修士論文

題目

GPGPUにおける
メモリアクセスの自動最適化手法

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

神谷　智晴(413M509)

内容梗概

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，シェアードメモリを用いるコードを自動生成する機構やスレッドマッピングを処理系が行う機構を MESI-CUDA 上に開発している．本研究では MESI-CUDA 処理系が生成するコードのメモリアクセス効率を向上させるため 2 つの手法を提案する．

カーネル関数内のデータアクセスパターンを自動で変更する手法では，グローバルメモリアクセスを最適化するためコード変換を行う．SIMD 型並列実行の単位であるワープ内のスレッド群が近傍アドレスを対象とする時，効率的なコアレッシングアクセスが可能である．そこで，多次元配列に対してスレッド内外の配列アクセス方向を入れ替える．この際，必要に応じて配列データの転置を行うことでよりコアレッシングアクセスの可能性を高めることができる．

シェアードメモリを用いるコードを自動生成する手法では，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う．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る．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することができる．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．

三重大学大学院　工学研究科

目 次

図 目 次

表 目 次

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][2] への関心が高まっている．また，CUDA[3] やOpenCL [4] 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）[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, 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機構 [7][8] や自動スレッドマッピング機構 [9] を開発している. 本研究では MESI-CUDA 処理系が生成するコードのメモリアクセス効率を向上させるため 2 つの手法を提案する．

カーネル関数内のデータアクセスパターンを自動で変更する手法では，グローバルメモリアクセスを最適化するためコード変換を行う．SIMD 型並列実行の単位であるワープ内のスレッド群が近傍アドレスを対象とする時，効率的なコアレッシングアクセスが可能である．そこで，多次元配列に対してスレッド内外の配列アクセス方向を入れ替える．この際，必要に応じて配列データの転置を行うことでよりコアレッシングアクセスの可能性を高めることができる．

シェアードメモリを用いるコードを自動生成する手法では，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う．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る．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することができる．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，自動スレッドマッピング機構について説明する．５章ではカーネル関数内のデータアクセスパターンを自動で変更する手法を示す．６章で，シェアードメモリを用いるコードを自動生成する手法について示す．７章では2つの手法それぞれの有無による 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．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．

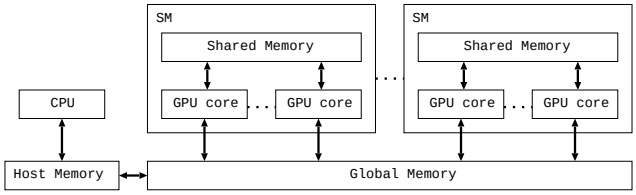

図 2.1: GPU のアーキテクチャモデル

2.2　CUDA

CUDA[10][11] 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 2.2 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 2.2：31 行）．このときに作成するスレッド数を指定する．

```
 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 2.2: 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 2.2：28-29 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 2.2：33 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 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．

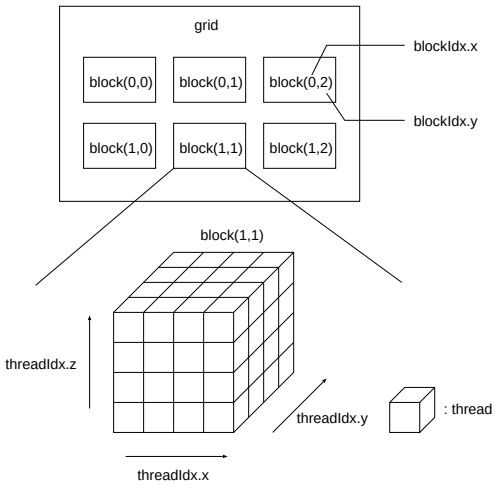

図 2.3: グリッド-ブロック-スレッド

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．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 2.2：21-23, 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 2.1）．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 [12][13]．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 [14] 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．

最新の CUDA6 とケプラー GPU は MESI-CUDA の仮想共有変数と同様な働きをする $UnifiedMemory$ を提供している．大きな相違点として，仮想共有変数はソースコードのトランスレータとして実装しているのに対し，CUDA6 の $UnifiedMemory$ はハードウェア/ドライバレベルで実装されている点である．我々の強みは本論文で提案したスレッドマッピングの最適化やメモリアクセスの最適化を，静的解析を利用してコンパイル時に行えることである．$UnifiedMemory$ の主な目的は容易に GPU プログラミングを可能にすることであり，高い性能を出すためには低レベルな API の使用が推奨されている．これとは対照的に，MESI-CUDA の目標はコンパイラの下で最適化を隠蔽することである．

OpenACC [15] と OpenMP-to-CUDA translation [16] は低レベルの仕様を用いない異なる GPGPU のアプローチである．幾つかの並列化の疑似命令を持った逐次プログラムが GPU 上で実行可能な並列プログラムへコンパイルされる．従ってコンパイラが自動でスレッド化されたコードの生成とデータや物理リソースに対するスレッドマッピングを行う．しかし，現在の OpenACC の性能は手動最適化された CUDA コードと比較すると負けてしまう．マッピング制御のためのプラグマとして $gang$ や $vector$ が提供されているが，手動最適化は困難であり CUDA プログラミングの様な低レベルな知識やハードウェア依存の知識を必要とする．我々の機構は OpenACC のスレッドマッピングの最適化にも適用できる可能性がある．

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 4.4：4 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 2.1 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 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 自動スレッドマッピング機構

我々は MESI-CUDA に自動スレッドマッピング機構を開発している．これによりユーザから低レベルな GPU の特徴を隠蔽し，デバイス依存の最適化をコンパイラが行うことでより柔軟な自動最適化が可能となる．

CUDA との互換性を保つため我々は<[<D_0, ..., D_{m-1}>]> という新しいカーネル起動の記法を導入した．この記法は $D_0 \times \ldots \times D_{m-1}$ の様に単一の任意次元のスレッドグリッドを表している．新しいカーネル起動と

```
 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 4.4: CUDA コードと等価な MESI-CUDA コード

表 4.1: 論理マッピングのためのビルトイン変数

`lGrid.dim[`i`]`, $(i = 0, \ldots, m-1)$	grid size (# of threads)
`lThread.Idx[`i`]`, $(i = 0, \ldots, m-1)$	thread index (in the grid)
`lGridDim.x`, ... `lGridDim.z`	alias to `lGrid.dim[`i`]` and
`lThreadIdx.x`, ... `lThreadIdx.z`	`lThread.Idx[`i`]` $(i = 0, 1, 2)$

グリッドをそれぞれ論理カーネル起動と論理グリッドと呼ぶ．また，表 4.1 に示す新しいビルトイン変数を導入する．

論理グリッドは次元とサイズに制限はない．それらはアプリケーションのデータ構造に合わせてマッピングされる．この様にコードは物理リソースから独立している．

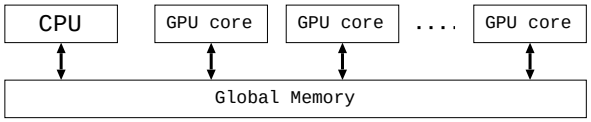

図 4.5: MESI-CUDA のプログラミングモデル

4.2.1 基本方針

我々は同じまたは近隣要素をアクセスするスレッドのグループを得るために配列のインデックス式を静的解析する．以下に用語の定義をする．

- ターゲット変数

  カーネル関数起動時に $D = D_0 \times \ldots \times D_{m-1}$ サイズの m 次元の論理グリッドが指定され，k 重ループの中で n 次元配列へのアクセスが発生するとする．各インデックス式 $e_0, \ldots, e_{n-1}$ において，全てのビルトイン論理インデックス変数 `lThread.Idx[0]`, ..., `lThread.Idx[m-1]` と全てのループ変数 i_0, ..., i_{k-1} をターゲット変数と見なす。

- インデックス式の正規形

  インデックス式 e の正規形を $N(e)$ として記述し以下の式で表す．

$$\begin{aligned} N(e) &= (C^I_0 \times \texttt{lThread.Idx[0]} + \ldots \\ &+ C^I_{m-1} \times \texttt{lThread.Idx[m-1]}) \\ &+ (C^L_0 \times i_0 + \ldots + C^L_{k-1} \times i_{k-1}) \\ &+ C \end{aligned}$$

  正規形はターゲット変数の項が一次である多項式でなければならない．C^I_p と C^L_q は各項の係数であり，C と共にループ内不変である．

カーネル関数は下記の条件を満たすものとする．

1. 全てのループ回数は固定でコンパイル時に分かっている．
2. 全ての配列インデックス式は正規形に変形できる．

現状，不定回ループと非線形なインデックス式は対応していない．しかし，多くのカーネル関数は解析できる．なぜなら，GPU プログラムでは不規則なアクセスパターンは非効率的で避けられる傾向にあるからである．条件を満たさない式は除外するため最適なマッピングは行わないが，コード生成は可能である．

各配列変数の登場時のインデックス式を静的解析し，スレッド間と配列のインデックス間の関係を取得する．また，インデックス式の範囲解析を行い，スレッドが各配列をアクセスする範囲を取得する．その結果を用いてメモリアクセスの最適化が適用できる様にスレッドをグループ化する．基本方針を以下に示す．

1. 両スレッド t_p, t_q について対応する各インデックス式 $e_0, \ldots, e_{m-1}$ が同じ値を持つとき同じ配列要素をアクセスする．t_p, t_q を同じワープに割り当てるとコアレッシングアクセスになる．

2. $e_0, \ldots, e_{m-2}$ が同じ値を持ち，e_{m-1} の値の差が 1 ならば t_p, t_q を同じワープに割り当てるとコアレッシングアクセスになる可能性が高い．なぜなら隣接アドレスをアクセスするからである．

全てのスレッドの組み合わせでこの方針を評価すると膨大な時間がかかる．加えて，もし論理物理スレッドが非線形の関係であるなら効率的なコードを生成することは困難である．それゆえ m 次元の論理グリッドを s 次元のタイルに分割し，各タイルを CUDA のスレッドブロックへ割り当てる．m 次元から優先する s 次元を決定するため，各次元は分割優先値を与えられ $P_0, \ldots, P_{m-1}$ と表す．我々は m 次元の各次元方向で隣接したスレッドのみを評価し，もし方針の条件を満たしていれば分割優先値を加算する．

4.2.2 静的解析

インデックス式の正規形への変換　カーネル関数内の各インデックス式を正規形に変換する．式中の非ターゲット変数はインデックス式の登場時に置き換え可能ならば置換する．

同じ要素へのアクセス検出　同じ要素へのアクセス回数を求めるため，各配列変数の出現に対してインデックス式を評価する．もし $e_0, \ldots, e_{n-1}$ 全てが `lThread.Idx[`r`]` の項を持たないならば r 次元を除く論理インデックス

が全て一致するスレッドは同じ要素をアクセスし，コアレッシングアクセスとなる．その時，$L \times 32$ を P_r に加算する．L はループ回数 $L_0, \ldots, L_{k-1}$ の積である．

隣接要素へのアクセス検出　たとえ $e_0, \ldots, e_{n-1}$ 中に全ての `lThread.Idx[0]`, ..., `lThread.Idx[`m`-1]` が存在したとしても，論理グリッド上で隣接するスレッドがメモリ上の隣接要素をアクセスするならば分割優先値を与えるべきである．そのため最右端のインデックス式 e_{m-1} を評価する．もし e_{m-1} が `lThread.Idx[`r`]` のみであり C_r^I が定数の時，$L \times \lceil 128/\{C_r^I \times$ `sizeof(`*type of* v_p *element*`)`$\}\rceil$ の値を P_r に加算する．この式はコアレッシングアクセスの効果を評価するものである．

4.2.3　スレッドマッピング

各スレッドの起動は実行時のグリッドで指定され，コアレッシングアクセスや明示キャッシュの可否は他の呼び出し時の構成に影響されない．この様に各スレッドのマッピングはカーネル関数ごとに独立して決定することが可能である．

CUDA ブロックへの論理次元マッピング　$P_{r_0} \geq \ldots \geq P_{r_{m-1}}$ となる様に $r_0, \ldots, r_{m-1}$ を並び替える．このとき `lThread.Idx[`r_0`]` を除く同じ論理インデックスを持つスレッドはコアレスドアクセスになる可能性が最も高い．よって r_0 方向に並ぶスレッドは 1 つの CUDA ブロックにグループ化されるべきである．これを r_0 結合と呼ぶ．

スレッドブロックのサイズ決定　ブロックサイズ B を 64, 128, 256, 512, 1024 の中から選ぶ．大きい値の方がスケジュールできるワープの数が多くなる．しかし，より多くの明示キャッシュ領域やレジスタを使用する可能性があり，シェアードメモリの容量を超えてしまったり同時実行可能なブロック数が減少することがある．

論理次元のマッピング　上記のブロックサイズ B を使用し，内部ブロックのマッピングを決定する．もし D_{r_0} が B より大きいとき，r_0 方向のスレッドは複数のブロックに分割されなければならない．CUDA のグリッドブロックサイズはそれぞれ $D_{r_0}/B \times D/D_{r_0}$ と $B \times 1 \times 1$ のように構成する．論理インデックス `lThread.Idx[`r_0`]` は `blockDim.x*blockIdx.x`

+ threadIdx.x に変換される．他の全ての次元は 1 次元に変換し，論理インデックスは blockIdx.y を使用しブロックインデックスに変換する．これを r_0 結合マッピングと呼ぶ．もし D_{r_0} や $D_{r_0} \times D_{r_1}$ などが B より小さい場合，次元は threadIdx.x, threadIdx.y, threadIdx.z にマッピングされる．グリッドとブロックのサイズや残りの論理インデックスは上記と同様に決定する．

もし $P_{r_0} \simeq P_{r_1} \gg P_{r_2}$ ならばスレッドは 2 つの次元を使用して分割される．このときブロックサイズ B は 64, 256,1024 のどれかでなければならない．CUDA のグリッドとブロックのサイズはそれぞれ $D_{r_0}/b \times D_{r_1}/b \times D/(D_{r_0} \times D_{r_1})$ と $b \times b \times 1$ で決定される．ただし，$b^2 = B$ である．これを $r_0 r_1$ 結合マッピングと呼ぶ．

5　自動スレッドマッピング機構の改良

5.1　概要

自動スレッドマッピング機構によりユーザはデバイス依存のパラメータや低レベルの知識を必要とせず効率的なスレッドマッピングが可能となる．しかし，現状のスレッドマッピング機構では論理インデックスのみに注目しておりループ変数は考慮していない．また，従来手法ではグリッド指定と配列のインデックス式のみ書き換えるため，ユーザがアクセス効率の悪いコードを記述してしまうと最適なマッピングができないことがあった．本手法では静的解析時に行う分割優先値の評価方法を見直し，条件を満たせばコアレッシングアクセスとなるようユーザが記述したコードの一部を書き換える．

5.2　分割優先値の再評価

本手法では 4.2.2 節の**隣接要素へのアクセス検出**で示した分割優先値の評価方法を改める．現状の評価方法では最右端のインデックス式 e_{m-1} が `lThread.Idx[`r`]` の項と定数項のみの時，分割優先値を加算する．この方法では e_{m-1} がループ変数の項と定数項の場合は評価を行わない．しかし，e_{m-1} がループ変数の項と定数項であったとしても e_{m-2} が `lThread.Idx[`r`]` の項と定数項のみの時，e_{m-1} と e_{m-2} を入れ替えることでコアレッシングアクセスになる可能性が高い．

例として，図.5.6 にステンシル計算を行う CUDA プログラムのカーネル関数を示す．5, 6 行目の式に注目する．各配列 $temp$，$power$，$result$ のインデックス式は e_1 がループ変数のみであり，e_0 は `lThread.Idx[`r`]` のみかつ C_r^I が 1 である．この時のアクセスパターンを図 5.7 に示す．太線に囲まれた領域はあるスレッドが 1 回のループイテレーションでアクセスする領域であり，点線で囲まれた領域は論理グリッドの y 軸が隣接するスレッドのアクセスする領域である．斜線部はそれらのスレッドがループ中にアクセスする領域を示している．各スレッドは同時に N 個離れたアドレスをアクセスすることになり，N が大きい場合コアレッシングアクセスにならない．しかし，隣接する各スレッドは縦方向に連続アクセスしており，e_0 と e_1 を入れ替えることでコアレッシングアクセスが可能となる．

そこで隣接要素へのアクセス検出を次のように変更する．

```
1 __global__ void single_iteration1(float result[][N], float temp[][N],
                  float power[][N],float Cap, float Rx, float Ry, float Rz,float step){
2   float delta;
3   int idx = blockDim.x*blockIdx.x+threadIdx.x;
4   for (int c = 0; c < N; c++) {
5     delta = (step / Cap) * (power[idx][c] +
        (temp[idx+1][c] + temp[idx-1][c] - 2.0*temp[idx][c]) / Ry +
        (temp[idx][c+1] + temp[idx][c-1] - 2.0*temp[idx][c]) / Rx +
        (80.0 - temp[idx][c]) / Rz);
6     result[idx][c] =temp[idx][c]+ delta;
7   }
8 }
```

図 5.6: ステンシル計算プログラムのカーネル関数

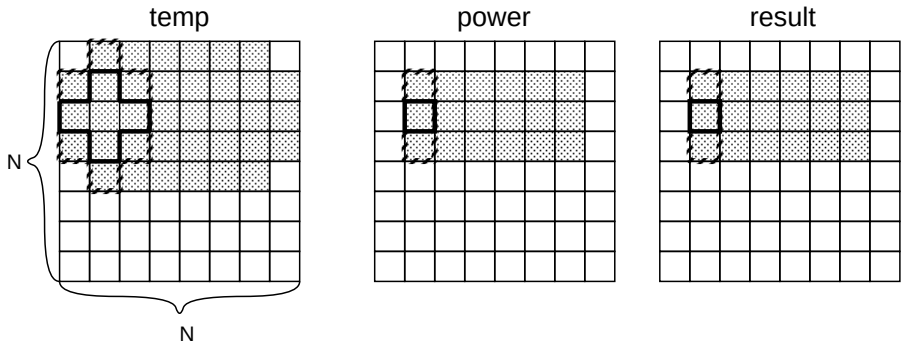

図 5.7: 図.5.6:5, 6 行の配列アクセスパターン

- e_{m-1} が lThread.Idx[r] のみであり C_r^I が定数の時,
  $L \times \lceil 128/\{C_r^I \times$ sizeof(*type of* v_p *element*)$\}\rceil$ の値を P_r に加算する.

- e_{m-1} が i_j でありループ内の依存が無く，e_{m-2} が lThread.Idx[r] の項と定数項のみの時,
  $L \times \lceil 128/\{C_r^L \times$ sizeof(*type of* v_p *element*)$\}\rceil$ の値を P_r に加算し，インデックス式入れ替えのフラグをたてる.

図 5.8 にインデックス式交換後のコードを，図 5.9 にインデックス式交換後の各配列のアクセスパターンをそれぞれ示す．太線と点線，斜線の領域は図.5.7 と同様である．インデックス式の入れ替えを行うことで，各スレッドは互いに隣接するアドレスをアクセスするためコアレッシングアクセスが可能となる．また，i_{k-1} のループ回数と D_r の値が異なるとき

入れ替えを行うと正しいアクセスが行われなくなる．そのためインデックス入れ替えと同時に i_{k-1} のループ回数の値と D_r の値も入れ替える．

```
1 __global__ void single_iteration1(float result[][N], float temp[][N],
                  float power[][N],float Cap, float Rx, float Ry, float Rz,float step){
2   float delta;
3   int idx = blockDim.x*blockIdx.x+threadIdx.x;
4   for (int c = 0; c < N; c++) {
5     delta = (step / Cap) * (power[c][idx] +
        (temp[c][idx+1] + temp[c][idx-1] - 2.0*temp[c][idx]) / Ry +
        (temp[c+1][idx] + temp[c-1][idx] - 2.0*temp[c][idx]) / Rx +
        (80.0 - temp[c][idx]) / Rz);
6     result[c][idx] =temp[c][idx]+ delta;
7   }
8 }
```

図 5.8: アクセスパターン変更後の CUDA コード

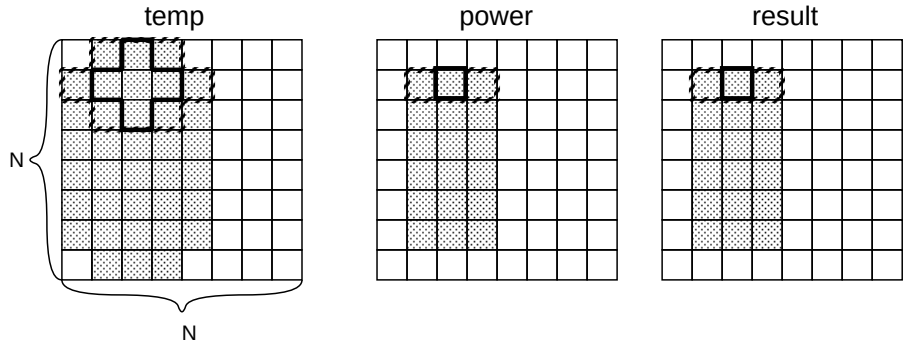

図 5.9: 図.5.8 のアクセスパターン

5.3 データ転置

1つのカーネル関数内で複数の配列が異なる論理グリッド方向にアクセスしている時，配列データを転置しインデックス式を入れ替えることで共にコアレッシングアクセスが可能となる．例として図 5.10 の様なコードを考える．d，e はそれぞれ任意の定数であるとする．配列 a と配列 b はそれぞれ異なる方向にアクセスしている．この場合，スレッドマッピングで x，y 方向どちらにマッピングしても，片方の配列アクセスはコア

レッシングアクセスにできなくなってしまう．配列 a のデータを転置しインデックス式を入れ替えることで a，b 共にコアレッシングアクセス可能となる．図 5.11 にインデックス交換後のコードをを示す．しかし，この様なデータアクセスの場合 2 次元以上のスレッドマッピングを行わないと a，b 共にコアレッシングアクセスとならない．

```
c[threadIdx.y][threadIdx.x]=a[threadIdx.y][d]+b[e][threadIdx.x];
```

図 5.10: 配列のアクセス方向が異なるコード

```
c[threadIdx.y][threadIdx.x]=a[d][threadIdx.y]+b[e][threadIdx.x];
```

図 5.11: 図 5.10 のインデックス入れ替え後のコード

データの転置を行うにあたり考慮しておかなければならないことがある．以下にそれらを示す．

- データ転置時のコスト

  データを転置する方法として CPU を使用する方法と GPU を使用する方法の 2 種類が考えられる．CPU を使用した場合処理に時間がかかり，GPU を使用した場合高速に処理可能だがデバイスメモリの使用容量が転置する配列ごとに 2 倍のデータ量となってしまう．デバイスメモリの使用量に応じてどちらで処理を行うか決定すべきである．

- 転置対象配列の依存関係

  転置はホスト側で対象配列のデータがデバイス側にコピーされる直前に行う．また，対象配列に書き込みが発生した場合は，GPU の処理が終了した後の *readback* 転送直後に転置を行いデータを元の状態に戻す必要がある．

- データ転置による効果の評価

  転置対象の配列データがカーネル関数内で複数回使用されており，それらのインデックス式が異なる場合，転置することによるメモリアクセスの改善，悪化を評価すべきである．加えて，転置時のコストも含めた評価関数を用いてデータ転置を行うかどうかの判断をすることが望ましい．

6　シェアードメモリを用いた自動最適化機構

6.1　概要

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．

以下，提案手法についての解析・コード生成の説明をした後，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．

6.2　解析

今回対象としたループ文を図 6.12 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++){
...}
```

図 6.12: 対象としたループ文

6.3 コード生成の概要

はじめにコード生成までの流れを図 6.13 に示す．解析によ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 6.2 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 6.14 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 6.15 は図 6.14 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

```
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
```

図 6.13: コード生成までの流れ

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

図 6.14: 配列アクセスコードの例

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 6.15 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．

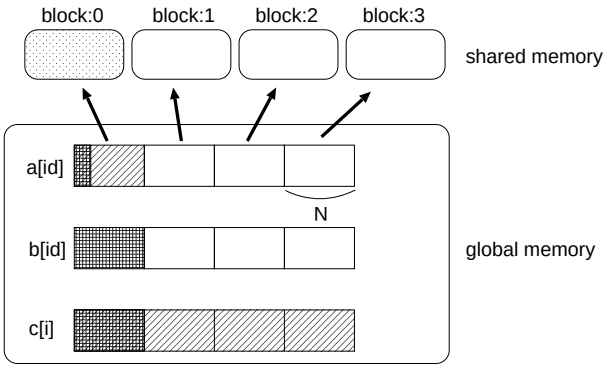

図 6.15: シェアードメモリへ格納する変数の例

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

図 6.16: 境界部分の格納を用いる文

6.4 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 6.16 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 6.16 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 6.16 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 6.19 に示す．境界部分の格納要素数はシェアー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．

6.5 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．

6.6 コード生成

6.3 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 2.2 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 6.17 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

```
 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 }
```

図 6.17: MESI-CUDA で生成した CUDA コード

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 6.17:8 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，6.3 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 6.18：(a)）．

そこで図 6.18：(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 6.19)．格納に用いるコードを図 6.20 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 6.17:9-16 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 6.17:17 行のようにコピーのすぐ後に同期を挿入している．

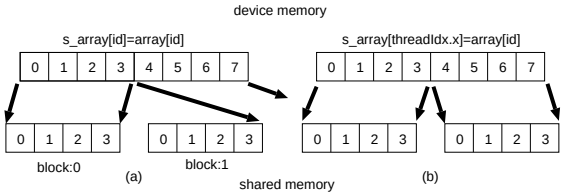

図 6.18: シェアードメモリへのデータコピー

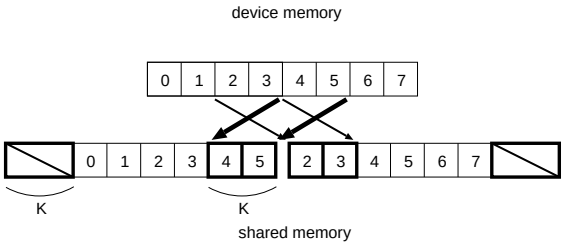

図 6.19: 境界部分の格納

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．

```
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];
```

図 6.20: シェアードメモリへの格納コード

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

図 6.21: データ入れ替えを行う対象のコード

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 6.21 に変換対象となるコードを，図 6.22 に変換後のコードをそれぞれ示す．なお図 6.21 と図 6.22 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 6.22 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．

```
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]
 }
```

図 6.22: 6.21 から変換したコード

式内にループ変数が存在する場合　6.3 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 6.23（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 6.23：(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 6.24 に変換対象となるコードを，図 6.25 に変換後のコードをそれぞれ示す．なお図 6.24 と図 6.25 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 6.25 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 6.25 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．

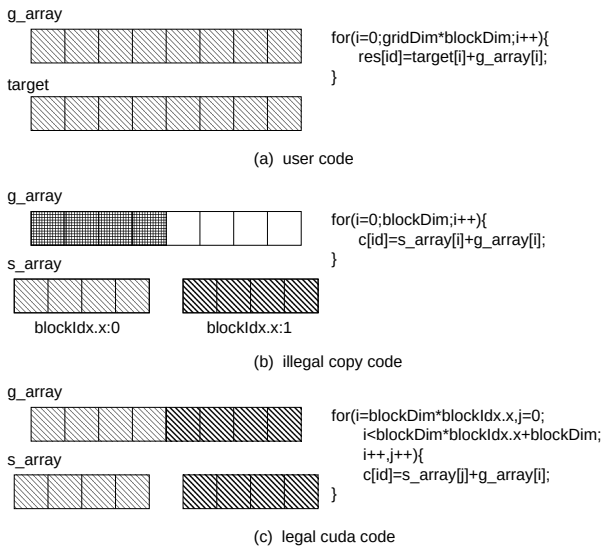

図 6.23: ループ時の配列へのアクセス

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 6.20 に示した代入文の右辺と左辺を交換したものとなる．

6.6.1　提案手法を用いた例

ここでは 6.6 節で示した手法を適用して図 4.4 を図 6.17 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 4.4:10 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 4.4:9-11

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

図 6.24: 従来手法の対象となるコード

```
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];
 }
}
```

図 6.25: 6.22 から変換したコード

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 6.17:8 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 6.17:9-16 行）．ここでは図 6.20 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 6.17:7 行），変数 a を _s_a に変更する．それに伴い，図 6.25 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．

7　評価

7.1　自動スレッドマッピング機構の改良

本手法の有用性を示すために，各マッピングにおける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．評価に使用したプログラムはステンシル計算プログラム ($N = 1024$，ループ数:10) と行列積を求めるプログラム ($N = 8192$) である．評価環境は 2 種類の実行環境

- Xeon E5-1620 3.60GHz，メモリ 16GB，GeForce GTX 980
- Core i7 4820 3.70GHz，メモリ 16GB，Tesla K20

をそれぞれ搭載した計算機を使用した．表 7.2 にステンシル計算の結果を，表 7.3 に行列積の結果をそれぞれ示す．

ステンシル計算のカーネル関数内に出現する配列アクセスは最右端のインデックス式がループ変数のみとなっている．そのため従来手法では正しい解析が行えず，マッピングが最適化されない．本手法を用いることでインデクスの入れ替えを行いメモリアクセスの効率化が可能となり，従来手法と比べて最大 200%の高速化に成功した．

行列積では従来手法でも最適なマッピングである xy 結合を選択することが可能である．しかし，出現配列が異なる方向をアクセスするため全ての配列がコアレッシングアクセスにならない．本手法を用いることでデータを転置しアクセス方向を調整できた．その結果，従来手法と比べて最大 8%の高速化に成功した．

表 7.2: ステンシル計算の実行時間 (秒)

	ステンシル計算			
GPU	マッピング方向	従来手法	提案手法	実行時間比 (%)
Tesla K20	x 結合	70.9	4.05	5.7
	y 結合	19.1	3.70	19.4
GeForce GTX 980	x 結合	27.4	4.27	15.6
	y 結合	5.07	4.26	84.0

表 7.3: 行列積の実行時間 (秒)

GPU	行列積			
	マッピング方向	従来手法	提案手法	実行時間比 (%)
Tesla K20	x 結合	31.4	31.4	100.0
	y 結合	513.9	100.0	19.5
	xy 結合	19.4	17.9	92.3
GeForce GTX 980	x 結合	12.9	13.3	103.1
	y 結合	162.0	84.6	52.2
	xy 結合	6.92	6.52	94.2

7.2 シェアードメモリを用いた自動最適化機構

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 7.4，表 7.5，表 7.6 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 7.4: TeslaC2050 での実行時間 (秒)

	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 7.5: GeForece GTX680 での実行時間 (秒)

	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

8 終わりに

我々はCUDAプログラミングを容易にするためのフレームワークMESI-CUDAの開発を行っている．仮想共有変数を用いることでMESI-CUDAはユーザから低レベルのメモリ管理やデータ転送を隠蔽する．また，自動スレッドマッピング機構によりユーザはデバイス依存の手動最適化を行う必要が無い．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と自動スレッドマッピング機構の改良をそれぞれ設計・実装

表 7.6: TITAN での実行時間 (秒)

	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

し，評価を行った．その結果，MESI-CUDA が生成するコードのメモリアクセス効率が向上し，実行時間を削減することができた．

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．加えて，より多くのアプリケーションでの評価が上げられる．

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] Owens, John D. and Luebke, David and Govindaraju,Naga and Harris, Mark and Krüger, Jens and Lefohn, Aaron E. and Purcell, Timothy J.: *A Survey of General-Purpose Computation on Graphics Hardware*,Computer Graphics Forum,80-113,(2007). (2013.06.20).

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

[4] *OpenCL - The open standard for parallel programming of heterogeneous systems*, http://www.khronos.org/opencl/, (2013.06.20).

[5] K. Ohno and D. Michiura and M. Matsumoto and T. Sasaki and T. Kondo, *A GPGPU Programming Framework based on a Shared-memory model*,Proc. 23th IASTED Intl. Conf. on Parallel and Distributed Computing and Systems,310--318,(2011).

[6] K. Ohno and M. Matsumoto and T. Kamiya and T. Maruyama, *Supporting Dynamic Data Structures in a Shared-memory Based GPGPU Programming Framework*,Proc. 24th IASTED Intl. Conf. on Parallel and Distributed Computing and Systems,122--131,(2012).

[7] 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).

[8] T. Kamiya and T. Maruyama and K. Ohno and M. Matsumoto, *Compiler-Level Explicit Cache for a GPGPU Programming Framework*,Proc. The 2014 Intl. Conf. on

Parallel and Distributed Processing Techniques and Applications,(PDPTA'14)(2014).

[9] 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).

[10] NVIDIA CUDA C Programming Guide,NVIDIA Corporation,April (2012).

[11] *CUDA C Best Practices Guide*,NVIDIA Corporation,January (2012).

[12] 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).

[13] Muthu Baskaran, J.Ramanujam and P.Sadayappan: *Automatic C-to-CUDA Code Generation for Affine Programs*,Springer Berlin / Heidelberg,(2010).

[14] 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).

[15] *OpenACC*, http://www.openacc-standard.org/,(2013.06.7).

[16] Lee, S. and Min, S. and Eigenmann, R.Eigenmann, *OpenMP to GPGPU: a compiler framework for automatic translation and optimization*,SIGPLAN Not.,vol. 44,101--110,(2009).