修士論文

題目

GPGPUフレームワーク MESI-CUDAにおける 自動最適化のための配列インデックスの静的解析手法

指導教員

大野　和彦　講師

平成26年度

三重大学大学院 工学研究科 情報工学専攻
コンピュータソフトウェア研究室

丸山　剛寛 (413M525)

三重大学大学院　工学研究科

内容梗概

現在主流の GPGPU 開発環境である CUDA では，一定数のコア毎にシェアードメモリを持つなど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．

我々の提案している MESI-CUDA では単純なメモリ構造モデルでプログラミング可能であるが，現在の MESI-CUDA が生成するコードは，手動で最適化された CUDA コードと比較すると実行時間が長くなることがある．そこで我々は MESI-CUDA において複雑なメモリ上のデータ配置や複数 GPU への負荷分散，スレッドマッピングなどを自動最適化する手法を開発している．

これらを実現するためには，ユーザの記述したコードを静的解析し，最適化に必要な情報を取得する必要がある．そこで，本手法では，配列アクセス時のインデックス式を静的解析することで各スレッドにおける配列要素のアクセス回数やアクセス範囲を求める．さらにこれをもとにシェアードメモリを共有するスレッド群のアクセス回数やアクセス範囲を求める．

三重大学大学院　工学研究科

Abstract

In CUDA which is current mainstream GPGPU development environment, the user has to deal complex memory structure such as each crowd of cores has shared memory.

The user can program with simple memory structure using MESI-CUDA which we are developing. However, the current implementation may generate inefficient code compared with the hand-optimized CUDA program. Therefore, we propose a scheme which automatically generates code such as data allocation on the complex memory structure, load balancing for multi-GPUs, and mapping threads.

To achieve these generating code, it is necessary to analyze statically the code written by the user and obtain parameter for optimization. Therefore, in our proposed scheme, statically analyzing index expressions of accessing array obtain access counts and accessed range of element of arrays by each thread. Furthermore, using these result obtain access counts and accessed range of element of arrays by each crowd of threads sharing shared memory.

三重大学大学院　工学研究科

目 次

1 はじめに **1**

2 背景 **2**

2.1 GPU アーキテクチャ 2
2.2 CUDA 2
2.3 CUDA における最適化手法 5

3 関連研究 **7**

4 MESI-CUDA **8**

4.1 MESI-CUDA 概要 8
4.2 現在の処理系の問題点 9

5 自動最適化手法 **10**

6 解析手法 **11**

6.1 概要 11
6.2 解析 12
6.2.1 シェアードメモリおよびマルチ GPU 最適化のための解析 13
6.2.2 スレッドマッピング最適化のための解析 14
6.3 適用例 14
6.4 アクセス範囲解析の改良 16

7 評価 **17**

8 終わりに **19**

謝辞 **20**

参考文献 **21**

図 目 次

2.1 GPU のアーキテクチャモデル 2
2.2 行列積を求める CUDA コード 3
2.3 グリッド-ブロック-スレッド 4
4.4 CUDA コードと等価な MESI-CUDA コード 9
4.5 MESI-CUDA のプログラミングモデル 10
6.6 論理スレッドを用いた MESI-CUDA コードの例 15
6.7 二次元配列アクセスの様子 16

表 目 次

6.1 式の範囲 14
6.2 解析結果 15
6.3 各点の座標 17
7.4 シェアードメモリ最適化の実行時間比較 (秒) 18
7.5 マルチ GPU 最適化の実行時間比較 (秒) 18
7.6 スレッドマッピング最適化の実行時間比較 (秒) 19

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，現在の MESI-CUDA が生成する CUDA コードは，手動で最適化された CUDA コードと比較すると実行時間が長くなることがある．そこで我々は MESI-CUDA において複雑なメモリ上のデータ配置や複数 GPU への負荷分散，スレッドマッピングなどを自動最適化する手法を開発している．これらを実現するためには，ユーザの記述したコードを静的解析し，最適化に必要な情報を取得する必要がある．本稿では，配列アクセス時のインデックス式を静的解析し，配列要素のアクセス回数やアクセス範囲を求める．

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 章では自動最適化手法を示し，6 章で今回提案する解析手法とその適用例について示す．7 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，8 章でまとめを行う．

2　背景

2.1　GPU アーキテクチャ

図 2.1 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．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 毎にオンチップメモリであるシェアードメモリを持つ．

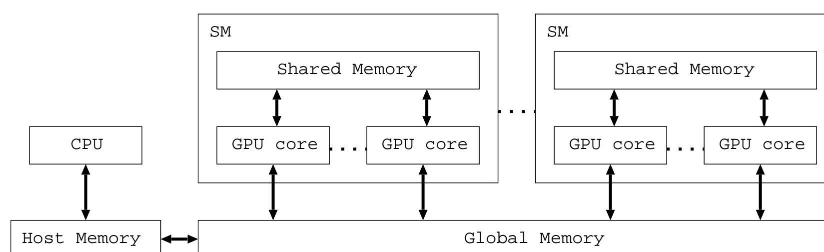

図 2.1: GPU のアーキテクチャモデル

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 2.2 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 2.2：5 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 2.2：26 行）．このときに作成するスレッド数を指定する．

```
 1 #define N 1024
 2 #define BX 128
 3 #define S (N*N*sizeof(int))
 4 int ha[N][N], hb[N][N], hc[N][N];
 5 __global__
   void transpose(int a[][N], int b[][N], int c[][N]){
 6   int k , tmp = 0;
 7   int row = blockDim.y*blockIdx.y+threadIdx.y;
 8   int col = blockDim.x*blockIdx.x+threadIdx.x;
 9   for(k = 0 ; k < N ; k++){
10     tmp += a[row][k] * b[k][col];
11   }
12   c[row][col] = tmp;
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 *da, *db, *dc;
18   dim3 dimGrid(N/BX, N);
19   cudaMalloc(&da, S);
20   cudaMalloc(&db, S);
21   cudaMalloc(&dc, S);
22   init_array(ha);
23   init_array(hb);
24   cudaMemcpy(da, (int*)ha, S, cudaMemcpyHostToDevice);
25   cudaMemcpy(db, (int*)hb, S, cudaMemcpyHostToDevice);
26   transpose<<<dimGrid, BX>>>
         ((int(*)[N]))da,(int(*)[N]))db,(int(*)[N]))dc);
27   cudaMemcpy((int*)hc, dc, S, cudaMemcpyDeviceToHost);
28   output_array(hc);
29   cudaFree(da);
30   cudaFree(db);
31   cudaFree(dc);
32 }
```

図 2.2: 行列積を求める CUDA コード

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 2.2：24-25 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 2.2：27 行）である．

グリッド・ブロック　現在の CUDA の仕様では，最高で $2147483647 \times 65535 \times 65535 \times 1024$ 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 2.3）．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．

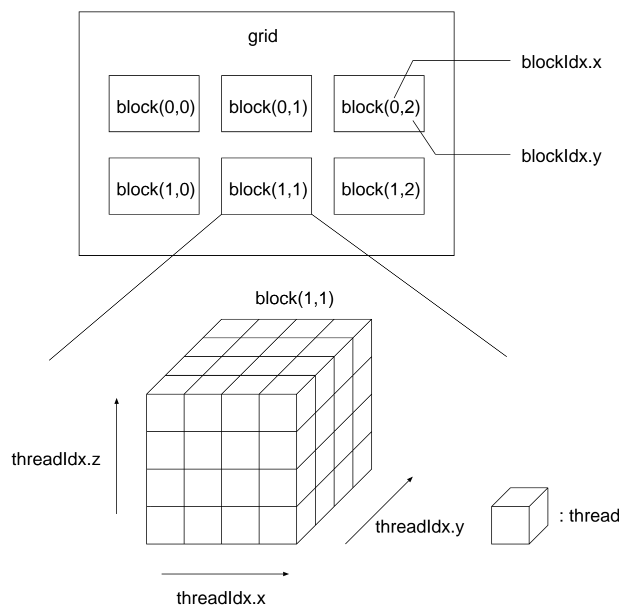

図 2.3: グリッド-ブロック-スレッド

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．

ワープ　CUDA では全てのスレッドが並列動作するように振る舞うが，ブロック中のスレッドは 32 スレッド毎にワープと呼ばれる実行単位にまとめられており，各 SM は一度に一つのワープを並列実行する．一つのワープの実行が終了すると，同じブロックの他のワープが同じ SM 上で実行される．CUDA では基本的にはブロック/スレッドレベルでプログラミングを行うが，ワープレベルの最適化を行うことで更なる高速化が可能である．しかし，そのためにはアーキテクチャの知識や低レベルなコーディングが必要である．

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 2.2：19-21, 29-31 行）．

シェアードメモリ　同一ブロック内のスレッドは GPU のシェアードメモリを共有して使用できる．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．

マルチ GPU　計算機に GPU を複数搭載して並列動作させることができる．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．

2.3　CUDA における最適化手法

CUDA では GPU のアーキテクチャを意識した低レベルなコードを記述することで最適化を行うことができる．

アクセスのコアレス化　ワープ内の各スレッドが同一の L2 キャッシュライン (128 バイト領域) をアクセスするならば，単一のトランザクションで処理され，コアレスアクセスとなる．これによりグローバルメモリへのアクセス回数が削減されるので高速化が可能である．そのためプログ

ラム中で配列をアクセスする場合は，ワープ内のスレッドが連続する要素をアクセスするようにスレッド数/ブロック数やインデックス式を調整するのが望ましい．

シェアードメモリの利用　シェアードメモリは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．そこで，使用するデータをシェアードメモリに格納する明示キャッシュによりカーネル関数の高速化が可能となる．シェアードメモリの生存期間はカーネル関数内であるため，最適化はカーネル関数単位で良い．シェアードメモリの容量はSM毎に48KB [1]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が困難であるが，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，48KB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．

マルチGPUの利用　マルチGPUを利用できる場合は，各GPUに処理を分割して振り分けることで高速化できる．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とその処理に必要なデータの転送を行うと性能が向上する．

スレッドマッピングの最適化　ブロック中のスレッドは32スレッド毎にワープにまとめられているので，ブロックの大きさは32の倍数にするのが良い．CUDAのスレッド/ブロックはGPUのコア/SMと密接に関係しており，総スレッド数が同じ場合でもスレッド数/ブロック数が異なると実行時間が大きく異なることがある．これは，CUDAではブロックをGPUのSMに，スレッドをSM内のコアにスケジューリングしながら割り当てているので，スレッド数を多く/ブロック数を少なくした場合はSM間の

[1]ハード的には64KB以上あるが，Kepler世代のGPUではシェアードメモリとL1キャッシュが1つのオンチップメモリを共有しており，ユーザはその比率を48KB:16KB, 32KB:32KB，16KB:48KBからしか選ぶことができない．また，最新のMaxwell世代のGPUではシェアードメモリとL1キャッシュが分離され容量も増えたが，ブロックあたりの容量は48KBのままである．

負荷がうまく分散されず，スレッド数を少なく/ブロック数を多くした場合はコア間の負荷がうまく分散されないためである．そのためスレッド数/ブロック数を調整するのが望ましい．

スレッドはビルトイン変数でアクセスするデータにマッピングされるが，一方で同一ブロック内のスレッドは同一SMに，threadIdx.xの値が連続しているスレッドが同一ワープにマッピングされる．このため，ビルトイン変数を使って配列変数の要素をアクセスする際に，同一ワープ内のスレッドが近接する要素をアクセスするようにマッピングされていれば，コアレスアクセスとなる．また，同一ブロック内のスレッド間で配列の同一要素にアクセスするか，あるいはブロック内全スレッドのアクセス範囲がシェアードメモリの容量以下であるかにより，この配列を明示キャッシュできるか否かが変わる．

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．

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．

プログラム中の配列アクセスを解析する手法は多数研究されている [12][13] ものの，実行時まで値の決まらない変数の影響などで一般に高精度の結果を得ることは難しい．しかし GPU ではその構造上，単純な配列アクセスになることが多い．非効率な分岐のダイバージェンスが生じるため `if/while` の使用は控えることが望ましいし，規則的なアクセスを行う方がコアレス化の可能性が高くなる．また，不規則的なアクセスが発生したとしてもそれに対して最適化を行うことが難しい場合が多い．そのため，今回提案するような簡単な解析でも十分な結果が得られると考えている．

4 MESI-CUDA

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 4.4 に図 2.2 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 4.4：3 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 2.1 の GPU アーキテクチャをそのままプログ

```
 1 #define N 1024
 2 #define BX 128
 3 __global__int ga[N][N], gb[N][N], gc[N][N];
 4 __global__
   void transpose(int a[][N], int b[][N], int c[][N]){
 5   int k , tmp = 0;
 6   int row = blockDim.y*blockIdx.y+threadIdx.y;
 7   int col = blockDim.x*blockIdx.x+threadIdx.x;
 8   for(k = 0 ; k < N ; k++){
 9    tmp += a[row][k] * b[k][col];
10   }
11   c[row][col] = tmp;
12 }
13 void init_array(int d[N][N]){...}
14 void output_array(int d[N][N]){...}
15 int main(){
16   init_array(a);
17   init_array(b);
18   transpose<<<dimGrid, BX>>>
         ((int(*)[N]))da,(int(*)[N]))db,(int(*)[N]))dc);
19   output_array(gc);
20 }
```

図 4.4: CUDA コードと等価な MESI-CUDA コード

ラミングモデルとして用いる．これに対し，MESI-CUDA では図 4.5 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 4.4 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．

4.2 現在の処理系の問題点

MESI-CUDA は，低レベルな記述をフレームワークで隠蔽するため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

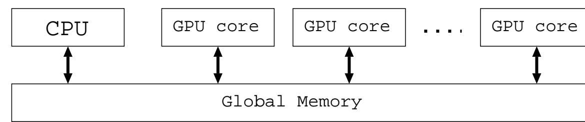

図 4.5: MESI-CUDA のプログラミングモデル

可能である．そのため，ユーザがメモリ階層の有効活用を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．しかし，現在の MESI-CUDA が生成する CUDA コードは，十分に最適化されておらず，手動で最適化された CUDA コードと比較すると実行時間が長くなることがある．

5　自動最適化手法

前述の問題を解決するために以下の自動最適化手法を提案している．

シェアードメモリの利用　CUDA ではシェアードメモリを使用することでプログラムの最適化が可能であるが，MESI-CUDA ではメモリ階層を簡潔にしているため処理系がシェアードメモリを使用する CUDA コードを自動生成する [14].

前述したようにシェアードメモリはブロック毎に存在し容量が非常に小さいため，ブロック内のスレッドが使用する部分のみを格納する．そこで，必要な領域を決定しシェアードメモリへ格納できるか判別するためにはブロック毎のアクセス範囲を解析する必要がある．また，シェアードメモリを効率的に用いるために使用頻度の高いデータを優先的に格納する．そこで，配列の格納優先順位を決定するためにはブロック毎のアクセス回数を解析する必要がある．

マルチ GPU の利用　現在の MESI-CUDA はマルチ GPU 環境に対応していない．そこで，MESI-CUDA が複数の GPU へ自動的に処理を振り分ける負荷分散を行う CUDA コードを自動生成する [15]．各 GPU へ処理を振り分けることで，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ではブロック間の同期手段が無くブロック単位でGPUのSMにスケジューリングすることから，同一のカーネル実行におけるブロック間の処理の依存は無くそれぞれの処理は独立していると考えられる．そこで処理の振り分けをブロック単位で行う．各GPUで必要なデータ領域を転送するためブロック毎のアクセス範囲を解析する必要がある．この結果を元に各GPUに振り分ける処理の量やそれに必要なデータ量を決定する．

スレッドマッピングの最適化　CUDAでは起動する総スレッド数を3次元のブロックと3次元のグリッドで指定する．また，現在のMESI-CUDAもCUDAと同様の記述方法を採用している．しかし，ブロックとグリッドの最適な組み合わせはGPUのアーキテクチャの知識が必要である．そこで，新たなスレッドマッピング手法を採用する[16]．この手法では論理マッピングを用いたスレッド生成構文を導入し，ユーザによる論理マッピングをコンパイル時に処理系が物理マッピングに変換する．これにより，ユーザは任意の次元のスレッドを使用することができ，プログラムの記述も簡略化できる．また，メモリアクセスの最適化の効果を高めるように処理系が物理マッピングを決定することもできる．

これを実現するために，各スレッドのアクセス範囲を求め，同一もしくは近接要素をアクセスするスレッドが同一SM上や連番になるようにマッピングする．

6　解析手法

6.1　概要

前述した自動最適化を実現するためには，ブロック/スレッドによる配列のアクセス範囲/アクセス回数といった情報が必要である．これらの情報を取得するためには，ユーザの記述したコードを静的解析する必要がある．そこで，MESI-CUDA上に配列インデックスの静的解析手法を提案する．

6.2 解析

各カーネル関数やカーネル関数から呼ばれる関数を解析し，各関数内でアクセスされる各変数のアクセス回数を求める．また，配列の添え字式を解析し，アクセスされる範囲を取得する．これらの関数は以下の前提条件を満たすものとする．

1. ループは定数回ループであり，ループ回数はコンパイル時には判明している．

2. 配列の添え字式は，すべてのループ変数およびビルトインのインデックス変数 (`threadIdx.x` や `blockIdx.x`) で表される一次式である．例えば，`a[i*N+j]` や `a[threadIdx.x/N+i]` は対象であるが，`a[i*j]` や `a[threadIdx.x/i]` は対象としない．

逐次プログラムではこれらの条件を満たすことが難しいが，CUDA プログラムでは満たすことが多いと思われる．非効率な分岐のダイバージェンスを防ぐため `if`/`while` 文の使用は避ける傾向がある．スレッド間のアクセス競合を防ぎ，静的に負荷を分散させるようにデータ分割が行われるため，添え字式は一般に単純な線形式であることが多い．もし，条件を満たさなかった変数は，最適化の対象としない．一部の変数だけが対象とならなった場合は，残りの変数で最適化が可能な時とそうでない時がある．例えば，シェアードメモリを利用する場合は，他の変数を格納すればよく，マルチ GPU を利用する場合は，その変数が読み込み専用ならば効率は悪いがその変数は全ての領域を転送すればよい．マルチ GPU を利用する場合において，書き込みが発生する変数が解析対象にできなかったならば，GPU 間の書き込みが競合する可能性があり，結果のマージができないので最適化できない．対象となる変数が無かった場合は，提案手法が適用されないだけであり，コンパイルや実行自体は正しく行うことができる．また，条件文がある場合は，分岐する確率は均等であるとみなす．そのため，`if`/`else if`/`else` ブロックでのアクセス回数は平均値とし，アクセス範囲は各ブロックでのアクセス範囲の和とする．

変数 v のアクセス回数を $access(v)$ とする．条件 1 により，変数のコード上の出現に対するアクセス回数は，その出現を含む全てのループのループ回数の積から求めることができる．そして，変数 v の各出現毎のアクセス回数の和により v のカーネル関数でのアクセス回数が求められる．また，条件 2 より添え字式はループ変数やインデックス変数に対して単調

減少/増加することから，配列のアクセスされる範囲は，ループ変数の最小値と最大値から添え字式を計算して求めることができる [2]．

グリッドとブロックのサイズがそれぞれS_g，S_bであるカーネル関数fでm次元の配列変数vがアクセスされるとき，ブロックb_p $(p=0,\ldots,S_g-1)$内のスレッド$t_{p,q}$ $(q=0,\ldots,S_b-1)$でアクセスされるvの要素の範囲は，以下のように求めることができる．

fのコード中におけるvの出現を$v^1,\ldots,v^k$，v^rにおける第s次元の添え字式を$e_s(v^r)$とする．$e_s(v^r)$の値をループ変数の最小値/最大値の組み合わせで計算する．条件2より，添え字式の計算された最小値/最大値は，$e_s(v^r)$の最小値$\mathtt{min}(e_s(v^r))$/最大値$\mathtt{max}(e_s(v^r))$となる．第s次元のインデックス式の範囲を$R_s(e_s(v^r))=[\mathtt{min}(e_s(v^r)),\mathtt{max}(e_s(v^r))]$とする．$v^r$のアクセス範囲は$m$次元の範囲で以下の式で表す．

$$R(v^r) \;=\; R_1(e_1(v^r))\times\ldots R_m(e_m(v^r))$$

変数vのスレッド$t_{p,q}$およびブロックb_pにおけるアクセス範囲は，以下の式で求めることができる．

$$\begin{aligned} R(v,t_{p,q}) &= \bigcup_{r=1}^{k} R(v^r,t_{p,q}) \\ R(v,b_p) &= \bigcup_{q=0}^{S_b-1} R(v,t_{p,q}) \end{aligned}$$

2つ範囲の和$R'\cup R''$は，範囲R', R''を含む最小の範囲とする．

6.2.1 シェアードメモリおよびマルチGPU最適化のための解析

全てのq，rにおける$e_s(v^r)$の最小値/最大値をそれぞれ$\mathtt{emin}(v,b_p,s)$と$\mathtt{emax}(v,b_p,s)$とすると，$R(v,b_p)=R_1(v,b_p)\times\ldots R_m(v,b_p)$，$R_s(v,b_p)=$ $[\mathtt{emin}(v,b_p,s)$，$\mathtt{emax}(v,b_p,s)]$となり，vのアクセスされる要素数とメモリ容量は以下の式で求めることができる．

$$\begin{aligned} size(R_s(v,b_p)) &= \mathtt{emax}(v,b_p,s)-\mathtt{emin}(v,b_p,s)+1 \\ size(R(v,b_p)) &= size(R_1(v,b_p))\times\ldots size(R_m(v,b_p)) \\ byte(R(v,b_p)) &= size(R(v,b_p))\times\mathtt{sizeof(}\textit{\texttt{type of}}\ v\mathtt{)} \end{aligned}$$

[2]modulo 演算子が使用される場合はこの限りではないが，求めることはできる．$a\%M(a$，Mそれぞれ整数とする)とすると，$\lceil min(a)/M\rceil$と$\lceil max(a)/M\rceil$が異なる場合は．最小値と最大値はそれぞれ0，$M-1$となる．等しい場合は，最小値と最大値は$min(a)$，$max(a)$のいずれかとなる．

変数 v のバイトあたりの平均アクセス回数を $\overline{access}(v)$ とすると以下の式で求めることができる．

$$\overline{access}(v_t) \quad = \quad access(v_t)/byte(R(v_t, b_p))$$

6.2.2　スレッドマッピング最適化のための解析

全ての $s(s = 0, \ldots, n^v - 1)$ において $e_s(v^r)$ 内である次元 q の論理スレッドインデックスが使用されていない場合，q 次元を除く論理スレッドインデックスが同じ全てのスレッドは同一要素をアクセスしていると考えられる．また，全ての論理スレッドインデックスが使用されていても，$e_{n^v-1}(v^r)$ で q 次元の論理スレッドインデックスのみが使用されている場合，q 次元方向で隣接するスレッドは隣接した要素をアクセスすると考えられる．

6.3　適用例

図 4.4 のプログラムに対する解析の適用例を示す．図 4.4 のカーネル関数では 3 つの配列 a，b，c がインデックス式 row，col，k でアクセスされている．各スレッド ID とループ変数の範囲はカーネル関数起動時の引数や `for` 文の条件式から取得でき，各インデックス式の範囲を求めることができる．図 4.4 での結果を表 6.1 に示す．

表 6.1: 式の範囲

式	最小値	最大値
`threadIdx.x`	0	127
`threadIdx.y`	0	0
k	0	1023
row	`blockDim.y*blockIdx.y`	`blockDim.y*blockIdx.y`
col	`blockDim.x*blockIdx.x`	`blockDim.x*blockIdx.x`+127

これにより，各配列のアクセス要素数，スレッドのアクセス回数，ブロックのアクセス回数，バイトあたりの平均アクセス回数は表 6.2 のようになる．

シェアードメモリを利用する場合は，バイトあたりの平均アクセス回数より，a，b，c の順に優先的に格納される．

表 6.2: 解析結果

配列	ブロックのアクセス要素数	スレッドのアクセス回数	ブロックのアクセス回数	バイトあたりの平均アクセス回数
a	1024	1024	1024*128	1024*128/(1024*sizeof(int))
b	1024*128	1024	1024*128	1024*128/(1024*128*sizeof(int))
c	128	1	1*128	1*128/(128*sizeof(int))

```
          ⋮
 6   int row = lThreadIdx.y;
 7   int col = lThreadIdx.x;
 8   for(k = 0 ; k < N ; k++){
 9    tmp += a[row][k] * b[k][col];
10   }
11   c[row][col] = tmp;
          ⋮
15 int main(){
          ⋮
18   transpose<[<N, N>]>
         ((int(*)[N]))ga,(int(*)[N]))gb,(int(*)[N]))gc);
          ⋮
20 }
```

図 6.6: 論理スレッドを用いた MESI-CUDA コードの例

マルチ GPU を利用する場合は，ブロックの各配列のアクセス要素数が各 GPU に振り分けるブロックあたりの転送量となり，a は 1024×sizeof(int) である．

図 6.6 に図 4.4 のプログラムを論理スレッドを用いて記述したプログラムを示す．`lThreadIdx.x` や `lThreadIdx.y` はそれぞれ x，y 次元のスレッドのインデックスである．a のインデックス式には `lThreadIdx.x` が使用されていないので，x 次元方向で隣接するスレッドは同じ要素をアクセスすることがわかる．また，c は第 2 次元のインデックス式で `lThreadIdx.x` のみを用いてアクセスされることから，x 次元方向で隣接するスレッドは隣接した要素をアクセスすることがわかる．そこで，x 次元方向のスレッドを優先的に `threadIdx.x` にマッピングすることで，メモリアクセスの効率化が可能となる．

6.4　アクセス範囲解析の改良

上記のアクセス範囲解析では不要な領域が含まれることがある．図 6.7 にその例として二重ループ中での二次元配列のアクセスの様子を示す．上記の解析手法では，図の点線部をアクセス範囲としてしまい，実際のアクセス範囲と異なる。

そこで，より正確なアクセス範囲を解析する手法も提案している．単一のスレッドのアクセス範囲の大きさを R1，二つのスレッドの重なっているアクセス範囲の大きさを R2 とすると、ブロックのアクセス範囲の大きさは R1*blockDim.x-R2*(blockDim.x-1) で求めることができる．図 6.7 の場合、添字を座標とみなすことで，R1 と R2 の面積はベクトルの外積で求めることができる．各頂点の座標は配列の添字中のループ変数とスレッドの ID を代入し求める．ネストしたループ変数 i,j のループ文で、ループ範囲はそれぞれ $[st1,en1]$，$[st2,en2]$ とする．$st1$，$en1$，$st2$，$en2$ は任意の定数式とする．このループ文中で，ある配列要素 $a[ix][iy]$ をアクセスする場合を考える．ix，iy は次式で表せるものとする．

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

$$p * i + q * j + r * \texttt{threadIdx.x} + s$$

ここで a，b，c，d，p，q，r，s は任意の定数式とする．fig .6.7 中の各点の座標は Table 6.3 のようになる．これらを用いてブロックのアクセス範囲の大きさを求める．

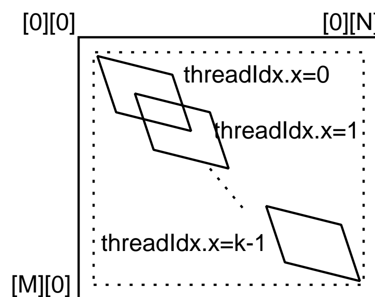

図 6.7: 二次元配列アクセスの様子

表 6.3: 各点の座標

点	座標
A	$(a*st1+b*st2+d,\ p*st1+q*st2+s)$
B	$(a*(en1-1)+b*st2+d,\ p*(en1-1)+q*st2+s)$
C	$(a*(en1-1)+b*(en2-1)+d,\ p*(en1-1)+q*(en2-1)+s)$
D	$(a*st1+b*(en2-1)+d,\ p*st1+q*(en2-1)+s)$
E	$(a*st1+b*st2+c+d,\ p*st1+q*st2+r+s)$

7　評価

提案した解析手法の有用性を示すために，本手法を用いた自動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 Tesla C2075，GeForce GTX 680，GeForce GTX TITAN，Tesla K20，GeForce GTX 980 を搭載した計算機を使用した．マルチ GPU 環境は同じ Kepler 世代である GTX 680 と GTX TITAN を搭載した計算機を用いた．評価は行列積の計算とヤコビ法による連立一次方程式の反復解法，FDTD 法による電磁場解析，2 次元画像の平滑化を行うプログラムをいくつかのデータサイズとブロックサイズの場合で行った．

シェアードメモリ最適化　シェアードメモリを利用する最適化の結果を表 7.4 に示す．

行列積のプログラムでは実行時間が大幅に短縮できていることがわかる．これは本解析手法によって適切な配列のアクセス解析がされており，これによって前述したシェアードメモリを効果的に用いることが可能になりメモリアクセスのレイテンシが短縮されたためである．しかし，残りのプログラムではブロックサイズや GPU によっては，あまり効果が無かったり，かえって実行時間が長くなっている．これはシェアードメモリへ格納する部分の平均アクセス回数が少なかったためだと思われる．そのため，今後の課題として平均アクセス回数によるこの最適化の適用の有無を決める必要がある．

マルチ GPU 最適化　マルチ GPU を利用する最適化の結果を表 7.5 に示す．

各プログラムの各データサイズで実行時間が短縮できていることがわかる．これは本解析手法によって適切な配列のアクセス解析がされてお

表 7.4: シェアードメモリ最適化の実行時間比較 (秒)

データサイズ	ブロックサイズ	C2075 非最適化	C2075 最適化	GTX 680 非最適化	GTX 680 最適化	GTX TITAN 非最適化	GTX TITAN 最適化	K20 非最適化	K20 最適化	GTX 980 非最適化	GTX 980 最適化
行列積											
8192	64	50.243	5.926	33.237	4.518	22.136	3.558	32.686	5.127	21.302	1.628
	256	26.110	5.927	17.889	4.519	13.076	3.560	19.429	5.126	6.866	1.625
	1024	26.215	5.927	15.036	4.518	10.628	3.560	15.820	5.127	6.931	1.627
ヤコビ											
1024	64	3.230	3.384	1.863	2.014	1.305	1.463	1.828	2.120	0.860	0.763
	256	1.859	1.922	1.190	1.315	0.851	0.999	1.163	1.434	0.598	0.564
	1024	2.020	2.483	1.334	1.381	0.777	0.824	1.078	1.160	0.555	0.560
FDTD											
10240	64	79.803	82.959	56.747	55.606	37.668	37.513	55.395	54.881	41.995	38.930
	256	50.544	50.297	38.885	38.618	25.351	25.277	39.347	39.095	24.359	24.137
	1024	58.848	57.861	36.005	31.834	22.776	21.219	35.160	33.535	23.571	23.502

り，これによってマルチ GPU への処理の振り分けが可能になり各 GPU の負荷が分散されたためである.

表 7.5: マルチ GPU 最適化の実行時間比較 (秒)

データサイズ	GTX 680	GTX TITAN	マルチ GPU
行列積			
512	0.0420	0.0297	0.0201
1024	0.326	0.225	0.147
2048	2.580	1.752	1.090
平滑化			
512	0.0398	0.0392	0.0252
1024	0.150	0.148	0.829
2048	0.586	0.575	0.299

スレッドマッピング最適化　スレッドマッピングの最適化の結果を表 7.6 に示す.

各プログラムの各データ/ブロックサイズで実行時間が短縮できていることがわかる. これは本解析手法によって適切な配列のアクセス解析がされており，これによって最適なスレッドのマッピングが可能になりメモリアクセスが最適化されたためである.

表 7.6: スレッドマッピング最適化の実行時間比較 (秒)

データサイズ	ブロックサイズ	C2075		GTX 680		GTX TITAN		K20		GTX 980	
		非最適化	最適化	非最適化	最適化	非最適化	最適化	非最適化	最適化	非最適化	最適化
行列積											
8192	64	384.204	5.926	294.886	4.518	139.456	3.558	206.064	5.127	151.825	1.628
	256	953.116	5.927	743.408	4.519	167.966	3.560	514.487	5.126	163.001	1.625
	1024	1316.029	5.927	1557.996	4.518	358.794	3.560	650.521	5.127	267.408	1.627
ヤコビ											
1024	64	11.542	3.384	11.077	2.014	7.443	1.463	11.041	2.120	3.280	0.763
	256	16.067	1.922	11.090	1.315	7.480	0.999	11.093	1.434	3.255	0.564
	1024	20.133	2.483	11.321	1.381	7.509	0.824	11.130	1.160	3.355	0.560
FDTD											
10240	64	495.268	82.959	321.354	55.606	213.703	37.513	317.766	54.881	216.635	38.930
	256	572.327	50.297	326.701	38.618	227.893	25.277	335.470	39.095	205.979	24.137
	1024	836.007	57.861	422.703	31.834	240.496	21.219	352.291	33.535	211.091	23.502

8　終わりに

本研究では配列アクセス時の添え字式を静的解析し，配列要素のアクセス回数やアクセス範囲を求める手法を提案した．性能評価の結果，本手法を用いることで自動最適化に必要な情報を取得し，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，より精度の高い方法を導入する必要がある．また，他の自動最適化を実現するために，それに対応した解析方法を考案する必要がある．

謝辞

本研究を行うにあたり，御指導，御助言頂きました大野和彦講師，並びに多くの助言を頂きました山田俊行講師に深く感謝致します．また，様々な局面にてお世話になりました研究室の皆様にも心より感謝いたします．

参考文献

[1] *GPGPU.org: General-Purpose computation on Graphics Processing Units*, http://www.gpgpu.org/, (2013.06.22).

[2] *NVIDIA Developer CUDA Zone*, http://developer.nvidia.com/category/zone/cuda-zone, (2013.04.27).

[3] *OpenCL - The open standard for parallel programming of heterogeneous systems*, 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*, http://www.openacc-standard.org/,(2013.06.7).

[11] Michael Wolfe: *Scaling OpenACC Across Multiple GPUs*,GPU Technology Conference 2014,(2014).

[12] Paul Feautrier: *Dataflow Analysis of Array and Scalar References*,International Journal of Parallel Programming,(1991).

[13] Béatrice Creusillet and Francois Irigoin: *Interprocedural array region analyses*,Languages and Compilers for Parallel Computing,46-60,(1996).

[14] Tomoharu Kamiya, Takanori Maruyama and Kazuhiko Ohno: *Compiler-Level Explicit Cache for a GPGPU Programming Framework*,the 2014 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'14),(2014).

[15] 丸山 剛寛, 田中 宏明, 水谷 洋輔, 神谷 智晴 and 大野 和彦: *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*, 情処研報 2014-HPC-145, pp. 1-8,(2014).

[16] Kazuhiko Ohno, Tomoharu Kamiya, Takanori Maruyama and Masaki Matsumoto: *Automatic Optimization of Thread Mapping for a GPGPU Programming Framework*,The Second International Symposium on Computing and Networking CANDAR'14,(2014).