卒業研究論文

適切なメモリの自動選択による CUDA 向け並列化フレームワーク OpenMPC の改良

指導教員　　津邑 公暁 准教授
松尾 啓志 教授

名古屋工業大学 工学部 情報工学科
平成 20 年度入学　20115028 番

内山 寛章

平成 24 年 2 月 8 日

適切なメモリの自動選択による CUDA 向け並列化フレームワーク OpenMPC の改良

内山 寛章

内容梗概

GPU は画像処理に特化したプロセッサである．そのため，以前はグラフィックスプログラマがシェーダ言語やグラフィックス API を用いて GPU 向けプログラムを記述することが一般的だった．しかしながら，現在では GPU が持つ広域なメモリバンド幅や，高いベクトル演算性能を活かして，画像処理以外の汎用演算を行わせる GPGPU が注目を集めている．そのために，GPU 向けの C 言語統合開発環境として CUDA が開発されている．これにより，幅広い分野のプログラマが GPU 向けのプログラミングを行うことが可能になった．しかし，CUDA を用いたプログラミングでは，GPU アーキテクチャの知識や CUDA 特有の記述などが必要であり，依然としてプログラマにとっての負担は大きい．

そこで，CUDA を利用する際のプログラマの負担を軽減するために，多くの CUDA 向けプログラミングフレームワークが研究されている．その中の一つに OpenMPC がある．これは，既存の CPU 向け並列プログラミングフレームワークである OpenMP の記法をベースにしている．そのため，プログラマにとって学習コストが少ない．そして，逐次プログラムに pragma を用いて指示を出すことで，GPU を用いた並列プログラムを容易に作成することが可能である. しかし，OpenMPC で単純に変換しただけのプログラムには 2 つの問題がある．1 つめは，変数の配置先として高速にアクセス可能なデバイス・メモリを選択することができないため，プログラムの実行時にメモリアクセスがボトルネックとなる可能性があることである．2 つめは，データを効率的に転送可能なメモリが使用されないため，CPU ― GPU 間でデータを高速に転送できないことである．そのため，これら 2 つの事から，OpenMPC では GPU の性能を十分に引き出すことが出来ていない．

そこで本論文では，OpenMPC でのメモリ選択を改良し，適切なデバイス・メモリへの変数配置や，データ転送の効率化を可能とする手法を提案する．これにより，プログラマの負担を増やすことなく，メモリアクセスやデータ転送の時間が削減される．

提案手法の有効性を確認するため，いくつかのベンチマークプログラムを用いて既存手法との実行速度を比較した．評価の結果，既存の OpenMPC に対して平均 14%,

最大で 42%の速度向上が得られることを確認した.

適切なメモリの自動選択による CUDA 向け並列化フレームワーク OpenMPC の改良

目次

1 はじめに

GPU (Graphics Processing Unit) は画像処理に特化したプロセッサであり，一般的に広域なメモリバンド幅や，高いベクトル演算性能を持つ．また，CPU と比べて搭載しているコアの数が多いため，並列処理に適したプロセッサである．以前は，グラフィックスプログラマによってシェーダ言語やグラフィックス API を用いたプログラミングが行われていたが，現在ではこの高い演算性能を活かして，GPU に画像処理以外の汎用演算を行わせる GPGPU が注目を集めている．特に，HPC (High Performance Computing) 分野において，GPU は CPU に比べて電力あたりの性能が高いことや，既存の HPC 向けのアクセラレータと比べて安価なことなどが理由となり，採用が進んでいる．そこで，GPU 向けの汎用的なプログラムを記述するために，GPU 向けの C 言語統合開発環境として **CUDA (Compute Unified Device Architecture)** が開発されている．

CUDA は GPU 上に存在する複数のコアに大量のスレッドを割り当て，その大量のスレッドに同じ命令を並列に実行させることで処理の高速化を実現する．しかし，CUDA を利用するためにはスレッドの割り当てや CPU と GPU の間での明示的なデータ転送など，GPU や CUDA に関する深い知識が必要であるため，プログラマにとっての負担が大きい．そこで，このような CUDA 特有の記述を隠蔽し，プログラマの負担を軽減するフレームワークが研究されている．

本研究ではそのような CUDA 向けフレームワークの 1 つである OpenMPC に着目する．OpenMPC は，既存の CPU 向け並列化フレームワークである OpenMP の記法をベースとしている．このため，プログラマにとって言語仕様やプログラミングモデルを新たに習得するコストが少ない．しかし OpenMPC では，GPU 上のメモリの中から適切なメモリを選択し，プログラマが明示的に使用する必要がある．また，CUDA のライブラリ関数を使用できないため，データの効率的な転送ができない．

そこで本研究では，GPU 上の適切なメモリを自動的に選択する手法を提案する．また，これに加え，効率的にデータを転送する手法も提案する．これらの手法を適用することによってプログラミングの困難さを抑えつつ，プログラムの高速化を図る．

以下，2 章で本研究で対象とする，CUDA の概要と既存研究について述べる．3 章では OpenMPC の抱える問題点を述べ，それを解決するための手法を提案し，4 章で提案手法の実装について述べる．5 章で本提案手法の評価を行い，最後に 6 章で結論を述べる．

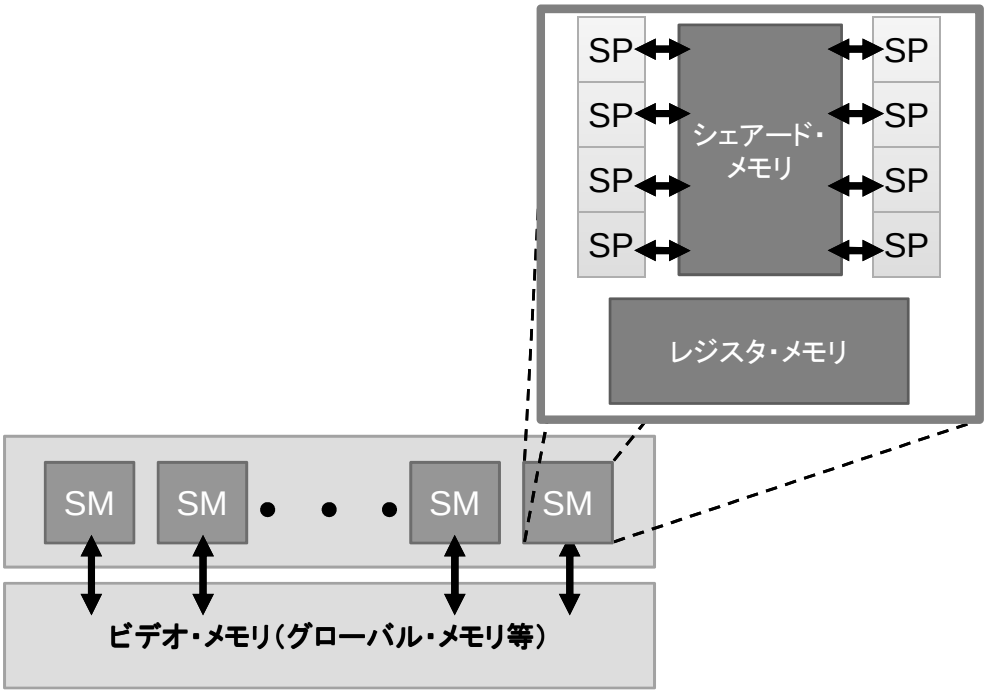

図 1: GPU のアーキテクチャ

2 背景

本章では既存の GPU 向けプログラミング環境である CUDA 及び，その複雑さを緩和する既存のフレームワークについて述べる．また，本研究で対象とする OpenMPC について説明する．

2.1 CUDA

初期の GPU 向けプログラムは，シェーダ言語などを用いてプログラムを記述する必要があった．しかし，シェーダ言語は本来，グラフィックス向けの技術である．そのため，プログラマには，GPU 向けプログラムを記述する際に，グラフィックスプログラミングの知識が要求されていた．そこで，NVIDIA 社により GPU 向けの C 言語統合開発環境として CUDA[1] が開発されている．CUDA はプログラムの記述に C 言語が使用できるため，プログラマに対してグラフィックスプログラミングの知識が不要となっている．

NVIDIA 社の GPU のアーキテクチャを**図 1** に示す．GPU の世代や製品により多少の違いはあるが，そのアーキテクチャのモデルは，ほぼ同じである．GPU チップの内部には多数のストリーミング・マルチプロセッサ (SM) が存在する．さらに各 SM 内

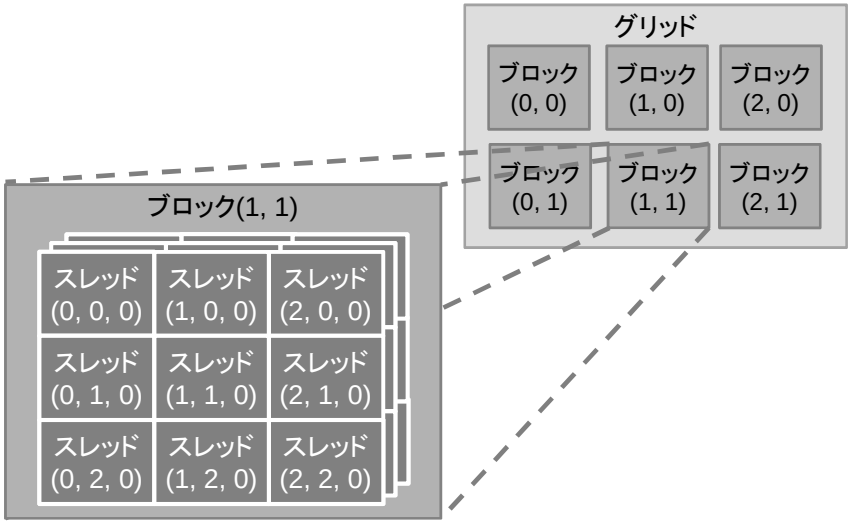

図 2: 階層的なスレッド管理 (実行構成)

には演算処理ユニットであるストリーミング・プロセッサ (SP) が 8 個存在する．この SM 内の 8 個の SP は，SIMD 型で同じ命令を実行する．以下では，CUDA の階層的なスレッド管理やメモリモデル，プログラミングモデルについて述べる．

階層的なスレッド管理

GPU は，SP を用いて複数のスレッドを並列実行することで高い演算性能を実現している．CUDA の仕様では，GPU に対して最大で 65535 × 65535 × 512 個のスレッドを実行させることが可能である．この大量のスレッドは**図 2** のように階層的に管理される．スレッドの集合をブロックと呼び，ブロック内でスレッドは x 軸方向，y 軸方向，z 軸方向の 3 次元的に配置されている．また同数のスレッドから構成されるブロックの集合をグリッドと呼び，グリッド内でブロックは 2 次元的に配置されている．このようなグリッドの次元やグリッド内のブロック数，およびブロックの次元やブロック内のスレッド数を実行構成と呼ぶ．GPU の性能を引き出すためにはこの実行構成を SP の数に対して適切に設定することが重要である．

メモリモデル

CUDA のメモリモデルを**図 3** に示す．CUDA では，レジスタ・メモリ，ローカル・メモリ，シェアード・メモリ，グローバル・メモリ，テクスチャ・メモリ，コンスタント・メモリが使用できる．これらはグラフィック・ボード上に存在するので，デバイス・メモリと呼ばれる．その詳細を**表 1** に示す．表中のアクセス欄はスレッドからのアクセス権を表し，R/W は読み書き可能，R は読み出しのみ可能を表す．一方で，CPU 側のメモリはホスト・メモリと呼ばれ，デバイス・メモリとホスト・メモリはアドレ

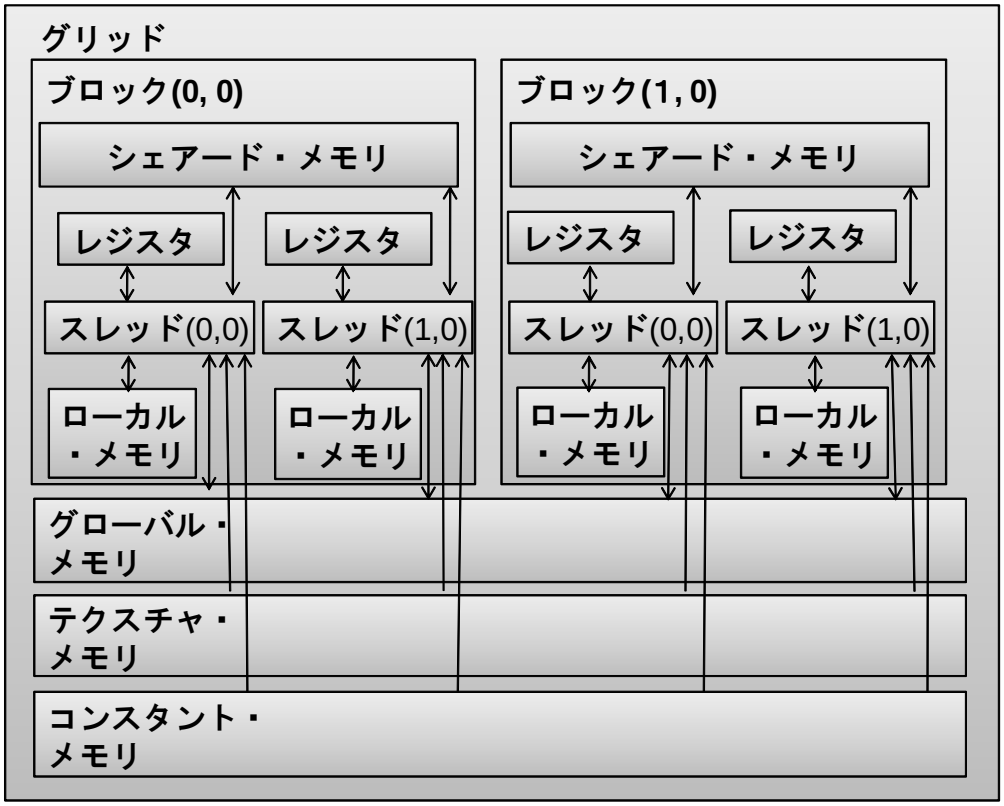

図 3: CUDA のメモリモデル

表 1: デバイス・メモリ

種類	場所	キャッシュ	アクセス
レジスタ・メモリ	チップ上	-	R/W
ローカル・メモリ	チップ外	されない	R/W
シェアード・メモリ	チップ上	-	R/W
グローバル・メモリ	チップ外	されない	R/W
テクスチャ・メモリ	チップ外	される	R
コンスタント・メモリ	チップ外	される	R

ス空間が異なるため，CUDA が提供する API を用いて明示的にデータをやり取りする必要がある．

デバイス・メモリのうちレジスタ・メモリには一時変数が割り当てられ，ローカル・メモリにはレジスタ・メモリから溢れた変数が割り当てられる．これら 2 つのメモリは

コンパイラが自動的に割り当てを決定する．そのため，レジスタ・メモリとローカル・メモリをプログラマが意識する必要はない．一方で，シェアード・メモリ，グローバル・メモリ，テクスチャ・メモリ，コンスタント・メモリの 4 つは，プログラマが領域を明示的に割り当てる必要がある．各メモリの物理的な場所，アクセスやキャッシュの有無などは異なるため，処理の特性に応じて適切なメモリを選択することにより，処理速度の向上が期待できる．各メモリを利用するためには，それぞれに用意されたアクセス関数や，変数修飾子を使わなければならない.

プログラミングモデル

CUDA を利用して記述されたプログラムは，CPU で実行されるホスト・コードと GPU で実行されるデバイス・コードで構成される．ホスト・コードには通常の C 言語プログラムに加えてカーネル関数コールなどが記述される．カーネル関数とは GPU に計算させる処理を記述した関数の事であり，これはデバイス・コードに記述される．カーネル関数コールと共に，カーネル関数を実行する実行構成を指定できる．なお，グリッド内の全てのスレッドは，同一のカーネル関数を実行する．また，各スレッドは固有の ID を持ち，その ID などを表す CUDA のビルトイン変数を用いることで当該スレッドがどのメモリアドレスにアクセスするのかを指定できる．

2.2 既存研究

CUDA を用いたプログラミングでは，ホスト・メモリ－デバイス・メモリ間の明示的なデータ転送や，実行構成などを記述する必要がある．また，CUDA プログラムを高速に実行させるには，CUDA や GPU に関する深い知識も必要となる．

そこで，プログラマの負担を軽減するために，CUDA プログラミングの繁雑さを緩和するための手法が多く提案されている．道浦ら [2] は CUDA プログラムにおけるホスト・メモリ－デバイス・メモリ間のデータ転送コードを自動的に生成するコンパイラを提案している．この手法では，ホスト・メモリ－デバイス・メモリ間で転送される変数をプログラマが指定する．コンパイラはこの情報を用いて，データ転送用コードや，データ領域確保用コードを生成する．しかし，この手法ではデータ転送のみに着目しており，CUDA を用いたプログラミングにおいて，並列性を意識する必要があるという問題は解決されていない．また，GPU で実行されるカーネル関数も自分で記述する必要がある．

また，CUDA を抽象化する事でプログラマの負担を軽減するフレームワークとして hiCUDA[3] が提案されている．この手法では，逐次プログラム中の，GPU で実行した

い場所を pragma で指示することで，hiCUDA コンパイラに GPU 向けのプログラムを生成させる．また，データの転送や実行構成も pragma で指示する．hiCUDA を用いて記述したプログラムは，直接 CUDA プログラムを記述した場合とほぼ変わらないパフォーマンスを出す事が可能である．しかし，hiCUDA コンパイラに指示を出すために，新たに hiCUDA の pragma を学習する必要があるため，プログラマの負担は依然として大きい．

2.3 OpenMPC

2.2 節で述べたように，CUDA プログラミングの繁雑さを緩和する手法は多く提案されている．しかしながら，これらの手法では依然として GPU アーキテクチャや並列プログラミングの知識が必要である．また，CUDA のカーネル関数コールやデータ転送などの記法を隠蔽しているフレームワークを用いる場合，そのフレームワークの利用法を新たに学ぶ必要があるため，プログラマの負担は依然として大きい．

そこで，OpenMP[4] の記法を用いて GPU へ並列化の指示を出すことで，プログラマの学習コストを軽減しつつ，プログラマから CUDA を隠蔽するフレームワークとして OpenMPC[5] が提案されている．OpenMP は，既に広く利用されている CPU 向け並列化フレームワークであり，これを用いたプログラミングでは，プログラマはソースコード中の並列化したい箇所を pragma で指定する．そして，コンパイラが pragma を解釈し並列プログラムを生成する．OpenMP を用いる利点として逐次プログラムとのソースコード互換性がある．すなわち，OpenMP で並列化の指示をしたソースコードは，pragma を無視した場合，逐次プログラムのソースコードとほぼ同一である．そのため，プログラマは OpenMP を用いることで既存の逐次プログラムを容易に並列プログラムに変換できる．このように OpenMP では，pragma で指示された箇所を CPU で並列に実行することでプログラムを高速化するが，OpenMPC では，pragma で指示された箇所をカーネル関数として切り出し，そのカーネル関数を GPU で並列に実行させることで高速化を実現している．また，OpenMPC コンパイラは並列化を指示された箇所で使われている変数を自動的に検出し，GPU へと転送するコードを自動的に生成する．これにより，プログラマはどの変数を GPU へ転送する必要があるかを考える必要がなくなる．さらに，カーネル関数を実行する際の実行構成も，並列化を指示した箇所のイタレーション回数から自動的に決定される．そのため，OpenMPC を利用すれば，プログラマが GPU に関する知識を所有していなくとも，OpenMP の知識があれば GPU で動作するプログラムを作成する事ができる．

```
1  int main() {
2      int a[N][N];
3          :
4      #pragma omp parallel for collapse(2) //並列化の指示
5      for( i = 0 ; i < N ; ++i )
6          for( j = 0 ; j < N ; ++j )
7              for( k = 0 ; k < N ; ++k )
8                  a[i][j] += b[i][k] * c[k][j];
9  }
```

図 4: OpenMP を用いて記述した行列積の演算

```
1  __global__ void kernel_0(int *a , int *b , int *c ){ //デバイス・コード
2      int i = blockId.x;
3      int j = threadId.x;
4      int k;
5      for( k = 0 ; k < N ; ++j )
6          a[i][j] += b[i][k] * c[k][j];
7   }
8  int main() { //ホスト・コード
9      int a[N][N];
10     int *gpu_a;
11         :
12     cudaMalloc( &gpu_a , sizeof(int)*N*N );//デバイス・メモリの確保
13     //デバイス・メモリへのデータ転送
14     cudaMemcpy( gpu_a , a , sizeof(int)*N*N , cudaMemcpyHostToDevice );
15     //カーネル関数コール
16     kernel_0<<< dimGrid , dimBlock >>>( gpu_a , gpu_b , gpu_c );
17         :
18     cudaFree( gpu_a );
19         :
20 }
```

図 5: OpenMPC コンパイラによるプログラム変換例

ここで，行列積の演算を例に OpenMPC コンパイラがプログラムを変換する際の動作を示す．**図 4** ではプログラム内の for ループを OpenMP の `parallel for` 構文を用いて並列化している．ここで，`parallel for` は直後に続く for ループを並列化の対象とする構文である．また，`collapse` 指示句を使い，for ループの 2 層目までを並列化

対象としている (4 行目)．この指示句は parallel for 構文のオプションとして用いられる．通常，parallel for 構文は対象となる for ループが多重ループの場合，その最外ループだけを並列化の対象とするが，このオプションを用いると，その内部を考慮にいれて並列化する．

このプログラムを OpenMPC コンパイラを用いて変換すると**図 5** のようになる．OpenMPC コンパイラが変換を開始すると，まず parallel for 構文で並列化を指示された for ループを kernel_0 関数として切り出す (1 行目)．そして，main 関数中の for ループが存在していた箇所はその関数の呼び出しに変換される (14 行目)．このとき，実行構成は変数 dimGrid と変数 dimBlock で指定されるが，この 2 つの変数の値は，コンパイラが並列化の対象となった元ループのイタレーション回数から自動的に決定する．この例では，collapse 指示句によって並列化の対象はループの 2 層目までとなっているので，変数 dimGrid と変数 dimBlock の値はループ 2 層分のイタレーションの合計回数から決まる．

さらに，OpenMPC コンパイラによって，演算に使われている配列 a に格納されているデータをデバイス・メモリに転送するためのコードが main 関数側に追加される．具体的には，まずデバイス・メモリのアドレスを指すための変数 gpu_a が新たに宣言され (10 行目)，CUDA ライブラリ関数である cudaMalloc 関数によってデバイス・メモリ上の領域を確保し，その領域を指すようにする (12 行目)．その後，cudaMemcpy 関数によって変数 a に格納されているデータをホスト・メモリからデバイス・メモリに転送し GPU 側で変数 a に格納されたデータを参照可能にしている (13 行目)．同様に変数 b，c に対しても，デバイス・メモリ上の領域確保とデータ転送を行う．

一方，カーネル関数である kernel_0 関数では行列積の演算が実行される．ここで，カーネル関数を実行しているスレッドが変換前のループでの何番目のイタレーションを担当するのかを，変数 blockId.x と変数 threadId.x で表している (2，3 行目)．この 2 つの変数は CUDA のビルトイン変数であり，それぞれグリッド中の何番目のブロックであるかとブロック中の何番目のスレッドであるかを表している．そして，これらを変換前のイタレータ変数の代わりに用いることで配列内の要素にアクセスする (6 行目)．

OpenMPC はこのように動作し，既存の OpenMP プログラムをそのまま GPU 向け並列プログラムへと変換できる．そのため，プログラマに GPU アーキテクチャの知識を要求することなく，GPU による高速化の恩恵を受けられる．

3　適切なメモリの選択

本章ではOpenMPCが抱える問題点とその解決策について説明する．

3.1　OpenMPCの問題点

OpenMPCはOpenMPのソースコードを書き換えることなくGPUプログラムを生成できるが，そのままでは高速に実行可能なプログラムは生成できない．これは，単純にOpenMPCで変換しただけのコードが，次のような2つの問題点を抱えているからである．

1つめの問題点は，デバイス・メモリを自動的に使い分ける事ができない点である．2.1節で述べたように，CUDAではサイズやアクセス速度の異なる複数種類のデバイス・メモリが使用可能である．このため，GPUの性能を十分に引き出すためには，これらの使い分けが重要である．しかし，OpenMPCコンパイラは特に指定が無い場合，デバイス・メモリのうち，最もアクセス速度の遅いグローバル・メモリを選択する．そのため，メモリアクセスがボトルネックとなり，プログラムが高速に実行できない．この時，グローバル・メモリ以外のデバイス・メモリを使うためには，pragmaを用いることによって，使用するメモリを明示的に指定する必要がある．そのため，OpenMPCを利用する際にデバイス・メモリを使い分けるには，GPUアーキテクチャの知識が必要となり，このことはプログラマにとっての負担となる．

2つめの問題点は，ホスト・メモリ－デバイス・メモリ間のデータ転送の効率が悪いという点である．ホスト・メモリ－デバイス・メモリ間のデータ転送にはDMA転送が使われる．この時，転送されるデータはOSによってページアウトされない領域に配置されている必要がある．このページアウトされない領域の事をページロックメモリと呼ぶ．ここで，mallocなどで確保した領域や静的に確保された領域に配置されたデータは，デバイス・メモリへデータを転送する際に，ページロックメモリへとコピーされた後にデバイス側へと転送される．また，ホスト・メモリに書き戻される時も同様に，ページロックメモリから通常の領域へのコピーが発生する． この無駄なコピーを防ぐ為に，CUDAにはページロックメモリに対してデータ用の領域を直接確保するためのライブラリ関数が用意されている．このライブラリ関数を用いることによって，データ転送の際の無駄なコピーが発生しなくなるため，データ転送を効率的に行える．しかし，OpenMPCではこのライブラリ関数を使うためのインターフェースが用意されておらず，データ転送の際に，必ずページロックメモリへのコピーが発生する．そ

表 2: プログラマが明示的に使用可能なデバイス・メモリの特徴

種類	キャッシュ	アクセス	スコープ	アクセス速度	サイズ
グローバル・メモリ	されない	R/W	Grid と Host	低速	製品依存
シェアード・メモリ	-	R/W	Block	高速	16KB(SM 毎)
テクスチャ・メモリ	される	R	Grid と Host	高速	製品依存
コンスタント・メモリ	される	R	Grid と Host	高速	64KB

のため，ホスト・メモリ－デバイス・メモリ間のデータ転送が高速に行えない．したがって，OpenMPC により生成されたプログラムはデータの転送効率が良いとは言えない．

3.2 コンスタント・メモリの自動使用及びデータ転送効率化の提案

前述のような問題点を解決するため，本研究では高速にアクセス可能なデバイス・メモリの自動使用及び，CUDA ライブラリ関数を用いた転送用データ領域の確保を提案する．

3.2.1 コンスタント・メモリの自動使用

CUDA では複数種類のデバイス・メモリが利用できるが，それぞれが異なる特性を持っている．このため，GPU で高速に実行できるプログラムを記述するには，その使い分けが重要である．しかし，デバイス・メモリを使い分けるためにはプログラマが GPU アーキテクチャについて十分に理解している必要があり，この事はプログラマにとって負担となる．そこで，CUDA で利用可能な，高速にアクセス可能であるデバイス・メモリの自動使用を提案する．適切なデバイス・メモリを自動的に使い分ける事が可能となれば，プログラマの負担を増やすことなく，高速なプログラムが記述できる．CUDA で利用可能なデバイス・メモリのうち，プログラマが明示的に使用可能なものの特徴を**表 2** に示す．グローバル・メモリより高速にアクセス可能なメモリは，シェアード・メモリ，テクスチャ・メモリ，コンスタント・メモリの 3 種類がある．以下にその詳細を示す．

シェアード・メモリ

シェアード・メモリにはブロック単位でのみアクセス可能である．そして高速にアクセス可能な 3 種類のメモリのうち，唯一 GPU からの書き込みが可能である．また，各 SM 毎に 16KB ずつ割り当てられるため，ブロック毎の一時データを置くのに適している．ただし，シェアード・メモリにデータを書き込んだ場合，そ

のデータにアクセスする際は，ブロック内で同期を取る必要がある．

テクスチャ・メモリ

テクスチャ・メモリはグリッド単位でのアクセスが可能で，CPU からは読み書きが可能だが，GPU からは読み出しのみが可能である．また，使用可能なメモリサイズは製品に依存する．テクスチャ・メモリに格納されるデータはテクスチャと呼ばれ，1〜3 次元の配列のように表現される．テクスチャには，CUDA ライブラリに用意されている専用のアクセサ関数を介して，int もしくは float 型の添字を使って配列のようにアクセスする必要がある．ここで，float 型の添字でアクセスした場合には，テクスチャ中の要素と要素の中間値を取得することが可能である．また，テクスチャ・メモリにデータを配置する際には，格納する型とビット数，そして要素毎のチャネル数を規定したフォーマットを宣言する必要がある．ここで，チャネル数とは 1 つの要素が保持可能なデータ数の事を指す．格納可能な型は，符号なし整数，符号付き整数，浮動小数の 3 種のうちいずれかに制限される．そして，1 つのテクスチャには 4 チャネルまでのデータが格納可能で，各チャネルのビット数は，格納す る型が整数の場合は 0，8，16，32 のいずれか 1 つ，浮動小数の場合は 0，32 のどちらか片方を指定する必要がある．

コンスタント・メモリ

コンスタント・メモリはテクスチャメモリと同様に，グリッド単位でのアクセスが可能である．CPU からは読み書き可能であり，GPU からは読み出しのみが可能となっている．サイズは 64KB で，シェアード・メモリよりも大きい．テクスチャ・メモリとは異なり，配置するデータのフォーマットを宣言する必要がない．そして，データ量が 64KB に収まるならば，どのような種類のデータでも配置可能である．

今回は上記の 3 種類のメモリのうち，シェアード・メモリよりもサイズが大きく，テクスチャ・メモリよりも汎用性に優れているコンスタント・メモリを自動的に選択する手法を提案する．これにより，既存手法ではグローバルメモリに配置されていたデータの中から，コンスタント・メモリに配置可能なデータを自動的に検出し配置することが可能となる。コンスタント・メモリはグローバル・メモリよりも高速にアクセスが可能であり，さらにキャッシュも搭載している．そのため，頻繁にアクセスされるデータが配置されれば，カーネル関数の実行が大幅に高速化できると考えられる．

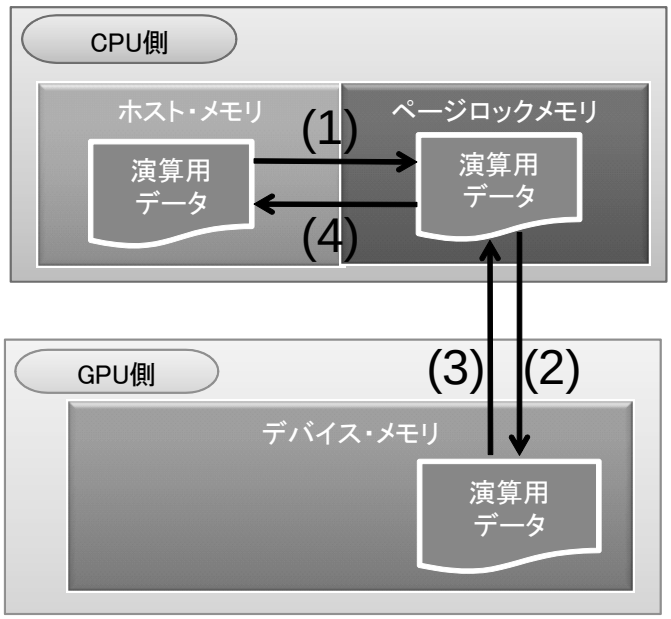

図 6: 既存手法でのデータ転送

3.2.2 CUDA ライブラリ関数を用いたデータ領域確保

ホスト・メモリ－デバイス・メモリ間で転送が行われるデータは，OS によってページアウトされることを防ぐために，ページロックメモリに配置されている必要がある．このため，ページロックメモリに配置されていないデータは，DMA 転送前にページロックメモリへとコピーされる．しかし，あらかじめページロックメモリへとデータが配置されていれば，このコピーは不要となる．CUDA には，ホスト・メモリ－デバイス・メモリ間でデータを効率良く転送するために，ページロックメモリへ直接データ領域を確保するためのライブラリ関数が用意されている．しかし，ページロックメモリは OS によってスワップアウトされないため，大量に確保しすぎると，システムが使える物理メモリの量が減ってしまい，システム全体のパフォーマンス低下を招いてしまう．そこで，デバイス・メモリへ転送されるデータを検出し，そのデータ用の領域のみを，このライブラリ関数を用いて確保するように OpenMPC コンパイラを改良することを提案する．

既存手法において，転送される変数は通常のメモリ領域に配置されるため，**図 6** に示すようにデータが転送される．まず，ホスト・メモリ中のページロックされていない通常のメモリ上に配置されたデータが，DMA 転送を行うためにホスト・メモリ中のページロックメモリへコピーされる (1)．そしてページロックメモリからデバイス・メモリへと DMA 転送される (2)．その後，カーネル関数の実行が終わり演算結果を CPU

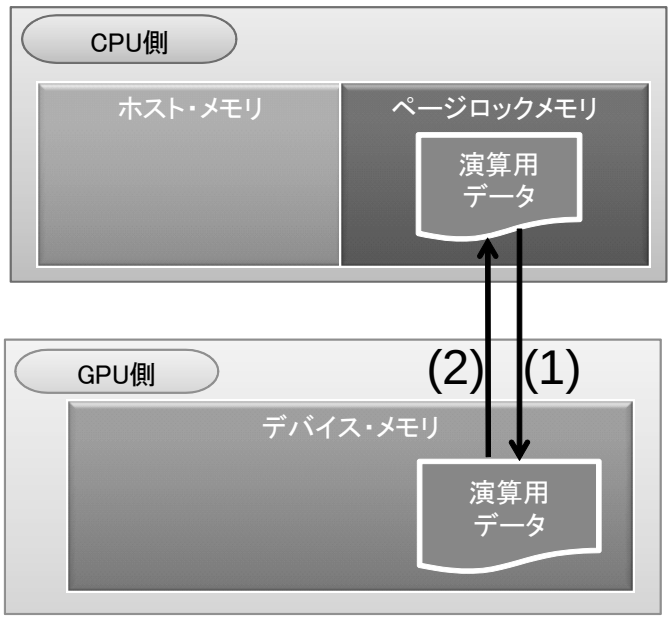

図 7: 提案手法でのデータ転送

が受け取る時も，デバイス・メモリからページロックメモリへと DMA 転送によってデータが転送され (3)，最後にページロックメモリから通常のメモリ領域上へとデータがコピーされる (4)．このようにホスト・メモリ－デバイス・メモリ間でデータが転送される度に通常のメモリとページロックメモリの間で無駄なコピーが発生する．この事は，データ転送の際にオーバーヘッドとなる．

一方，提案手法では，CUDA ライブラリ関数を用いて，あらかじめページロックメモリ上にデータ用の領域を確保するため，**図 7** に示すようにデータが転送される．この場合は，データが既にページロックメモリに配置されているため，図 6 の (1) や (4) のようなホスト・メモリ内での無駄なコピーが発生せず，デバイス・メモリへ直接転送される．このように，通常メモリとページロックメモリの間でのコピーがなくなるため，ホスト・メモリ－デバイス・メモリ間のデータ転送時間が削減され，プログラム全体の実行時間が短縮できる．

4　OpenMPC コンパイラの改良

提案手法を実現するために OpenMPC コンパイラを改良した．本章では改良したコンパイラでのコンパイルフローと提案手法の実装方法について述べる．

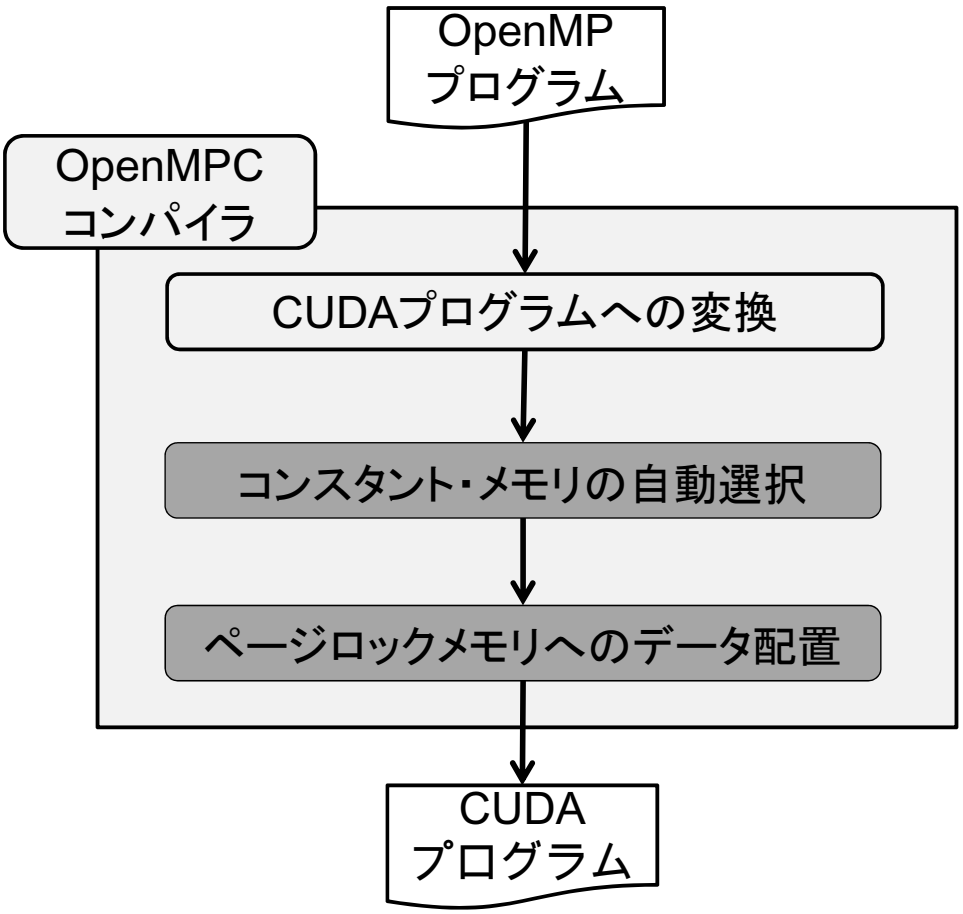

図 8: 提案手法でのコンパイルフロー

4.1 コンパイルフロー

OpenMPC は，CUDA プログラムへの変換時に詳細な指示を出すために使われる pragma と，その pragma と OpenMP の pragma を解釈して CUDA プログラムのソースコードを生成するコンパイラの 2 つで構成される．本研究では OpenMPC コンパイラを改良し，提案手法を実現する処理を追加実装した．**図 8** に改良したコンパイラでのコンパイルフローを示す．

改良したコンパイラでは既存のコンパイラが生成した構文木を解析し，GPU へ転送される変数や，デバイスコードの呼び出しを検出する．そして，それを基に提案手法を適用しプログラムの高速化をはかる．

```
1 void kernel_1(int *a,int *b,int *c , int *d) {
2     //アクセスするインデックスを作成
3     int i = threadId.x + blockId.x * blockDim.x;
4
5     a[i] = b[i] + c[i];//代入文
6     c[i] += ++ d[i]; //複合代入文とインクリメント
7 }
```

図 9: 代入及びインクリメント・デクリメント有無の判定

4.2 コンスタント・メモリの自動使用

表 2 で示したように，コンスタント・メモリは CPU からは読み書きが，GPU からは読み出しのみが可能である．また，一度に 64KB までのデータが配置可能である．そのため，コンスタント・メモリに配置可能なデータは，カーネル関数内で書き込みがされておらず，さらにデータサイズが 64KB 以下のものに限られる．また，コンスタント・メモリに配置するデータを格納する変数は，そのスコープがソースコード単位であり，1 つのソースコードでコンスタント・メモリへ配置するデータの合計サイズが 64KB を超えた場合はコンパイルエラーとなる．コンスタント・メモリに配置できる条件を満たす変数が複数存在する場合は，1 つのソースコード毎に合計のデータサイズが 64KB を越えなければ，条件を満たす全ての変数を検出した順にコンスタント・メモリへ配置する．以下に，条件を満たす変数の検出方法について述べる．

4.2.1 カーネル関数での書き込み有無の判定

GPU で実行されるスレッドはコンスタント・メモリへ書き込みが出来ない．そのため，コンスタント・メモリへ配置可能なデータは，GPU で実行されるカーネル関数内で，値が変更されないデータのみに限られる．ここで，値が変更されないデータとは，次のような変数に格納されているデータである．

- 代入が行われない変数
- インクリメント，デクリメントされない変数

代入が行われた変数は，値の変更結果を反映する必要があるため，コンスタント・メモリへ配置できない．また，インクリメント，デクリメントされた変数も加減算の結果をメモリへ反映する必要があるため，データをコンスタント・メモリへ配置できない．これに加え，カーネル関数内で他の関数の引数として使われる変数については，呼出先の関数内での書き込みの有無を判定出来ないため，判定対象から除外する．

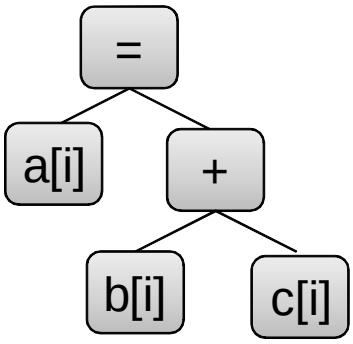

図 10: 代入文

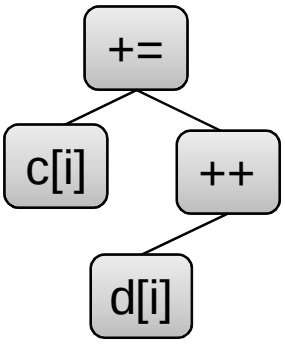

図 11: 複合代入文とインクリメント

図 9 に示すコードを例に，変数に対する代入及びインクリメント，デクリメントの有無の判定方法を説明する．図 9 の 5 行目から生成される構文木を**図 10** に，6 行目から生成される構文木を**図 11** に示す．

まず，図 10 を見ると変数 a が=演算子の左の子となっているため，変数 a は値が代入されることが分かる．そのため変数 a はコンスタント・メモリに配置する変数の候補から外れる．また，右の子からは変数 b，c に対してインクリメント，デクリメントされていないことが分かる．そのため，この式を解析し終えた段階では，まだ変数 b，c はコンスタント・メモリに配置可能な変数として見なされる．続いて，図 11 を見ると，変数 c が+=演算子の左の子となっていることが分かる．これにより，引数 c は値が代入される対象であることがわかるため，コンスタント・メモリに配置する変数の候補から外れる．さらに，右の子を見ると，変数 d に対してインクリメントがされていることがわかる．そのため，変数 d もコンスタント・メモリに配置する変数の候補から外れる．このように書き込みの有無を判定していくと，最終的にカーネル関数 `kernel_1` 内では，変数 b に対してのみ書き込みが行われていないことがわかる．よって，このカーネル関数では，変数 b として渡される引数の格納するデータのみがコンスタント・メモリへと配置されるデータの候補となる．

4.2.2 転送されるデータサイズの判定

続いて，変数のデータサイズの判定方法について述べる．転送されるデータサイズの判定には，変数の型と要素数を用いる．OpenMPC は動的に確保した領域を指すポインタ変数を用いたプログラムを変換できないため，静的な変数のみを対象とするからである．そのため，データサイズは静的に見積もることが可能である．コンスタント・メモリのサイズは 64KB であるため，転送されるデータサイズの合計が 64KB 以下である必要がある．

```
1  void Function_0() {
2      int x[256][256];
3      int y[1024];
4      int* gpu_x;
5      int* gpu_y;
6          :
7      //デバイス・メモリへのデータ転送
8      cudaMemcpy(gpu_x , x , sizeof(int)*256*256 , cudaMemcpyHostToDevice);
9      cudaMemcpy(gpu_y , y , sizeof(int) * 1024 , cudaMemcpyHostToDevice);
10
11     //カーネル関数コール
12     kernel_A<<< dimGrid , dimBlock >>>( gpu_x , gpu_y );
13         :
14 }
```

図 12: データサイズの判定

転送されるデータを格納している変数の探索を，**図 12** に示すサンプルプログラムを用いて説明する．このプログラムにおいて，カーネル関数 `kernel_A` は引数に `gpu_x` と `gpu_y` を与えられる (11 行目)．そのため，この 2 つの変数の指す領域へデータを転送している場所を探索する．その結果，`cudaMemcpy` 関数によって変数 `x` と `y` から `gpu_x` と `gpu_y` への転送がそれぞれ行われることが分かる．したがって，変数 `x` と `y` のデータサイズを見積もれば良い事が分かる (8，9 行目)．ここで，変数 `x` は型が int であり 256×256 個の要素を持つ二次元配列である (2 行目)．そのため，データサイズは `sizeof(int) × 256 × 256 = 256KB` と計算される．その結果，データサイズが 64KB を越えることが分かり，コンスタント・メモリに配置されるデータの候補から外れる．一方で，変数 `y` の型は変数 `x` と同じだが，配列の要素数は 1024 個である (3 行目)．したがって，データサイズは `sizeof(int) × 1024 = 4KB` と計算される．そのため，コンスタント・メモリへ配置される候補となる．よって，カーネル関数 `kernel_A` 内で 2 つめの引数に対して代入が行われないと判定されていれば，変数 `y` のデータはグローバル・メモリではなくコンスタント・メモリへ配置できる．

次に，**図 13** に示すようなサンプルプログラムにおける探索について説明する．この場合は，変数 `gpu_z` と `gpu_w` がカーネル関数 `Kernel_B` の引数となっている (11 行目) ため，この 2 つの変数が指す領域へデータを転送する関数を探索する．この時，`cudaMemcpy` 関数によって変数 `z` と `w` が保持するデータがデバイス・メモリへ転送されることがわ

```
1  void Function_1() {
2      float z[2048];
3      float w[2048];
4      float* gpu_z;
5      float* gpu_w;
6            :
7      //デバイス・メモリへのデータ転送
8      cudaMemcpy(gpu_z , z , sizeof(float)*2048 , cudaMemcpyHostToDevice);
9      cudaMemcpy(gpu_w , w , sizeof(float)*2048 , cudaMemcpyHostToDevice);
10     //カーネル関数コール
11     kernel_B<<< dimGrid , dimBlock >>>( gpu_z , gpu_w );
12 }
```

図 13: 複数の変数がコンスタント・メモリに配置可能な場合

かる (8,9 行目)．今回は，変数 z，w のサイズが共に sizeof(float) × 2048 の 8KB である．そのため，この 2 つのサイズを合計しても 16KB となり，コンスタント・メモリに配置可能なサイズの上限である 64KB 以下である．この時，カーネル関数側でデータの書き込みが行われていない場合は，変数 z，w が共にコンスタント・メモリに配置される．

4.3 CUDA ライブラリ関数を用いたデータ領域の確保

ホスト・メモリ－デバイス・メモリ間でデータを効率良く転送するために，転送されるデータが配置される領域を，CUDA ライブラリ関数を用いて確保するように変更する．この時，領域の確保方法が変更される変数がローカル変数の場合と，グローバル変数の場合で，領域を確保及び解放するタイミングが異なるため，それぞれの場合について説明する．

まず，領域の確保方法を変更する変数がローカル変数だった場合の例を**図 14** に示す．変換前のプログラムを見ると，関数 Function_0 のローカル変数である変数 a が cudaMemcpy 関数の引数となっており，ホスト・メモリ－デバイス・メモリ間でのデータ転送が行われる事がわかる (8 行目)．このため，この例では，変数 a が CUDA ライブラリ関数での領域確保の対象となることがわかる．その結果，変換後のプログラムでは，変数 a はポインタ変数として宣言され (2 行目)，CUDA ライブラリ関数である cudaMallocHost 関数を使ってデータ領域を確保するように変更されている (6 行目)．

```
1
2  int Function_0(){
3    int a[N];
4    int *gpu_a;
5      :
6      :
7    //デバイス・メモリへのデータの転送
8    cudaMemcpy(gpu_a,a,sizeof(int)*N);
9      :
10     :
11     :
12   return 0;
13 }
```

変換前のプログラム

```
1  int Function_0(){
2    int *a;//ポインタとして宣言
3    int *gpu_a;
4      :
5    //領域を確保
6    cudaMalloHost(&a,sizeof(int)*N);
7      :
8    //デバイス・メモリへのデータ転送
9    cudaMemcpy(gpu_a,a,sizeof(int)*N);
10     :
11   cudaFreeHost(a);//領域を開放
12   return 0;
13 }
```

変換後のプログラム

図 14: CUDA ライブラリ関数を用いたデータ領域確保の例 (ローカル変数)

さらに，ローカル変数は関数を抜けた時点で使用されなくなるため，cudaFreeHost 関数が関数 Function_0 の return 文の直前に挿入され，確保した領域が解放される (11 行目).

次に，転送される変数がグローバル変数だった場合の例を，**図 15** に示す．このサンプルプログラムでは，GPU へ転送される変数 b は関数 Funcion_1 のローカル変数ではなく，グローバル変数として宣言されている (1 行目)．そのため，プログラム中の様々な関数が変数 b にアクセスする可能性がある．そのため，関数 Funcion_1 で領域の確保と解放を行ってしまった場合，他の関数から変数 b へアクセスしようとしたときに，領域が確保されていないため，プログラムが正常に実行されない．そこで，GPU へ転送される変数がグローバル変数だった場合は，データ転送が行われる関数では無く main 関数で領域の確保と解放を行うように変更する (13,15 行目)．これにより，関数 Funcion_1 以外の関数から変数 b へのアクセスが発生した場合でも，プログラムは正常に実行される．

5 評価

既存の OpenMPC コンパイラと提案手法を実装した OpenMPC コンパイラを比較し，評価を行った．

```
1  int b[M];//配列として宣言
2
3  void Function_1()
4  {
5    int *gpu_b;
6        :
7    //デバイス・メモリへのデータ転送
8    cudaMemcpy(gpu_b,b,sizeof(int)*M);
9        :
10 }
11
12 int main(){
13       :
14       :
15       :
16   return 0;
17 }
```

変換前のプログラム

```
1  int *b; //ポインタとして宣言
2  void Function_1()
3  {
4    int *gpu_b;
5        :
6    //デバイス・メモリへのデータ転送
7    cudaMemcpy(gpu_b,b,sizeof(int)*M);
8        :
9  }
10 int main(){
11       :
12   //領域の確保
13   cudaMallocHost(&b,sizeof(int)*M);
14       :
15   cudaFreeHost(b);//領域の開放
16   return 0;
17 }
```

変換後のプログラム

図 15: CUDA ライブラリ関数を用いたデータ領域確保の例 (グローバル変数)

5.1 評価環境

評価環境を**表 3** に示す．GPU として NVIDIA 社の GeForce GTX280 を使用した．GeForce GTX280 は GT200 アーキテクチャを採用しており，30 個のストリーミング・マルチプロセッサ (SM) を搭載している．さらに，各 SM 上にはそれぞれ 8 個のストリーミング・プロセッサ (SP) が搭載されており，計 240 個の SP を持つ．

また，評価には行列積の演算を行う MatrixMul，モンテカルロ法を用いて円周率の計算を行う PI (montecarlo)，積分法を用いて円周率の計算を行う PI (Integral)，マンデルブロー集合の面積の推定を行う Mandelbrot，ヤコビ法を用いてヘルムホルツ方程式を解く Jacobi の 5 種類のプログラムを用いた．このうち，PI (Integral)，Mandelbrot，Jacobi は OMPSCR[6] に含まれるベンチマークプログラムである．なお，MatrixMul で演算に用いた行列のサイズは 128×138 であり，PI (montecarlo) での計算の試行回数は 10 万回である．また，PI (Integral)，Mandelbrot は引数に OMPSCR のデフォルト値を用いたが，Jacobi のみ計算に用いる行列サイズを 200×200 としている．これはデ

表 3: 評価環境

OS	Fedora15
CPU	Core2Quad
Frequency	2.83GHz
Memory	3GB
GPU	GeForce GTX280
Number of multiprocessors	30
Number of cores (SP)	240 (30 × 8)
CUDA version	4.0
Compute capability	1.3
Compiler	gcc 4.6.1
Compile options	-O3
OpenMPC Compiler version	0.2

フォルトでは 5000×5000 の行列を用いていたが，このサイズでは既存の OpenMPC コンパイラが変換したプログラムが実行できなかったためである．

5.2 評価結果

評価結果を**図 16** に示す．グラフは，左が既存の OpenMPC コンパイラが出力したプログラム，右が提案手法を適用したコンパイラが出力したプログラムの実行時間を表している．グラフの縦軸は実行時間を表しており，既存手法の実行時間を 1 として正規化している．プログラムの実行時間を，ホスト・メモリからデバイス・メモリへのデータ転送，カーネル関数での計算時間，デバイス・メモリからホスト・メモリへのデータ転送，その他の 4 項目に分類し，それぞれに要した時間をプロットした．その他には，GPU 側のデータ領域の確保と開放，実行構成の設定などが含まれている．

評価の結果，最大で 42%, 平均で 14%の高速化が確認できた．項目別では，デバイスメモリからホスト・メモリへのデータ転送は最大 14%，平均 7%の高速化が，カーネル関数の計算時間は最大 25%，平均 7%の高速化が，デバイス・メモリからホスト・メモリへのデータ転送は最大 3%平均 2%の高速化がそれぞれ確認できた．

特に Jacobi は他のプログラムと比べて，高速化が顕著に見られる．これは Jacobi は他のプログラムと異なり，カーネル関数とそれに伴うデータ転送が複数回実行されたためであると考えられる．

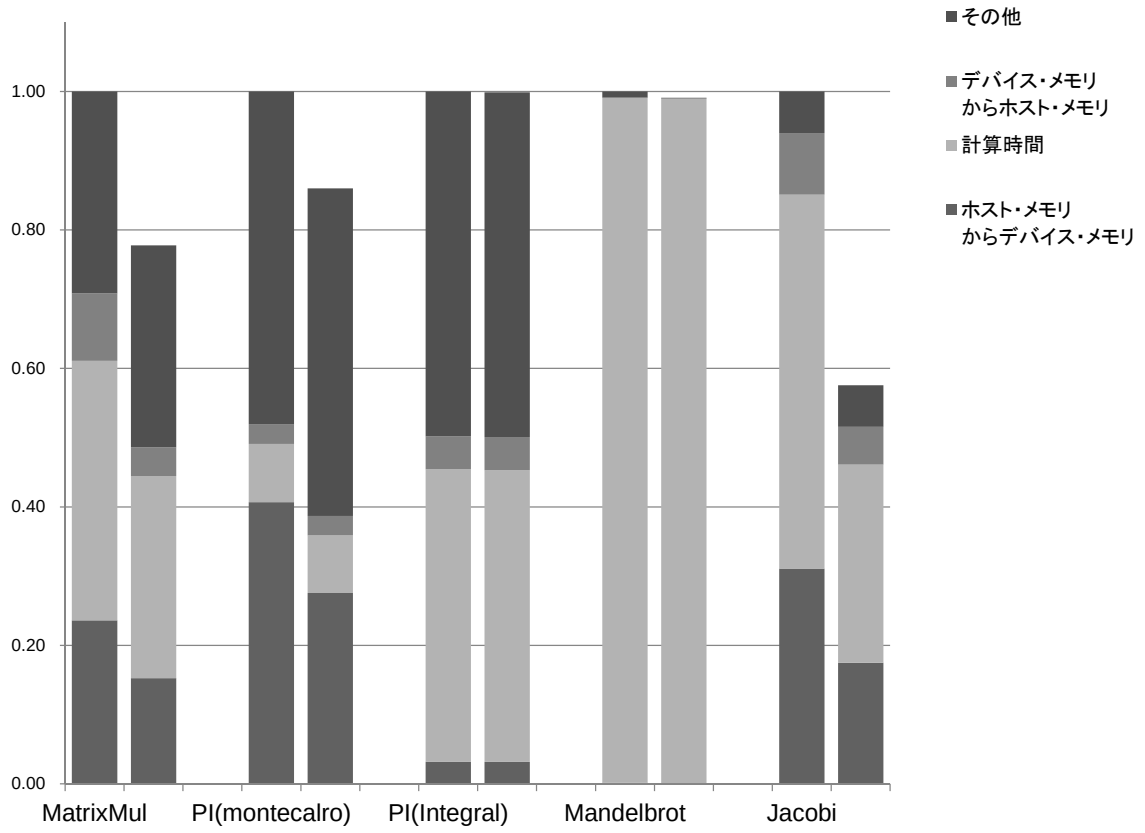

図 16: 評価結果

5.3 考察

評価結果を考察したところ，結果に 3 つの傾向があることが明らかになった．まず，データ転送の時間及びカーネルの実行時間の両方で高速化が確認できたものが挙げられる．このパターンには MatrixMul と Jacobi の 2 つが該当する．この 2 つのプログラムでは，カーネル関数を実行する際に必要となるデータの中に，大きな配列があったことが転送時間の削減に大きく関係している．そのため，提案手法により大きなデータのコピーが削除され，データの転送時間が削減されたと考えられる．そして，計算結果を格納するための変数も，ある程度サイズが大きい配列だったため，デバイス・メモリからホスト・メモリにデータを転送する時間も削減されている．さらに，カーネル関数内でコンスタント・メモリへ配置されたデータへのアクセスが頻繁に行われたため，カーネル関数の実行が高速化されたと考えられる．

次に，カーネル関数での計算時間はあまり変化がなく，データの転送時間のみが削減されたものが挙げられる．このパターンに該当した物は PI (montecarlo) と Mandelbrot

の 2 つがある．この 2 つのプログラムは共に，アルゴリズムとしてモンテカルロ法 [7] を用いている．モンテカルロ法はランダムな値を採択するかどうかを繰り返し，近似解を求めるアルゴリズムである．そのため，アルゴリズムを実行する際に乱数が必要となる．ここで，GPU では rand 関数を使うことが出来ないため，ランダムな値が必要な際は，事前に配列などで用意しそれを GPU へと転送する必要がある，と言う事実がある．つまり，この 2 つのプログラムでは，事前に用意された乱数列をデバイス・メモリに転送する際に，提案手法であるページロックメモリにデータを配置する手法の効果が確認できたと考えられる．ただし，デバイス・メモリからホスト・メモリへのデータ転送については，計算結果だけを転送するため，転送されたデータサイズが小さく，提案手法による効果が得られなかったと考えられる．また，この 2 つのプログラムでは カーネル関数は値を採用するかどうかの比較と，リダクション演算のみを行っていた．そのため，カーネル関数での処理は元々時間がかかるものではない上に，コンスタント・メモリへ配置したデータへのアクセスはカーネル関数内でのごく一部となり，計算時間は短縮されなかったと考えられる．

最後に，提案手法と既存手法で，実行時間に有意な変化がなかった物が挙げられる．このパターンには PI (Integral) が該当する．このプログラムは OpenMPC で変換を指定した for ループで，ループのイタレータとループ内での局所変数のみを演算に使用していた．そのため，変換後のカーネル関数に対して転送されたデータ量が少量であり，データ転送の効率化が有効ではなかったと考えられる．また，コンスタント・メモリへのデータ転送も行われたが，コンスタント・メモリに配置されたデータへのアクセス回数が少ないため，高速化がほぼ確認できなかったと考えられる．そのため，このような GPU 側でデータを用意し演算をするプログラムでは提案手法が有効ではないと考えられる．

6　おわりに

本論文では，CUDA 向け並列化フレームワーク OpenMPC を改良し，適切なデバイス・メモリを自動的に選択する手法を提案した，また，デバイス・メモリへと転送される変数を検出し，自動的にページロックメモリへと配置する手法を提案した．提案手法が有効であることを確認するため，幾つかのベンチマークプログラムを用いて評価した．その結果，既存手法に対して，平均で 14 パーセント，最大で 42 パーセントの速度向上が確認できた．

今後の課題として，コンスタント・メモリに配置するデータの選択を，より適切に

行うことが挙げられる．現在の実装では，コンスタント・メモリに配置可能なデータが複数ある場合，最初に検出された物から優先的に配置するようにしている．そのため，カーネル関数内で頻繁にアクセスされる変数よりも，されない変数の方が優先的に選択される可能性がある．したがって，コンスタント・メモリに配置可能なデータが複数存在した場合に，頻繁にアクセスされるデータを優先的に配置すれば，メモリアクセス時間がより短縮できると考えられる．さらに，より多くの種類のデバイス・メモリを自動選択できるようにすることも課題として挙げられる．今回は，コンスタント・メモリのみを自動選択の対象としたが，高速にアクセス可能なデバイス・メモリには，テクスチャ・メモリとシェアード・メモリも存在する．これらのメモリも，状況に応じて適切に選択可能にすることで，よりプログラマの負担が軽減されると考えられる．

謝辞

本研究のために，多大な御尽力を頂き，御指導を賜わった名古屋工業大学の松尾啓志教授，津邑公暁准教授，齋藤彰一准教授，松井俊浩准教授に深く感謝致します．また，本研究の際に多くの助言，協力をして頂いた松尾・津邑研究室および齋藤研究室の方々に深く感謝致します．特に，研究に関して貴重な意見を下さった今井満寿巳氏，近藤勝彦氏，稲葉崇文氏，小野亜実氏に感謝致します．

参考文献

[1] NVIDIA Corp.: *NVIDIA CUDA Programming Guide*, 2.0 edition (2008).

[2] 道浦悌, 大野和彦, 佐々木敬泰, 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論文集, Vol. 2011, pp. 221–222 (2011).

[3] Han, T. D. and Abdelrahman, T. S.: hiCUDA: High-Level GPGPU Programming, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 22, pp. 78–90 (2011).

[4] Dagum, L. and Menon, R.: OpenMP: an Industry Standard API for Shared-Memory Programming, *IEEE Computational Science and Engineering*, Vol. 5 (1998).

[5] Lee, S. and Eigenmann, R.: OpenMPC: Extended OpenMP Programming and Tuning for GPUs, *SC'10: Proceedings of the 2010 ACM/IEEE conference on Supercomputing*, IEEE press (2010).

[6] Dorta, A., Rodriguez, C. and de Sande, F.: The OpenMP source code repository, *Parallel, Distributed and Network-Based Processing, 2005. PDP 2005. 13th Euromicro Conference on*, pp. 244 – 250 (2005).

[7] 宮武修, 中山隆: モンテカルロ法 (1960 年), 日刊工業新聞社 (1960).