の使用頻度が高い配列を検出し，その配列についてシェアードメモリを使用する CUDA コードを自動生成する．今回実装する機構の対象とした MESI-CUDA プログラムは，1 次元のグリッド・ブロックで一重ループ中の 1 次元配列を扱うプログラムであり，シェアードメモリに変換する対象配列のアクセスが連続であるものとする．

5.2 解析

今回対象としたループ文を図 6 に示す．st，en は任意の定数式とする．このループ文中で，ある配列要素 $A[ix]$ をアクセスする場合を考える．ix は次式で表せるものとする．

$$a * i + b + c * \texttt{blockIdx.x} + d * \texttt{threadIdx.x}$$

ここで a，b，c，d は任意の定数式とする．本手法では，配列のアクセス範囲とアクセス頻度を解析する．

各スレッドのアクセス範囲は，ix 中のループ変数 i に for 文中から取得したその最小値と最大値を代入することで求めることができ，tc=b+c*blockIdx.x+d*threadIdx.x とすると，[a*st+tc ， a*(en-1)+tc] となる．また，1 スレッド内のアクセス回数は，ループ回数と一致するので (en-st) 回である．

次に，ブロック内のアクセス範囲は，ix 中の threadIdx.x にその最小値 (0) と最大値 (blockDim.x-1) を代入することで求めることができ，[a*st+b+c*blockIdx.x, a*(en-1)+b+c*blockIdx.x+d*(blockDim.x-1)] となる．したがって，ブロック内でアクセスされる範囲の大きさ (アクセスされる要素数) は {a*(en-1-st)+d*(blockDim.x-1)} である．また，ブロック内のアクセス回数は，各スレッド内のアクセス回数とブロック内のスレッド数の積で求められ，(en-st)*blockDim.x 回である．ブロック内のアクセス回数をブロック内のアクセス範囲の大きさで割ることで，配列の要素あたりの平均アクセス回数を求めることができ，次式で表せる．
{(en-st)*blockDim.x}/{a*(en-1-st)+d*(blockDim.x-1)}

```
for (i = st; i < en; i++){
...}
```

fig . 6　対象としたループ文
Fig. 6　Target Loop Statement

5.3 コード生成の概要

はじめにコード生成までの流れを図 7 に示す．解析によ

```
if(2 回以上アクセスがある配列がある)
    if(変換対象の配列が 1 つ)
        配列をシェアードメモリに格納
    else
        各配列のアクセス回数を解析
        1 番大きいものを格納配列とする
        本機構を使用しコード生成
else
    本機構を使用せずコード生成
```

fig . 7 コード生成までの流れ
Fig. 7 Applying Proposed Method

りシェアードメモリに格納することで高速化が見込める配列が存在する場合，本手法を適用する．このとき変換対象の配列が複数存在する場合は 5.2 節で示した解析方法でアクセス回数を求める．それを使い以下に示す方法でシェアードメモリに格納する配列を求めコード生成を行う．

図 8 に示すコード例を用いてコード生成の概要を説明する．図 9 は図 8 の配列 a，b，c のアクセス範囲を図示したものである．配列 a，b，c は要素数が同じですべてグローバルメモリ上にあるとし，網掛部は一つのスレッドのアクセス範囲を，斜線部は blockIdx.x が 0 のブロック内の全スレッドのアクセス範囲をそれぞれ示す．シェアードメモリに格納する配列は，ブロック内で必要な全要素の大きさが　シェアードメモリの容量< sizeof(配列の型)*N となるように指定しなければならない．また，ブロック内でのアクセス回数が多いほど効果が大きい．配列 c はブロック内のスレッドのアクセス範囲が配列全体であるため，データ容量がシェアードメモリの容量を超えてしまい格納できない．一方，配列 a，b は配列全体のデータ数は大きいもののブロック単位でのアクセス範囲は小さい．シェアードメモリは一つあたりの容量は小さいが SM 毎に存在するため，配列 a，b の様にブロック内のアクセス範囲が小さければその部分のみを抜き出すことで格納することができる．また，図 8 の例ではブロックでのアクセス範囲は配列 a，b ともに等しいが，配列 b は全スレッドがシェアードメモリに格納する部分をアクセスしている．この場合，配列 b の方がアクセス回数が多いためシェアードメモリに格納する対象とする．本来，アクセス回数が大きいものから順にシェアードメモリの容量を超えるまで配列を格納していくことが望ましい．しかし，現在の手法ではアクセス回数が最も大きいもの一つを格納している．

格納する配列が決まり，値の格納やコードの変換を行う際，グローバルメモリ上の配列とシェアードメモリ上の配列との要素数が異なるため幾つかの問題が発生する．グローバルメモリ上の配列から値をシェアードメモリ上の配列に代入する際，グローバルメモリ上の配列インデックスは連続しているがシェアードメモリ上の配列インデックスは各ブロックごとに 0 から始まるためグローバルメモリ上

```
for(i = 0 ; i < N; i++)
 sum +=b[blockIdx.x+i]*c[threadIdx.x*N+i];
a[id] = sum;
```

fig . 8 配列アクセスコードの例
Fig. 8 Code Example of Array Accesses

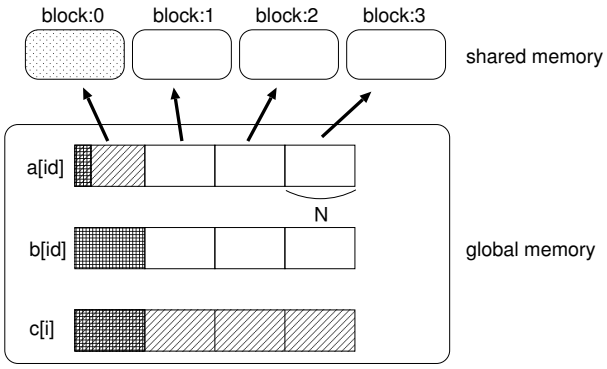

fig . 9 シェアードメモリへ格納する変数の例
Fig. 9 Allocating Array on Shared Memory

```
for(i = 0 ; i < N; i++)
data1[id]=data2[id]+data2[id+1]+data2[id-1];
```

fig . 10 境界部分の格納を用いる文
Fig. 10 Code Example Handling Boundary Data

の配列インデックスをそのまま使用できない．また，アクセス先をシェアードメモリ上の配列に変更するとループ文でのアクセスの仕方も変わるため，ループ変数や配列のインデックスを変更する必要がある．

5.4 境界部分の格納

前節で示した方法でシェアードメモリへデータを格納する場合，対象配列のインデックスによっては効率が悪いことがある．図 10 にその例を示す．従来手法の場合，シェアードメモリ格納対象となる配列は最もアクセス回数の大きいもののみであった．今，図 10 の文の data2[id] を格納対象の配列だとする．従来手法では配列名が同じでもインデックスが異なっていれば別の配列と見なしており格納対象としていなかった．しかし，図 10 の data2[id+1] や data2[id-1] のように格納対象配列と配列名が同じであり参照するデータの範囲がほぼ等しい配列が存在する場合，これらの配列も格納対象とすることでより効果的にシェアードメモリを利用することを考える．

シェアードメモリを使用する際，使用するデータが全てシェアードメモリ上に格納されていることが条件となる．従来の格納方法では data2[id+1] や data2[id-1] の変数が使用するデータは一部シェアードメモリ上に存在しない．そこでシェアードメモリに格納するデータの境界部分を重複して格納することで上記の配列をシェアードメモリ上のアクセスに変換できるよう拡張する．境界部分の格納の様子を図 14 に示す．境界部分の格納要素数はシェアー

```
for(i=0;i<L;i++){
 res[id+d]=target[i+a]*_g_v[b*id+c]
}
```

fig . 11 データ入れ替えを行う対象のコード
Fig. 11 Target Code for Data Swapping

```
for(i=0;i<L-blockDim.x;i+=blockDim.x){
 _s_v[treadIdx.x]=_g_v[threadIdx.x+i];
 __syncthreads();
 for(k=0;k<blockDim.x;k++)
 res[id+d]=_s_v[i+k+a]*_g_v[b*id+c]
 }
 if(threadIdx.x<L-i){
 _s_v[treadIdx.x]=_g_v[threadIdx.x+i];
 __syncthreads();
 for(k=0;k<blockDim.x;k++)
 res[id+d]=_s_v[i+k+a]*_g_v[b*id+c]
 }
```

fig . 12 Fig.11 から変換したコード
Fig. 12 Code Transformed from Fig.11

ドメモリに格納する配列の型と要素数を考慮して一定数取ることができる．これにより，元々シェアードメモリに格納されているデータに加えその前後のデータそれぞれ K 個ずつシェアードメモリ上に存在することとなる．使用するデータがこの範囲内に収まる配列が今回の手法の格納対象となる．

ある SM 上で配列のある要素がシェアードメモリ上にコピーされているとき，他の SM 上ではグローバルメモリ上でその要素をアクセスする場合がある．また，境界部分については複数のシェアードメモリ上に同じ要素をコピーし，それぞれアクセスする可能性がある．このとき，同時に複数個所で書き込みが発生するとデータの整合性が取れなくなる可能性がある．しかし，CUDA ではブロックの異なるスレッド間で実行中に同期をとることができず，このような競合的書き込みの結果はもともと保証されていない．したがって，本手法を用いても実用上問題ないと言える．

5.5 シェアードメモリのデータ入れ替え

アクセス回数が最大の配列をシェアードメモリに格納することでより効果的に使用できる．しかし，アクセス回数が多い配列が存在してもシェアードメモリの容量を超えていて格納できない場合がある．そのため従来手法ではシェアードメモリの格納対象を選出する時，シェアードメモリに格納可能な大きさの配列のみを格納候補としていた．そこで本手法ではアクセス回数が最大の配列を格納するため，スレッドの動作に合わせてシェアードメモリのデータを入れ替える機能を追加する．

5.6 コード生成

5.3 節で示した方法から得た変数に対し，以下の流れでコード生成を行う．

また，図 2 のプログラムに対し，提案手法を用いて生成されたコードを図 19 に示す．

(1)シェアードメモリ上に領域確保するコードの挿入
(2)グローバルメモリからシェアードメモリへデータコピーするコードを挿入
(3)グローバルメモリアクセスのコードをシェアードメモリアクセスするコードへ変換，それに伴う配列インデックスの変換
(4)シェアードメモリからグローバルメモリへデータをコピーするコードを挿入

シェアードメモリの領域確保

解析からシェアードメモリに格納する配列のアクセス範囲を得ており，その範囲分と境界部分の容量をまとめて確保する．変数宣言の最後にシェアードメモリの領域を確保するコードを挿入する（図 19:8 行）．

グローバルメモリからシェアードメモリへのデータコピー

CUDA でデータをコピーする場合，配列のインデックスに id を用いて各スレッドが異なる配列の要素を代入する方法がよく用いられる．しかし，5.3 節で述べたようにコピー元とコピー先でインデックスがずれているため正しく格納できない（図 13 : (a)）．

そこで図 13 : (b) のようにシェアードメモリ側の配列 s_array のインデックスを threadIdx.x とすることで正しい場所に格納できる．また，境界部分の要素を格納するため余分に領域を確保する場合はそれも考慮する必要がある (図 14)．格納に用いるコードを図 15 に示す．N，M，L はそれぞれシェアードメモリの要素数，ブロック内のスレッド数，グローバルメモリの要素数を表している．このコードをシェアードメモリの領域を確保した後すぐに挿入する（図 19:9-16 行）．threadIdx.x が 0 と BLOCKx-1 となるスレッドに境界部分のデータのコピーを行わせている．また，シェアードメモリはブロック内の全スレッドがアクセスするため，最後のスレッドがコピーを終了するまで他のスレッドは計算を始めずに待機する必要がある．そのため，図 19:17 行のようにコピーのすぐ後に同期を挿入している．

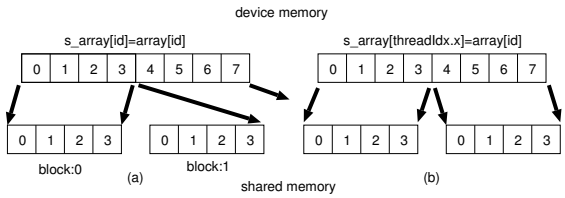

fig . 13 シェアードメモリへのデータコピー
Fig. 13 Copying Data to Shared Memory

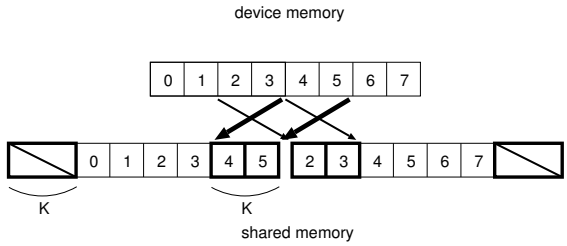

fig . 14 境界部分の格納
Fig. 14 Copying of Boundary Data

```
for(i = threadIdx.x ; i < N; i +=M)
_s_v[i+K]=_g_v[i+blockDim.x
*blockIdx.x+(N-M)*blockIdx.x];
if(id!=0&&threadIdx.x==0)
for(j = 0; j < K;j ++)
_s_v[threadIdx.x+j] = _g_v[id-K+j];
if(i != L-1 && i == N-1){
for(j =0;j<K;j++)
_s_v[i+K+1+j] = _g_v[id+1+i];
```

fig . 15 シェアードメモリへの格納コード
Fig. 15 Code Copying to Shared Memory

シェアードメモリへアクセスするコードに変換，それに伴うインデックスの変換

本機構ではコード変換を行う際，変換対象の変数を含む式内のループ変数の有無により 3 通りの変換を行っている．

式内にループ変数が存在しない場合

グローバルメモリアクセスをしていたコードの変数名を変更する．

式内にループ変数が変換する配列のインデックスにのみ存在する場合

シェアードメモリの入れ替えはこの形の時のみ行う．

図 11 に変換対象となるコードを，図 12 に変換後のコードをそれぞれ示す．なお図 11 と図 12 の各配列名は対応している．L，a，b，c，d は int 型の定数式とする．$target$ はシェアードメモリへの格納対象となる配列である．このコードに変換を行うと図 12 の様になる．今，配列 $target$ の要素 L が大きく使用する全てのデータがシェアードメモリ上に格納できないとする．ループ変数の増加値を 1 から blockDim.x に変更し，内側に新たなループ文を 0 から blockDim.x の範囲で 1 ずつ増加させるように挿入する．これによりシェアードメモリに格納できる容量で分割して処理を行うことができる．シェアードメモリの入れ替えを行う際，2 つのループ文の間でグローバルメモリからシェアードメモリへのデータコピーを行う．このとき $BLOCKx$ の値がシェアードメモリの要素数以上の時，不正なデータ転送が起こってしまう．そのため転送前に if 文で制御し，不正なデータ転送を防いでいる．また，target のインデックス i+a を i+k+a に変更する．

式内にループ変数が存在する場合

5.3 節で述べたように格納対象を含む式にループ変数が含まれている時，シェアードメモリ上の配列にアクセス先を変更するとループ文内のアクセスも変更する必要がある．以下に簡単な例を示す．図 16（a）の様なコードを考える．配列 g_array，res，target はグローバルメモリ上，配列 s_array はシェアードメモリ上にあるとし，target の値を s_array に代入することとする．このとき target の前半 4 要素と後半 4 要素を，blockIdx.x=0，1 のブロックにそれぞれ格納している．シェアードメモリの要素数に合わせようとループ数を変更すると配列 g_array の前半 4 要素を二重にアクセスしてしまい正しい結果を得ることができない（b）．そこで，図 16 :（c）のようにループ変数を二重化し，配列のインデックスを変換することで各ブロックが正しい場所へアクセスできるようにしている．

図 17 に変換対象となるコードを，図 18 に変換後のコードをそれぞれ示す．なお図 17 と図 18 の変数名は対応している．L，a，b，c，d は int 型の定数式とする．$target$ はシェアードメモリへの格納対象となる配列である．このコードに変換を行うと図 18 の様になる．ループの範囲を blockDim.x × blockIdx.x-blockDim.x × blockIdx.x + blockDim.x − 1 と変更し，さらに 0-blockDim.x − 1 の範囲で変化するループ変数を加える．ループの内側にもう 1 つループ文を 0-L − 1 の範囲で blockDim.x ずつ増加させるように挿入する．シェアードメモリ上の配列のインデックスは追加したループ変数（図 18 の j）に変更する．また，グローバルメモリ上の配列のインデックス id は threadIdx.x + k（内側のループ変数）に変更する．

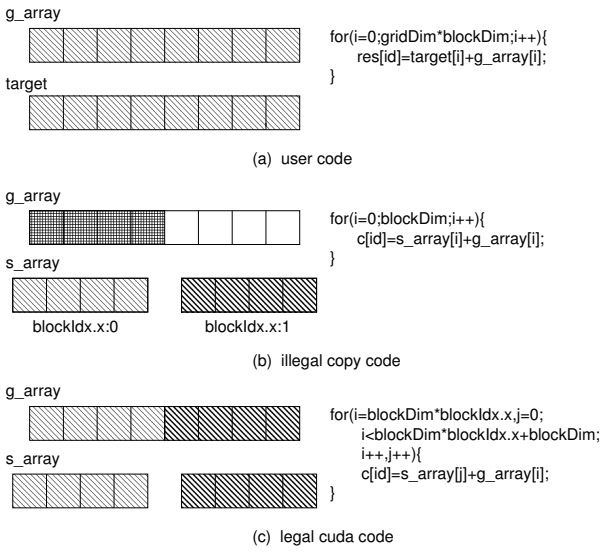

fig . 16 ループ時の配列へのアクセス
Fig. 16 Array Accesses in Loop

シェアードメモリからグローバルメモリへのデータのコピー

シェアードメモリに書き込みが行われた場合ループ文の後にグローバルメモリの配列へデータのコピーを行うコードを挿入する．この際のコードは図 15 に示した代入文の

```
for(i=0;i<L;i++){
 res[id+d]=target[i+a]*_g_v[b*i+id+c]
}
```

fig . 17 従来手法の対象となるコード
Fig. 17 Target Code of Conventional Method

```
for(i=blockDim.x*blockIdx.x,j=0;
 i<blockDim.x*blockIdx.x+blockDim.x;i++,j++){
 for(k=0;k<L;k+=blockDim.x){
 res[threadIdx.x+k+d] = _s_v[j+a]
 *_g_v[b*i+threadIdx.x+k+c];
 }
}
```

fig . 18 Fig.17 から変換したコード
Fig. 18 Code Transformed from Fig.17

右辺と左辺を交換したものとなる．

5.6.1 提案手法を用いた例

ここでは 5.6 節で示した手法を適用して図 4 を図 19 に変換する過程を示す．解析結果からシェアードメモリに格納する変数が a（図 4:10 行）となったとする．この場合，コード生成の対象となる部分は図 4:9-11 行である．はじめにシェアードメモリの領域確保を行うコードを挿入する（図 19:8 行）．続いてシェアードメモリへのデータのコピーを行うコードを挿入する（図 19:9-16 行）．ここでは図 15 で示した文の N，M が共に BLOCKx なためループ文は一回で終了する．また，変換対象の式に変数 a は 1 つだけなので K も 0 となる．次に，ループ文の変換を行う．変換のために必要な変数を宣言し（図 19:7 行），変数 a を _s_a に変更する．それに伴い，図 18 に示した様にループ文を変更していく．今回は，シェアードメモリへの書き込みが行われていないためシェアードメモリからグローバルメモリへのデータコピーは行わない．以上で変換が完了する．

```
 1 #define BLOCKx 512
 2 #define N 2048
 3 #define K 0
 4 __global__ void transpose(int *a, int *b, int *c){
 5   int k;
 6   int id=blockDim.x*blockIdx.x+threadIdx.x;
 7   int _j,_l,_m,_n;
 8   __shared__ int _s_a[BLOCKx+2*K];
 9   for(_l=threadIdx.x;_l<BLOCKx;_l+=BLOCKx)
10     _s_a[_l+K] =
         a[_l+BLOCKx*blockIDx.x+
      (BLOCKx-BLOCKx)*blockIdx.x)];
11   if(id!=0 && threadIdx.x==0))}
12     for(_j=0;_j<K;_j++)
13       _s_a[threadIdx.x+_j] = a[id-K+_j];
14   if(_l!=N-1 && _l==BLOCKx-1))}
15     for(_j=0;_j<K;_j++)
16       _s_a[_l+K+1+_j] = a[id+1+_l];
17   __syncthreads();
18   c[id] = 0;
19   for(k=blockDim.x*blockIdx.x,_n=0;
        k<blockDim.x*blockIdx.x+blockDim.x;
        k++,_n++){
20      for(_m=0;_m<N _m+=blockDim.x){
21        c[threadIdx.x+_m] += _s_a[_n+K]
                        * b[threadIdx.x+_m+(k*N)];
22     }
23   }
24 }
```

fig . 19 MESI-CUDA で生成した CUDA コード
Fig. 19 CUDA Code Generated by MESI-CUDA Compiler

6. 評価

実装した自動最適化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 CUDA プログラムの実行時間の比較を行った．評価環境は 3 種類の実行環境

- Core i7 930 2.80GHz，メモリ 6GB，TeslaC2050
- Core i7 3820 3.60GHz，メモリ 8GB，Geforce GTX680
- Xeon E5-1620 3.60GHz，メモリ 16GB，TITAN

をそれぞれ搭載した計算機を使用した．評価には拡散方程式と，ヒストグラムを求めるプログラムを用いた．拡散方程式はデータサイズが 8192，16384，32768，65536 の場合に 1000 回拡散処理を行った時の実行時間を測定した．ヒストグラムはデータサイズ 3200，6400，12800，25600 の場合の実行時間を測定した．結果を表 1，表 2，表 3 にそれぞれ示す．表からわかるように，拡散方程式では GTX680 使用時，データサイズ 32768 の場合に実行時間が従来手法と比べて約 73%短縮されている．ヒストグラムでは TeslaC2050 使用時，データサイズ 6400 の場合に実行時間が従来手法と比べて約 8%短縮されている．

これは，本機構によって生成したコードが前述したシェアードメモリを効果的に使用しており，これによってメモリアクセスのレイテンシが短縮されたためである．ヒストグラムにおいては従来手法と提案手法とで格納した配列のアクセス回数の差が大きくなかったためあまり性能向上が得られなかった．また，本機構は Fermi コア（TeslaC2050）と Kepler コア（GTX680，TITAN）の両方で性能向上が得られたことから，GPU アーキテクチャの環境に左右されずに一定の効果が上げられるといえる．

表 1　TeslaC2050 での実行時間 (秒)
Table 1　Execution Time on TeslaC2050

	拡散方程式		
	従来手法	境界部分格納	実行時間比 (%)
8192	0.0929	0.0872	93.9
16384	0.164	0.160	97.6
32768	0.259	0.245	94.6
65536	0.479	0.425	88.7
	ヒストグラム		
	従来手法	データ入れ替え	実行時間比 (%)
256	2.168	2.008	92.6
512	4.222	3.910	92.6
1024	11.575	11.520	99.5
2048	45.132	42.577	94.3

表 2　GeForece GTX680 での実行時間 (秒)
Table 2　Execution Time on GeForce GTX680

	拡散方程式		
	従来手法	境界部分格納	実行時間比 (%)
256	0.148	0.0684	46.2
512	0.288	0.0840	29.2
1024	0.573	0.155	27.1
2048	0.474	0.287	60.5
	ヒストグラム		
	従来手法	データ入れ替え	実行時間比 (%)
256	4.373	4.126	94.4
512	6.208	5.802	93.5
1024	13.243	12.494	94.3
2048	39.272	39.151	99.7

表 3　TITAN での実行時間 (秒)
Table 3　Execution Time on TITAN

	拡散方程式		
	従来手法	境界部分格納	実行時間比 (%)
256	0.133	0.0873	65.6
512	0.152	0.0935	61.5
1024	0.271	0.159	58.7
2048	0.423	0.251	59.3
	ヒストグラム		
	従来手法	データ入れ替え	実行時間比 (%)
256	1.183	1.106	93.5
512	2.442	2.349	96.2
1024	7.707	7.297	94.7
2048	28.591	26.973	94.3

7. おわりに

本研究では MESI-CUDA 上に，シェアードメモリを利用する自動最適化機構を設計・実装し，評価を行った．その結果，本機構を用いることで適切な配列のアクセス解析が行われ，シェアードメモリを利用する CUDA コードが自動生成できた．今後の課題として，本研究では簡単な配列のアクセスにのみ対応しているが，より複雑な場合に対応していく必要がある．また，コード生成アルゴリズムが対応しているプログラムの範囲が狭いため，より汎用的なアルゴリズム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* におけるデータ転送を自動化する *MESI-CUDA* の提案, 先進的計算基盤システムシンポジウム SACSIS2012,201-209,(2012).
[5] 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).
[6] 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).
[7] Muthu Baskaran，J.Ramanujam and P.Sadayappan: *Automatic C-to-CUDA Code Generation for Affine Programs*,Springer Berlin / Heidelberg,(2010).
[8] 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).
[9] *OpenACC*, 入手先 ⟨http://www.openacc-standard.org/⟩,(2013.06.7).
[10] 神谷 智晴, 丸山 剛寛, 松本 真樹 and 大野 和彦: *GPGPU* のシェアードメモリを利用する自動最適化機構, 情報処理学会研究報告 2013-HPC-140(30),1-8,(2013).