Vol.2013-HPC-140 No.30
2013/8/2

GPGPUのシェアードメモリを利用する自動最適化機構

神谷 智晴[1]　丸山 剛寛[1]　松本 真樹[1,†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 が生成するコードはプログラマが手動最適化した CUDA コードと比べて実行時間が長くなることがある．そこで，我々は MESI-CUDA 上に，メモリアクセスレイテンシの短いシェアードメモリを自動で使用する機構を実現した．本手法では，シェアードメモリに格納するデータ選出のため静的解析により各データのアクセス頻度を求める．続いて解析結果を元にデータのシェアードメモリへのコピーコードを挿入し，アクセス先の変数名と配列インデックスの書き換えを行う．提案手法適用の有無によるプログラムの実行時間を比較して評価を行った結果，本機構により実行時間を最大約 1/3 まで短縮できた．

キーワード：並列コンピューティング，GPGPU，CUDA

Automatic Optimization Scheme using Shared Memory in GPGPU

Tomoharu Kamiya[1]　Takanori Maruyama[1]　Masaki Matumoto[1,†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. 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 model automatically 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. Thus, we propose a scheme which automatically generates code using the fast shared memories on GPU. Using static analysis, our scheme estimates the access frequency of arrays and determines which array should be cached on the shared memories. Then the target code is modified to copy the array to the shared memories. The code accessing the array is also modified to access the copy. The evaluation result shows that using our scheme, the execution time is reduced to 1/3 at maximum.

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

[1] 三重大学大学院 工学研究科
Mie University
[†1] 現在，株式会社　医用工学研究所
Presently with Medical Engineering Institute,Inc.

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実現した．

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．

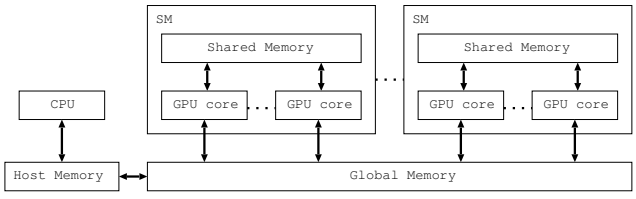

図 1 GPU のアーキテクチャモデル
Fig. 1 GPU Architecture Model

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 : 34 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 : 31-32 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 : 36 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．

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

```
 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, *hb, *hc;
19   int *da, *db, *dc;
20   int i, t;
21   cudaMallocHost(&ha,N*N*sizeof(int));
22   cudaMallocHost(&hb,N*N*sizeof(int));
23   cudaMallocHost(&hc,N*N*sizeof(int));
24   cudaMalloc(&da,N*N*sizeof(int));
25   cudaMalloc(&db,N*N*sizeof(int));
26   cudaMalloc(&dc,N*N*sizeof(int));
27   dim3 grid(N/BLOCKx,N/BLOCKy);
28   dim3 block(BLOCKx,BLOCKy);
29   init_array((int(*)[N])ha);
30   init_array((int(*)[N])hb);
31   cudaMemcpy(da, ha , N*N*sizeof(int),
                        cudaMemcpyHostToDevice);
32   cudaMemcpy(db, hb , N*N*sizeof(int),
                        cudaMemcpyHostToDevice);
33   for (i = 0 ; i < N ; i++){
34     transpose<<<N/BLOCKx,BLOCKx>>>(da+(i*N),
                                    db,dc+(i*N) );
35   }
36    cudaMemcpy(hc, dc, N*N*sizeof(int),
                        cudaMemcpyDeviceToHost);
37   cudaFree(da);
38   cudaFree(db);
39   cudaFree(dc);
40   cudaFreeHost(ha);
41   cudaFreeHost(hb);
42   cudaFreeHost(hc);
43   return 0;
44 }
```

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

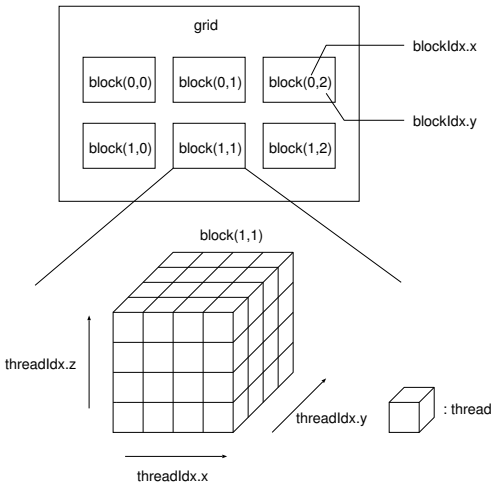

図 3 グリッド-ブロック-スレッド
Fig. 3 grid-block-thread

ブロックの数だけ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 2：24-26，37-39 行）．

シェアードメモリ

シェアードメモリは同一ブロック内のスレッドが共有して使用できるオンチップメモリである（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，バンクと呼ばれるモジュールに分割されており，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 [7][8]．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 [9] 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[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．

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．

```
 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 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．

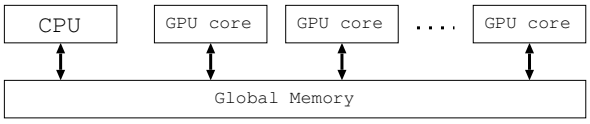

図 5　MESI-CUDA のプログラミングモデル
Fig. 5　Programming Model for MESI-CUDA

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

現在の実装はグローバルメモリを使用するコードしか

生成することができない．そのため，プログラマが手動最適化した CUDA コードに比べ実行時間が長くなることがある．

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使用する CUDA コードを自動生成する機構を実現した．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．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．

$$\{(en-st)*\texttt{blockDim.x}\}/\{a*(en-1-st)+d*(\texttt{blockDim.x}-1)\}$$

平均アクセス回数が高いものは参照頻度も高いものと考えられるので，これをアクセス頻度としている．

```
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

図 7 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 8 は図 7 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変数は，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を超えてはならない．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 7 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変数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．

格納する変数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

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

fig . 7 配列アクセスコードの例
Fig. 7 Code Example of Array Accesses

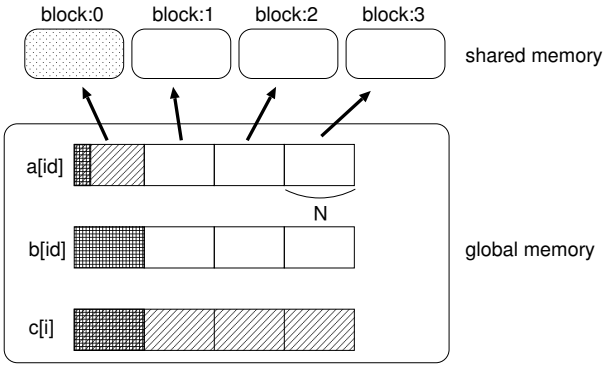

fig . 8 シェアードメモリへ格納する変数の選別

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 コード生成

5.4 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 14 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変数のアクセス範囲を得ており，その範囲分だけ容量を確保する．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 14:7 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，5.4 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 9 : (a)）．そこで図 9 : (b) のようにシェアードメモリ側の配列 s_arrray 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．

格納に用いるコードを図 10 に示す．N,M はそれぞれシェアードメモリの要素数とブロック内のスレッド数とを表している．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 14:8-10 行）．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 14:11 行のようにコピーのすぐ後に同期を挿入している．

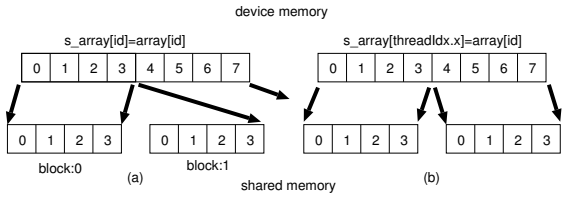

fig . 9 シェアードメモリへのデータコピー
Fig. 9 Data Copy to Shared Memory

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

5.4 節で述べたように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 11（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 11 : (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 12 に変換対象となるコードを，図 13 に変換後のコードをそれぞれ示す．なお図 12 と図 13 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 13 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 13 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．

```
for(i = threadIdx.x ; i < N; i +=M)
shared_memory[i]=global_memory[id+(i-1)*M];
```

fig . 10 シェアードメモリへの格納文
Fig. 10 Code Copying to Shared Memory

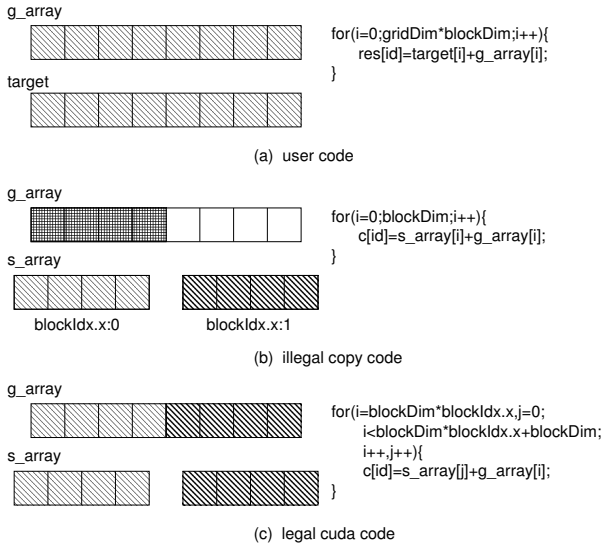

fig . 11 ループ時の配列へのアクセス
Fig. 11 Array Access in Loop

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

fig . 12 対象となるコード
Fig. 12 Target Code

```
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] = shared_memory[j+a]
 *global_memory[b*i+threadIdx.x+k+c];
 }
}
```

fig . 13 変換後のコード
Fig. 13 Transformed Code

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 10 に示した代入文の右辺と左辺を交換したものとなる．

5.4.1 提案手法を用いた例

ここでは 5.5 節で示した手法を適用して図 4 を図 14 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 14:7 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 14:8-10 行）．ここでは図 10 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．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 14:6 行），変数 a を _s_a に変更する．それに伴い，図 13 に示した様にループ文を変更していく．

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

fig . 14 MESI-CUDA で生成した CUDA コード
Fig. 14 Generated Program using MESI-CUDA

```
for(i = threadIdx.x ; i < N; i +=M)
shared_memory[i]=global_memory[a*id+b+(i-1)*M];
```

fig . 15 定数倍アクセスの格納文
Fig. 15 Statement to Store Constant Access

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．

5.4.2 二次元配列の記述

CUDA ではデバイスメモリに領域を確保する場合 cudaMalloc を使用する必要があるため多次元配列をとることは困難である．しかし，CUDA で扱うようなプログラムのデータが 2 次元配列であることは多々ある．そのため，ユーザは 1 次元配列にデータを変更する必要があった．MESI-CUDA ではその作業を自動化するためにカーネル起動時に 2 次元配列の記述を可能とするように拡張を行っている．これを生かすため，5.5 節で示した手法では一次元配列に連続してアクセスする場合のみシェアードメモリを使用したコード生成を行っていたが，配列のインデックスが $a*i+b$（i はループ変数）の様な定数間隔アクセス時もシェアードメモリを使用したコード生成を行えるように拡張した．図 15 に格納するためのコードを示す．N はシェアードメモリの容量，M はブロック内のスレッド数，a はアクセス間隔，b はアクセス開始点とする．このコードは 5.3 節で説明した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と同じ場所にコードを挿入し同期をとる．

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 2 種類の実行環境 Core i7 930 2.80GHz，メモリ 6GB，TeslaC2050 と Core i7 3820 3.60GHz，メモリ 8GB，Geforce GTX680 をそれぞれ搭載した計算機を使用した．評価には行列積と逆行列を求めるプログラムを用いた．正方行列の一辺の大きさが 256，512，1024，2048 の場合の実行時間を測定した．結果を表 1，表 2 にそれぞれ示す．表から分かるように，TeslaC2050 では一辺の大きさが 2048 の場合に両方のプログラムで実行時間が約 1/3 まで短縮されている．GTX680 では一辺の大きさが 2048 の場合に行列積では約 1/3，逆行列では約 2/3 まで短縮されている．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．また，本機構は Fermi コア（TeslaC2050）と Kepler コア（GTX680）の両方で結果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

	行列積			逆行列		
	不使用	使用	実行時間比 (%)	不使用	使用	実行時間比 (%)
256	0.0276	0.0282	102.1	1.520	0.969	63.7
512	1.437	1.346	93.6	6.434	3.042	47.2
1024	5.708	2.956	51.7	44.730	20.190	45.1
2048	23.632	7.551	31.9	120.751	53.194	44.1

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

	行列積			逆行列		
	不使用	使用	実行時間比 (%)	不使用	使用	実行時間比 (%)
256	0.1617	0.1465	90.6	0.6883	0.4784	69.5
512	0.8635	0.4737	54.9	2.889	2.051	70.9
1024	3.421	1.701	49.7	15.03	11.15	74.2
2048	13.99	5.351	38.2	90.01	64.97	72.2

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).