修士論文

題目

GPU のシェアードメモリ自動利用機構における大規模データへの対応

指導教員

大野 和彦 講師

2023 年

三重大学大学院 工学研究科 情報工学専攻
コンピュータアーキテクチャ研究室

齊藤紡生 (421M514)

齊藤紡生

内容梗概

近年，GPU を用いて汎用計算を行う GPGPU の利用が活発である．しかし，CUDA などの現在主流の GPU プログラミング開発環境では GPU アーキテクチャを強く意識してプログラミングする必要がある．これに対し，我々は低レベルコードの記述量を減らした開発環境である MESI-CUDA を開発している．GPU は高速・小容量のシェアードメモリを複数搭載しており，MESI-CUDA はこれを利用するコードを自動生成するが，現在の手法ではデータサイズが大きい場合にうまく扱えない．本稿では，この問題を解決するデータ入れ替え機構を提案する．また，同じシェアードメモリを共有するスレッドのグループであるブロックの大きさを適切に指定するのが難しいという問題もあり，このブロックサイズの自動化手法も提案する．さらに，ブロックの集まりであるグリッドの次元が 1 次元のみであったのに対し，2 次元まで対応できるように拡張する．

シェアードメモリ入れ替え手法では大規模データの場合シェアードメモリにそのまま格納できないという問題に対し，シェアードメモリに格納可能な最大データサイズの情報をもとにシェアードメモリに格納できるサイズのデータに分割し，データを入れ替えている．これによりシェアードメモリに格納できないサイズのデータの場合でもシェアードメモリを利用できるようになり，速度を向上させることに成功した．

ブロックサイズ自動決定手法ではブロックサイズの最大が 1024 であることを考慮し，データサイズの 8 分の 1 と 1024 を比較し，小さい方をブロックサイズとしている．これにより最適なブロックサイズを必ず選択することは出来ないが，速度低下が著しくないブロックサイズを選択することが可能となり，実行速度への影響を押さえながらブロックサイズの指定を不要にした．

2 次元グリッドの適用では MESI-CUDA の記法を変更し，スレッドマッピングに制限を付けることによりブロックの集まりであるグリッドの次元を 2 次元まで対応させた．これによりカーネル関数を複数回呼び出す必要性がなくなり，並列度が上がることでカーネル関数呼び出しのオーバヘッドが減ったため，実行速度が向上した．

これらの手法を適用したプログラムを評価した結果，従来手法と比べて実行速度が向上した．また，従来手法と比較して MESI-CUDA コードにおける低レベルコードの記述量が減少した．

三重大学大学院　工学研究科

Abstract

In recent years, the use of GPGPUs, which use GPUs for general-purpose computation, has been active. However, the current mainstream GPU programming development environments such as CUDA require programming with a strong awareness of the GPU architecture. In contrast, we are developing MESI-CUDA, a development environment that reduces the amount of low-level code to be written. GPUs are equipped with multiple high-speed, small-capacity shared memories, and MESI-CUDA automatically generates code to use these memories. However, current methods do not handle large data sizes well. In this paper, we propose a data swapping mechanism to solve this problem. Another problem is that it is difficult to specify the appropriate size of blocks, which are groups of threads that share the same shared memory, and we propose an automated method for block size. Furthermore, the grid, which is a collection of blocks, has only one dimension. However, we extend it to support two dimensions.

The shared memory replacement method addresses the problem of large data that cannot be stored in shared memory as is. This method enables the use of shared memory even when the size of data cannot be stored in shared memory, thereby improving the speed.

The automatic block size determination method takes into account that the maximum block size is 1024, and compares one-eighth of the data size with 1024, and uses the smaller of the two as the block size. Although the optimal block size cannot always be selected, it is possible to select a block size that does not significantly reduce the speed, thus eliminating the need to specify the block size while minimizing the impact on execution speed.

For the 2D grid application, the MESI-CUDA notation has been changed to support up to two dimensions of the grid, which is a collection of blocks, by restricting the thread mapping. This eliminated the need for multiple kernel function calls, and the increased parallelism reduced the overhead of kernel function calls, resulting in increased execution speed.

Evaluation of programs using these methods showed that the execution speed was improved compared to the conventional method. In addition, the

amount of low-level code in the MESI-CUDA code was reduced compared to the conventional method.

目 次

図 目 次

表 目 次

1　はじめに

GPU は現在までムーアの法則に従い性能向上を続けており，様々な用途に使われている．中でも並列処理を用いた演算性能が注目され，GPU で汎用的計算を行う GPGPU (General Purpose computation on Graphics Processing Units) [1] が人工知能などの分野を中心に盛んに利用されている．GPU プログラミング開発環境としては CUDA[2] や OpenCL[3] などがあるが，GPU の性能を十分に発揮するためにはどちらも GPU アーキテクチャを深く理解し，低レベルなコーディングが必要とされる．そのため，GPU プログラミングをするユーザは GPU のアーキテクチャを十分に理解する必要があり，コーディングの難易度は高い．単純な処理内容だけでなく，GPU プログラミングではホスト側 (CPU) とデバイス側 (GPU) でデータをやり取りする必要があるため，転送用のコードも記述しなければならない．また，GPU のメモリ構造は複雑な階層構造となっており，サイズやアクセス速度も異なるため適切に使い分ける必要がある．そこで我々はデータ転送を自動化する GPU プログラミング用フレームワーク MESI-CUDA (Mie Experimental Shared-memory Interface for CUDA) [4][5][6] を開発している．MESI-CUDA では共有メモリ型 GPGPU プログラミングモデルを利用しているため，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も自動的に行い，デバイスに依存しないプログラムを容易作成することが可能になる．現状の MESI-CUDA ではアクセス速度の速いシェアードメモリの利用を自動的に適用する最適化が特定の条件下では利用可能である．しかし，データサイズが大きい場合にはシェアードメモリは利用せず，グローバルメモリのみを利用するコードとなり，速度が低下する．また，並列化を行う際にどの程度並列化するのかの数値はユーザーが決定しなければならず，GPU プログラミングの抽象化が中途半端な状態となっている．さらに，現在の MESI-CUDA ではブロックの集まりであるグリッドが 1 次元でしか利用できず，カーネル関数を複数回呼び出す必要があり，関数呼び出しのオーバヘッドにより実行速度が低下している．そこで我々は MESI-CUDA 上で大規模データでもシェアードメモリを利用できるようなシェアードメモリ自動利用機構 (以下シェアードメモリ自動入れ替え手法)，ユーザーがブロックサイズを決定する必要のないブロックサイズ自動決定手法，2 次元グリッドが利用可能となる 2 次元グリッド適用手法の 3 つの手法を提案する．

シェアードメモリ自動入れ替え手法では，従来手法では対応できなかったシェアードメモリに格納できないサイズのデータであっても，シェアー

ドメモリに格納可能な最大データサイズの情報をもとにシェアードメモリに格納可能なサイズのデータに分割し，データを入れ替える．ブロックサイズ自動決定手法では GPU アーキテクチャを深く理解していないユーザが決めるのは難しかったブロックサイズを，ブロックサイズの最大が 1024 であることを考慮し，データサイズの 8 分の 1 と 1024 を比較を行い，小さい方をブロックサイズとしている．2 次元グリッド適用手法では MESI-CUDA の記法を変更し，スレッドマッピングに制限をつけることで並列度を向上させ，速度を向上させることに成功した．

以下 2 章では CUDA と GPU アーキテクチャ，MESI-CUDA について解説する．3 章では関連研究を紹介し，4 章で提案手法について解説する．5 章では 3 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，評価する．最後に 6 章でまとめを行う．

2 背景

2.1 CUDA と GPU アーキテクチャ

CUDA[2] は NVIDIA 社 GPU の汎用並列コンピューティングプラットフォームであり，C 言語を拡張した文法を用いて GPU プログラムを開発することが出来る．しかし，最適化されたコードを記述するためには

NVIDIA 社 GPU のアーキテクチャを深く理解する必要がある．この章では NVIDIA 社 GPU のアーキテクチャについて解説する．

2.2 GPU の基本的なハードウェア構造

GPU は多数の計算コアを持つ高並列型プロセッサであり，並列計算を得意としている．一般的に CPU と併用して利用され，CPU 側のメモリと GPU 側のメモリは互いに独立している．そのため，計算に必要なデータなどはそのたびに CPU 側から GPU 側のメモリ，もしくはその逆へとデータ転送を行う必要がある．GPU のメモリ構造は階層構造となっており，主にグローバルメモリとシェアードメモリに分かれている．グローバルメモリは容量は大きいがアクセスが遅く，シェアードメモリは容量は小さく，アクセスが速い．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時間及びデータ転送時間の割合が非常に大きく，アクセス速度をどのようにして上げるかが高速化において重要である．そのため，限られた容量しかないがアクセス速度の速いシェアードメモリに局所性の高いデータを格納することで実行時間を短くすることが可能である．しかし，CUDA ではシェアードメモリの利用は完全にユーザーに委ねられており，格納できるデータサイズも限られているため，GPU

アーキテクチャに理解が深いユーザーが意識的にコーディングしなければシェアードメモリを効果的に利用することはできない．

2.3　どのように並列化処理が行われるか

並列化処理を行う場合，一般的にはスレッドと呼ばれるプログラム処理の実行単位ごとに処理内容を決め，その処理内容を同時に各スレッドが実行する．CUDA でも同様の手法で並列化処理が行われるが，CUDA では大量のスレッドを階層的に管理するための概念としてグリッド，ブロックと呼ばれるものを導入している．グリッドの中に複数のブロックが存在し，ブロックの中に複数のスレッドが存在する．グリッド，ブロックともに 3 次元で構成されているが，2 次元空間で利用することが多い．CUDA の仕様では，最高で 65535 × 65535 × 512 個のスレッドを実行できるが，1 つのブロックが管理する最大のスレッド数は 1024 である．前述したシェアードメモリはブロックごとに存在するメモリであり，ブロック内のスレッド同士で共有される．また，本研究では 1 ブロックあたりのスレッド数をブロックサイズと呼ぶ．

CUDA にはビルトイン変数が存在し，宣言することなくカーネル関数内で使用することが出来る．各ブロック，スレッドにそれぞれ番号が割り当て

られており , blockIdx.x で x 次元方向のブロックの番号を , threadIdx.x で x 次元方向のスレッド番号を得ることが出来る . また , blockDim.x で x 次元方向のスレッドの個数を得ることが出来 , x の部分を y , z とすることで y 次元方向 , z 次元方向の値にすることができる . これらのビルトイン変数を用いて各スレッドごとに一意な値を得ることが出来る .

CUDA のスレッドの階層構造の例として 2 次元グリッド 2 ブロックのグリッドブロックの図を図 2.1 に記載する . 図 2.1 ではブロックサイズが BS × BS であり , th(1,0) が x 次元方向のインデックスが 1 , y 次元方向のインデックスが 0 であるスレッドを意味する . このようにブロックが n 次元的にマッピングされているグリッドを n 次元グリッド , ブロック内のスレッドが n 次元的にマッピングされているものを n 次元ブロックと本研究では呼ぶ .

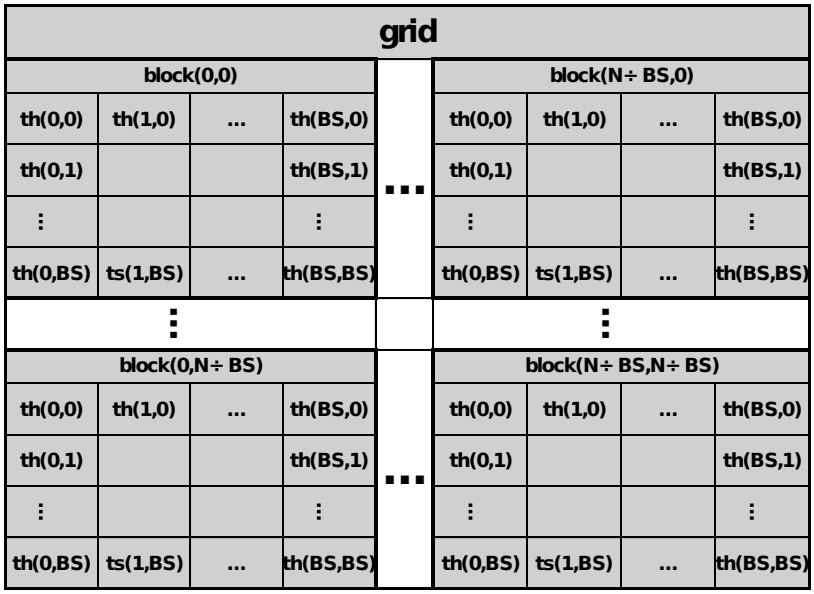

図 **2.1:** 2 次元グリッド 2 次元ブロックの場合のスレッド構造

CUDA では CPU 側をホスト，GPU 側をデバイスと呼ぶ．デバイス上で実行される関数はカーネル関数と呼ばれ，ホスト側のコードからカーネル関数を呼び出すことで，デバイス（GPU）上でカーネル関数を実行することができる．この際にスレッドの数を指定する必要があり，これらはユーザーによって指定される．

2.4 CUDA でどのように書くか

ここでは CUDA での実際のプログラム記法について説明する．CUDA ではデバイス上で実行される関数はカーネル関数と呼ばれ，CUDA プロ

グラム上ではカーネル関数に__global__または__device__を付与して記述する．それに対し，修飾子のない関数や__host__の修飾子がついた関数はホスト側で呼び出される．また，カーネル関数内で変数の型宣言前に修飾し__shared__を付けることでシェアードメモリ上にデータ領域が確保される．

次に具体的なプログラム例を用いて説明する．図 2.2 に CUDA プログラム例を示す．6 行目にカーネル関数が記述されており，その中身が 1 スレッドが担当する処理となっている．8 行目ではビルトイン変数を用いてスレッドごとに一意なインデックスを求めている．27 行目でカーネル関数がホスト側で呼び出されており，ブロックの数が N ÷ BS (ブロックサイズ)，1 ブロックあたりのスレッド数が BS 個となるようにスレッドを指定してカーネル関数を起動している．カーネル関数の呼び出し前後ではメモリの確保とデータ転送，メモリの開放を行っている．

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

図 2.2: 行列積の CUDA プログラム例

2.5 MESI-CUDA

MESI-CUDA フレームワークはデータ転送コードやメモリ確保· 解放などのコードを自動的に生成することで低レベルコードの記述量を減らす GPU プログラミングフレームワークである．入力は MESI-CUDA コードであり，出力が最適化された CUDA コードであるので，ユーザは MESI-CUDA でのコーディングが可能であれば CUDA プログラムを作成することが可能となる．仮想的な共有メモリ環境のモデルを採用しており，変数宣言の修飾子として`__global__`を付けることでホストとデバイス両方よりアクセス可能な共有変数が提供される．共有変数を利用することでホストメモリ，デバイスメモリを意識せずに記述することができ，データ転送の記述，変数の使い分けが不要になる．仮想ホストメモリとデバイス (GPU) への処理の分配やカーネル関数の記述はユーザー自身が記述する必要がある．

MESI-CUDA では GPU を利用するプログラム作成は容易になるが，一般的に手動最適化したコードよりは効率が落ちる．それに対し，静的解析などを行い，可能な限り手動最適化コードに性能を近づけることを目的の 1 つとして本研究を行っている．

2.5.1 MESI-CUDA プログラム例

図 2.2 で示した行列積の CUDA プログラムを MESI-CUDA で記述したものを図 2.3 にプログラム例として示す . 4 行目で共有メモリ変数を宣言することでホストとデバイスを意識することなくプログラミングすることが可能となっている . 19 行目でカーネル関数を呼び出しているが , MESI-CUDA ではその前後でメモリの転送や確保をする必要がない . CUDA コードと比べてコード量も少なく , 低レベルコードが無くなっていることが分かる .

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

図 **2.3:** 行列積の MESI-CUDA プログラム例

3　関連研究

MESI-CUDA 以外にも，低レベルなアーキテクチャを隠蔽し，より容易な GPU プログラミング環境を提供する研究が行われている．

OpenACC[7] や OpenMP-to-CUDA translation[8] では逐次プログラムに対して指示文を挿入することにより，GPU で実行可能な並列プログラムへコンパイルされる．そのため，コンパイラが自動でスレッド化されたコードの生成と物理リソースやデータへの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遅い．マッピング制御のための指示文も用意はされており，ある程度は手動で最適化することも可能だが，CUDA によるプログラミング同様に GPU アーキテクチャに対する深い理解が必要となる．

神谷らの研究 [9] では MESI-CUDA でシェアードメモリを自動利用する最適化手法を提案している．この手法では 2 回以上アクセスがある配列が複数ある場合，全ての配列に対してアクセス回数を解析し，1 バイト当たりの平均アクセス回数が最大の配列をシェアードメモリ内に格納する．しかし，データのサイズが大きい場合は対象となるデータがシェアードメモリの容量を超えてしまうため，シェアードメモリに格納できないという問題がある．また，神谷らの手法ではカーネル関数を起動する際のブ

ロックやスレッドの数を手動で決めているが，GPU プログラミングに詳しくない人物が最適なブロックサイズを適切に選択するのは困難である．

4 提案手法

4.1 シェアードメモリ自動入れ替え手法

現在の MESI-CUDA 処理系ではデータが大規模すぎる場合，シェアードメモリにデータを格納することが出来ないため，利用することが出来ない．この問題に対して，格納するデータを自動で入れ替えることで，大規模データに対してもシェアードメモリを自動で利用できるようにする．今回実装する機構の対象とするプログラムが満たす条件は MESI-CUDA プログラムは 2 次元までのグリッド，1 次元ブロックで 1 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格納する対象配列のアクセスが連続であるものとする．また，データサイズは 2 のべき乗であるとする．

静的解析によりシェアードメモリに格納する候補として配列 a が選ばれたと仮定して，ブロック内でアクセスする a の範囲がシェアードメモリの容量を超える場合，スレッド内でループを用いて複数回のアクセスを行うケースが想定される．アクセスされる a の要素がメモリ上で連続

しているなら，元のループを k 分割し，各分割ループ毎に必要な範囲をシェアードメモリに格納することでブロック内アクセス範囲の大きさにかかわらずシェアードメモリを利用することが可能になる．k はシェアードメモリに格納する配列を k 分割した時にシェアードメモリ最大容量よりも小さくなるように決定する．以下にシェアードメモリ自動入れ替え手法のコード生成フローを示す．

1. シェアードメモリに格納する配列を k 分割し，シェアードメモリ最大容量より小さく，ブロックサイズの倍数になるように k の値を決定する

2. 元の単一ループを入れ子の 2 つのループに分割する（内側ループは元のループ回数の $1/k$，外側ループのループ回数は k ループ）

3. 配列に格納されているデータを置き換えるコードを追加挿入する

4. 配列のインデックス式を修正する

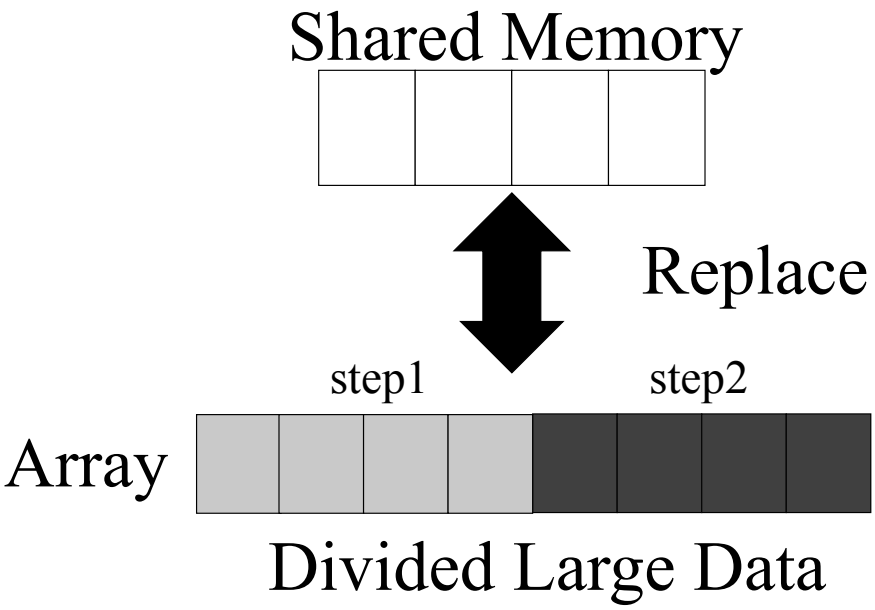

図 **4.4:** シェアードメモリの入れ替え

```
if(2 回以上アクセスがある配列が存在する)
  各配列のアクセス回数を解析
  1 番多いものを格納配列とする
  if(配列がシェアードメモリに格納できない)
    ループ毎に必要な範囲のデータを格納
  else
    シェアードメモリに配列の全てを格納するコードを生成
else
  シェアードメモリは使用せずにコードを生成
```

図 **4.5:** シェアードメモリ自動入れ替え手法のコード生成フロー

図 4.6 に従来手法でのカーネル関数の記述，図 4.7 に MESI-CUDA ファイルを本手法を適用して出力した CUDA ファイルのカーネル関数部分を

記載する．図 4.6 の 5 行目ではインクリメントを行い，ループを N 回行っている．これに対し提案手法適用後の CUDA コードである図 4.7 ではループを 2 つに分割し，6 行目の外側のループでは分割されたデータサイズ分だけイテレータ変数を増やしている．7 行目の内側の 1 つ目のループでシェアードメモリへデータを格納し，9 行目の 2 つ目のループで分割されたデータサイズ分の演算を行う．このようにループを 2 重に分割することでシェアードメモリに格納するデータの入れ替えを自動で行っている．

```
1  __global__ void kernel(int *a, int *b, int *c){
2    int i;
3    int id=blockDim.x*blockIdx.x+threadIdx.x;
4    c[id]=0;
5    for(i=0;i<N;i++){
6      c[id] += a[i] * b[(i*N) + id];
7    }
8  }
```

図 **4.6:** 従来手法によって出力される行列積の CUDA プログラム

```
1  int i,j;
2  __shared__ int s_a[SM_SIZE];
3  int id = blockDim.x * blockIdx.x +threadIdx.x;
4  c[id] = 0;
5
6  for(i=0;i<N;i+=SM_SIZE){
7    for(j=threadIdx.x;j<SM_SIZE;j+=blockDim.x)
8      s_a[j] = a[i+j];
9    for(j=0;j<SM_SIZE;j++)
10     c[id] += s_a[j] * b[(i+j)*N+ id];
11 }
```

図 **4.7:** 提案手法によって出力される行列積の CUDA プログラム

4.2　ブロックサイズ自動決定手法

現在の MESI-CUDA 処理系ではブロックサイズはユーザが自分で決定する必要があるが，本研究が想定しているユーザーは GPU アーキテクチャに詳しくないユーザーであり，適切なブロックサイズを設定するのは難しい．そこでブロックサイズをデータサイズから自動決定することでブロックサイズを隠蔽する手法を提案する．本手法はあくまでブロックサイズの隠ぺいを目的としており，速度的な向上は目的としていない．

N,BS をそれぞれ対象配列 a の要素数，ブロックサイズとした時，以下のようにブロックサイズを決定する．

1. BS= 2^t,t は 2^t=N/8 を満たす最大の整数値である

2. 1024 と BS で小さい値をブロックサイズとして採用する

CUDA のブロックサイズの上限は 1024 であるため，1024 と比較して小さい値をブロックサイズとして採用している．本手法では必ず最適なブロックサイズを選択できるわけではないが，最低でも 1 次元方向に 8 個のブロックを確保することができるため，ある程度の性能は発揮できると考えられる．これにより，ブロックサイズを隠蔽した MESI-CUDA ファイルでも出力ファイルではブロックによるスレッドマッピングが可能と

なる．

4.3　2 次元グリッドの適用

現在の MESI-CUDA 処理系では 1 次元グリッド，1 次元ブロックのカーネル関数しか利用できないため，スレッド数が足りない場合はカーネル関数を複数回呼び出す必要がある．プログラム中に GPU で並列化可能な処理が存在する場合，基本的には 1 回のカーネル関数の呼び出しで実現するのが望ましい．そのため，スレッド数が足りない場合はグリッド，ブロックの次元を増やすもしくはブロックサイズの拡大などで対応するべきである．そこで 2 次元グリッド 1 次元ブロックのカーネル関数の呼び出しが可能な MESI-CUDA 記法を提案する．2 次元グリッドの自動化において一番問題となるのがブロックサイズをどのように決定するべきかだが，先ほど述べた手法によってブロックサイズは決まっているものとする．またすべてのプログラムに対応するのは難しいため，2 次元グリッドの場合は要素数 N*N の配列に対して x 方向に N スレッド，y 方向に N スレッドとし，合計で N × N 個のスレッドを生成するものとする．

従来手法ではカーネル関数の呼び出しの際にブロックの数，ブロック内のスレッドの数を 1 次元で指定して引数としていたが，提案手法では

グリッドの次元数と 1 次元方向に生成するスレッド数を引数として渡す．ブロックサイズは自動決定されているため，生成するスレッド数をブロックサイズで割ったものをブロックの数とし，1 ブロックのスレッド数をブロックサイズとすることでグリッドの次元数× 指定された数のスレッド数を生成することが可能である．

従来手法での行列積の記述を図 4.8 に示す．従来手法では 1 次元グリッド 1 次元ブロックでのスレッド生成となっており，スレッドの数が足りないため，複数回カーネル関数を呼び出している．カーネル関数の呼び出しにはオーバヘッドがあり，これが速度低下の一因となっている．

```
1  ...
2
3  int main(int argc, char *argv[]){
4    init_array(a);
5    init_array(b);
6
7    for(int i=0;i<N;i++){
8      kernel<<<N/BS, BS>>>(a +(i*N), b , c +(i*N));
9    }
10 }
```

図 **4.8:** 従来手法による行列積の MESI-CUDA プログラム例

提案手法での行列積の記述を図 4.9 に示す．7 行目に示すとおり，カーネル関数の呼び出しは 1 回となっており，オーバヘッド削減による実行速度向上が見込める．グリッドは 2 次元の指定となっており，1 次元方向

あたりのスレッド数が N であるので N × N 個のスレッドが生成される．

```
1 ...
2
3 int main(int argc, char *argv[]){
4   init_array(a);
5   init_array(b);
6
7   kernel[[[2, N]]](a, b , c);
8 }
```

図 **4.9:** 提案手法による行列積の MESI-CUDA プログラム例

5 評価

N 次正方行列の行列積とヤコビ法を評価プログラムとして実行速度の測定を行った．実行環境を以下に示す．

表 **5.1:** 評価プログラムの実行環境

CPU	AMD Ryzen 7 3700X 3.60Ghz
メモリ	48GB
GPU	GeForce RTX 2070 SUPER

N=2^t であり，どちらのプログラムも N=4096 までは対象とする配列がシェアードメモリに格納することが可能だが，N=8192 からは従来手法ではシェアードメモリを使用出来ない．また，ブロックサイズ (1 ブロック当たりのスレッド数) を BS と表記する．

5.1 シェアードメモリ自動入れ替え手法の評価

N=8192，BS=256 で評価を行った．従来手法ではシェアードメモリを利用できないのに対し，提案手法ではシェアードメモリを利用している．行列積，ヤコビ法ともに提案手法の方が高速になっていることが分かる．行列積のプログラムはデータの局所性が低く，シェアードメモリの効果が低めの結果になっている．ヤコビ法のプログラムでは前後のインデックスの値を用いて計算をするため，シェアードメモリに格納したデータの局所性が高いことから大幅な速度向上に繋がったと考えられる．

表 **5.2:** シェアードメモリ自動入れ替え手法の評価結果

実験プログラム	従来手法 (s)	提案手法 (s)	実行時間比 (%)
行列積	24.22	21.64	89.4
ヤコビ法	929.54	301.54	32.4

表 5.3 に行列積のみの結果，表 5.4 にヤコビ法のみの結果を記載する．行列積はシェアードメモリの効果が薄いため，N=4096 までの場合と N=8192 のときにシェアードメモリを使用できない場合でそこまで差はない．ヤコビ法ではデータの局所性が高く，シェアードメモリの効果が高いため，N=8192 になると同時に従来手法では劇的に実行速度が落ちている．提案手法による大規模データへの対応が効果的であったといえる．

表 **5.3:** 行列積の評価結果 (シェアードメモリ入れ替え手法)

N	従来手法 (s)	提案手法 (s)
1024	0.24	0.23
2048	1.27	1.28
4096	5.18	5.18
8192	24.22	21.64

表 **5.4:** ヤコビ法の評価結果 (シェアードメモリ入れ替え手法)

N	従来手法 (s)	提案手法 (s)
1024	5.24	5.32
2048	20.62	20.64
4096	81.62	81.03
8192	929.54	301.54

5.2 ブロックサイズ自動決定手法の評価

N=4096 で評価を行い，従来手法では最も実行時間の短かった際に使用していたブロックサイズを選択し，提案手法では自動的にブロックサイズを決定して評価を行った．実行時間の括弧の中がブロックサイズである．最適なブロックサイズは選択できていないが，どちらのプログラムも実行速度が著しく下がるようなブロックサイズの決定は行っていないことがわかる．

表 **5.5:** ブロックサイズ自動決定手法の評価結果

実験プログラム	従来手法 (s)	提案手法 (s)	実行時間比 (%)
行列積	5.18(256)	5.29(512)	102.1
ヤコビ法	80.51(128)	81.05(512)	100.7

表 5.6 に行列積のみの結果，表 5.7 にヤコビ法の結果を示す．どちらのプログラムも著しい速度低下を発生させるようなブロックサイズに決定してはいないが，データサイズが大きくなるにつれて適切なブロックサイズとのずれが広がる傾向にある．もう少し違う N での実験をすることができれば，より最適なブロックサイズを決定するための式が求められる可能性はある．

表 **5.6:** 行列積の評価結果 (ブロックサイズ自動決定手法)

N	従来手法 (s)	提案手法 (s)
1024	0.20(128)	0.20(128)
2048	1.27(256)	1.27(256)
4096	5.18(256)	5.29(512)

表 **5.7:** ヤコビ法の評価結果 (ブロックサイズ自動決定手法)

N	従来手法 (s)	提案手法 (s)
1024	5.12(128)	5.12(128)
2048	19.95(128)	20.62(256)
4096	80.51(128)	81.05(512)

5.3　2 次元グリッドの適用の評価

N=4096，BS=256 で評価を行った．行列積では実行時間が 9.85%まで削減されているのに対し，ヤコビ法では 92.4%となっている．これは行列積が 3 重ループなのに対し，ヤコビ法では 2 重ループのため，並列度を上げた場合の速度向上率が低くなっていると考えられる．また，ヤコビ法ではシェアードメモリによるアクセス速度向上が著しく，本手法の速度向上率を上回っているため，本手法の効果が薄いと考えられる．

表 **5.8:** 2 次元グリッド適用の評価結果

実験プログラム	従来手法 (s)	提案手法 (s)	実行時間比 (%)
行列積	5.18	1.00	19.3
ヤコビ法	81.62	75.17	92.1

表 5.9 に行列積の結果，表 5.10 にヤコビ法の結果を示す．N=4096 のときの同様に行列積に対しては全体的に効果的であり，ヤコビ法に対しては効果は薄いと考えられる．

表 **5.9:** 行列積の評価結果 (2 次元グリッドの適用)

N	従来手法 (s)	提案手法 (s)
1024	0.239	0.012
2048	1.271	0.086
4096	5.176	0.997

表 **5.10:** ヤコビ法の評価結果 (2 次元グリッドの適用)

N	従来手法 (s)	提案手法 (s)
1024	5.24	4.93
2048	20.62	19.05
4096	81.62	75.17

5.4 複数の提案手法の評価

N=8192，従来手法は一番高速なブロックサイズを選択し，提案手法ではブロックサイズを自動的に決定した．行列積ではシェアードメモリ自動入れ替え手法，ブロックサイズ自動決定手法，2 次元グリッドの適用の 3 つの提案手法を適用したもので評価を行った．ヤコビ法は逐次プログラムで 2 重ループであり，シェアードメモリ自動入れ替え手法を適用するためにはカーネル関数内に for 文がなければならないため，シェアードメモリ自動入れ替え手法と 2 次元グリッドの適用は同時に適用できない．今回はシェアードメモリ自動入れ替え手法の方が高速であるため，ブロックサイズ自動決定手法とシェアードメモリ自動入れ替え手法の 2 つの手法を適用したもので評価を行う．また，今回は同条件で OpenACC，手動で最適化したプログラムとも比較する．どちらのプログラムも OpenACC と比べて提案手法の速度が速く，手動最適化したプログラムよりも遅く

なっている．しかし，提案手法を用いたプログラムは手動最適化したプログラムの実行速度に近く，十分な性能があると言える．

表 **5.11:** 複数の提案手法適用した場合の評価

実験プログラム	OpenACC(s)	従来手法 (s)	提案手法 (s)	手動最適化 (s)
行列積	101.52	24.22	1.00	0.83
ヤコビ法	444.12	929.54	302.57	257.63

6　おわりに

我々は GPU 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．MESI-CUDA では既に仮想共有変数を用いることで低レベルのメモリ管理やデータ転送を隠蔽し，データサイズが小さければシェアードメモリの利用も自動で行うことが可能である．本研究では大規模データを利用したプログラムであっても MESI-CUDA 上で大規模データでもシェアードメモリを利用できるようなシェアードメモリ自動利用機構，ユーザーがブロックサイズを決定する必要のないブロックサイズ自動決定手法，2 次元グリッドが利用可能となる 2 次元グリッド適用手法の 3 つの手法を提案した．シェアードメモリ入れ替え手法と 2 次元グリッド適用手法を利用した行列積とヤコビ法のプログラムで評価を行ったところ，大規模データを利用した MESI-CUDA コードでも実行時

間を削減することが出来た．ブロックサイズ自動決定手法を適用して同様の評価を行ったところ，著しい速度低下はない状態でブロックサイズの自動決定に成功した．

今後の課題として，本研究ではブロックはいまだに 1 次元にしか対応していないため，ブロックも 2 次元に対応すればより速度を向上できる可能性がある．2 次元グリッドの際は N× N のスレッドマッピングしか許されていないため，2 次元グリッドの際のスレッドマッピングの拡張によってより柔軟性が向上する可能性がある．また，評価結果から考えられるに，ブロックサイズ自動決定手法の式が最適ではないため，より効果的にブロックサイズを決定するための式が必要である．

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，高木一義教授，深澤祐樹技術員に深く感謝致します．また，助言などで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] Dimitrov, Martin, Mike Mantor, and Huiyang Zhou. "Understanding software approaches for GPGPU reliability." Proceedings of 2nd Workshop

on General Purpose Processing on Graphics Processing Units. 2009.

[2] NVIDIA DEVELOPER,https://developer.nvidia.com/cuda-toolkit,(2023.01.23).

[3] OpenCL - The open standard for parallel programming , http://www.khronos.org/opencl/,(2023.01.22).

[4] Ohno, Kazuhiko, and Rei Yamamoto. "Dynamic Task Scheduling Scheme for a GPGPU Programming Framework." 2015 Third International Symposium on Computing and Networking (CANDAR). IEEE, 2015.

[5] Ohno, Kazuhiko, et al. "Supporting dynamic data structures in a shared-memory based GPGPU programming framework." Proc. 24th IASTED Intl. Conf. on Parallel and Distributed Computing and Systems. 2012.

[6] Kamiya, Tomoharu, et al. "Compiler-level explicit cache for a GPGPU programming framework." Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2014.

[7] OpenACC, http://www.openacc-standard.org/,(2023.01.23).

[8] Lee, Seyong, Seung-Jai Min, and Rudolf Eigenmann. "OpenMP to GPGPU: a compiler framework for automatic translation and optimization." ACM Sigplan Notices 44.4 (2009): 101-110..

[9] 神谷智晴, et al. "GPGPU のシェアードメモリを利用する自動最適化機構." 研究報告ハイパフォーマンスコンピューティング (HPC)2013.30 (2013): 1-8.

[10] Yang, Yi, et al. "A GPGPU compiler for memory optimization and parallelism management." ACM Sigplan Notices 45.6 (2010): 86-97.

[11] Kamiya, Tomoharu, et al. "Compiler-level explicit cache for a GPGPU programming framework." Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2014.