修士論文

題目

マルチGPU上の
効率的な実行手法のための
コード生成機構

指導教員

大野　和彦　講師

平成28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

山本　怜 (415M519)

内容梗概

GPGPU(General Purpose computation on Graphics Processing Units) はその高い計算性能により様々な分野から注目を集めている. しかし, 現在主流の GPGPU 開発環境である CUDA は個々の GPU 性能を引き出すために手動チューニングが必要であり, ユーザの大きな負担となる.

我々が開発する MESI-CUDA は単純なメモリモデルによる GPGPU プログラミングが可能で, ユーザの追加負担なしに自動最適化を行う. MESI-CUDA 処理系はコンパイラとランタイムによる動的タスクスケジューリング機構を導入しており, マルチ GPU への動的負荷分散や分散メモリデータのキャッシュ化を実現している. コンパイラはデータ転送・スレッド起動を独立デバイス上で分割実行するルーチンをコード生成することで本機構に寄与しているが, 従来の実装では実行効率が低下する場合がある.

本研究では, MESI-CUDA コンパイラにおけるコード生成機構を拡張することで, 動的タスクスケジューリング機構の改良を行った. 従来のデータ転送ルーチンでは分割実行に必要なメモリ領域単位で転送が行われるため, 局所的なメモリデータを分散する等の操作をランタイムから実行できない. そのため分散メモリデータ間に依存関係がある場合においても, ランタイムはより最適なメモリデータ分割によるキャッシュ管理が不可能である. この問題に対し, コンパイラはランタイムが要求する分割実行粒度に付随するメモリ領域をパラメータとして取得できるルーチンを新たにコード生成する. ランタイムはこのルーチンを活用することで分散メモリデータ間の依存検出が可能となり, 最適分割されたメモリデータのキャッシュ管理が可能となる.

三重大学大学院　工学研究科

Abstract

GPGPU(General Purpose computing on Graphics Processing Units) gets attention from various fields because of high computational performance. However, major developing environment, such as CUDA requires hand tuning to exploit individual GPUs performance. Thus, the users must devote coding effort to optimize GPGPU programs using CUDA.

We are developing MESI-CUDA that enables to programming GPGPU programs on a simple memory model and optimizes automatically without additional burden. MESI-CUDA framework introduces dynamic task scheduling scheme consists of compiler and runtime, and implements dynamic load balancing and distributed data caching. The compiler generates subroutine codes that execute split data transfer and threads invocation on an independent device to realize this scheme. However, the conventional implementation causes inefficient execution because the compiler generates not optimized codes in some cases.

In this research, we extend the code generation scheme in the compiler to improve the dynamic scheduling scheme. The conventional subroutine performs data transfer just as needed for split execution. Thus, the runtime can't execute operations such as partial memory data distribution using the subroutine. Therefore, even when there is a dependency relation between distributed memory data, cache management by optimum memory partitioning is impossible. To solve this problem, the compiler generates a new subroutine that the runtime can acquire the memory region associated with the granularity of split execution as a parameter. The runtime can detect dependency between distributed memory data and manage optimally partitioned memory data using this subroutine.

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(Graphics Processing Unit) は CPU に比べめざましい演算性能の向上を見せている. そのため, GPU を本来の用途である画像処理以外の汎用的な計算に用いる GPGPU(General Purpose computation on GPUs)[1] はその計算性能により様々な分野から注目を集めている. 現状利用されている GPGPU 開発環境として CUDA[2] や OpenCL[3] が存在する. しかし, これらの開発環境は GPU の性能を引き出す上で低レベルのコーディングを要求するため, ユーザの負担が大きい問題がある. また, CPU(ホスト)・GPU(デバイス) はそれぞれ自身のみがアクセスできるホストメモリ・デバイスメモリを搭載する. ユーザはデバイスを扱う上でホストメモリ・デバイスメモリ間のデータ転送をプログラム中に記述する必要があり, ハードウェア構成を意識したプログラミングが求められる. また複数の GPU を一台のマシンに搭載するマルチ GPU 環境では, 個々のデバイスの負荷状況や転送データを考慮したスケジューリングが必要になる等, ユーザは更なる負担を強いられる.

我々は CUDA より簡単にプログラミング可能な GPGPU フレームワーク MESI-CUDA[10] を開発している. MESI-CUDA は仮想共有変数メモリモデルによるプログラミングを可能にしており, ホスト・デバイス間のデータ転送の記述は不要となる. また, マルチ GPU 環境においても個々のデバイスをユーザに隠蔽したまま複数のデバイスを自動利用する. そのため, ユーザは単一の高性能なデバイスによるプログラミングができる一方で, マルチ GPU による高速化の恩恵を自動的に得ることができる.

MESI-CUDA 処理系は更なるマルチ GPU 対応に向けて開発を進めており, 動的負荷分散や各デバイスへの転送データのキャッシュ化により実行効率を高めている. しかしホスト・各デバイス間のメモリ管理の最適化が不十分であるため, アプリケーションによっては手動最適化した CUDA コードに比べ性能が低下する場合がある. 本研究はマルチ GPU 対応に向けて処理系に導入した動的タスクスケジューリング機構を機能拡張することでマルチ GPU 対応のメモリ管理手法を提案する.

以下, 2 章では背景として CUDA とマルチ GPU について解説し, 3 章で MESI-CUDA のプログラミングモデルと機能について説明する. 4 章では本研究で提案するメモリ管理手法と適用例を示し, 5 章で MESI-CUDA コンパイラにおけるコード生成機構の実装を解説する. 6 章では提案手法による自動最適化と手動最適化で CUDA プログラムの実行時間を比較した結果を示し, 7 章で関連研究を紹介する. 最後に, 8 章でまとめを行う.

三重大学大学院　工学研究科

2 背景

2.1 CUDA

CUDA はnVIDIA 社が提供するコンパイラ・ライブラリを含めたGPGPU 統合開発環境であり, ユーザはC/C++を拡張した文法とライブラリ関数を用いてCUDA プログラムを開発する. CUDA プログラミングモデルを図2.1 に示す. CUDA において, CPU 側はホスト, GPU 側はデバイスと呼ぶ. デバイスはPCI-Express を通じてホストにより制御され, ホストから与えられる計算処理を数千個のCUDA コアで並列実行する. ホスト・デバイスの各CPU コア・CUDA コアは図2.1 に示すように, 自身が接続するホストメモリ・デバイスメモリにのみそれぞれアクセスする. ホストメモリ・デバイスメモリ間のデータ転送はユーザ自身がCUDA ライブラリ関数を用いて記述する必要がある.

ステップシミュレーションを実行するCUDA コードを図2.2 に示す. ユーザはデバイス上で動作する処理 (カーネル) を関数として記述する (図2.2:7-19 行, 20-24 行). カーネル起動はCUDA 独自の記法により, スレッドの集まりであるスレッドブロックの数・サイズを指定してスレッド群を生成する (図2.2:41 行, 42 行). カーネル起動時に生成される各スレッドの動作はビルトイン変数を用いて記述する (図2.2:8-9 行, 21-22 行). ホストメモリ・デバイスメモリの記述はまずホスト・デバイス毎にメモリを確保する (図2.2:5 行, 31-33 行). 次に, ホストメモリに初期化されたデータのデバイスメモリへの転送 (download 転送) をカーネル起動前に行う (図2.2:34-37 行). 最後に, デバイスメモリ上の書き込みデータをホストメモリに転送し (readback 転送), カーネル実行結果をホストから読み込む (図2.2:44-45 行).

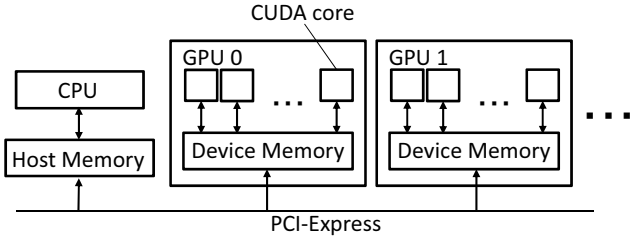

図 2.1: CUDA プログラミングモデル

```
 1 #define N 8192
 2 #define STEP_ITERATION 1000
 3 #define BS 256
 4 #define S (sizeof(double)*N*N)
 5 double h_power[N][N], h_temp[N][N], h_result[N][N];
 6 double *d_power, *d_temp, *d_result;
 7 __global__ void stencil(double result[][N], double temp[][N], double power[][N],
                          double Cap, double Rx, double Ry, double Rz){
 8   int x = blockDim.x * blockIdx.x + threadIdx.x;
 9   int y = blockDim.y * blockIdx.y + threadIdx.y;
10   int n = (y > 0) ? y-1 : y;
11   int s = (y < N - 1) ? y+1 : y;
12   int w = (x > 0) ? x-1 : x;
13   int e = (x < N - 1) ? x+1 : x;
14   double tx = (temp[y][w] + temp[y][e] - 2.0 * temp[y][x]) * Rx;
15   double ty = (temp[n][x] + temp[s][x] - 2.0 * temp[y][x]) * Ry;
16   double tz = (AMB_TEMP - temp[y][x]) * Rz;
17   double delta = Cap * (power[y][x] + tx + ty + tz);
18   result[y][x] = temp[y][x] + delta;
19 }
20 __global__ void copy(double temp[][N], double result[][N]){
21   int x = blockDim.x * blockIdx.x + threadIdx.x;
22   int y = blockDim.y * blockIdx.y + threadIdx.y;
23   temp[y][x] = result[y][x];
24 }
25 void read_input(double data[][N]){...}
26 void write_output(double data[][N]){...}
27 void set_scala(double *Cap, double *Rx, double *Ry, double *Rz){...}
28 int main(int argc, char **argv){
29   int i;
30   double cap, rx, ry, rz;
31   cudaMalloc(&d_power, S);
32   cudaMalloc(&d_temp, S);
33   cudaMalloc(&d_result, S);
34   read_input(h_power);
35   read_input(h_temp);
36   cudaMemcpy(d_power, (double *)h_power, S, cudaMemcpyHostToDevice);
37   cudaMemcpy(d_temp, (double *)h_temp, S, cudaMemcpyHostToDevice);
38   set_scala(&cap, &rx, &ry, &rz);
39   for (i = 0; i < STEP_ITERATION; i++){
40     dim3 grid(N/BS, N);
41     stencil<<<grid, BS>>>((double(*)[N])d_result, (double(*)[N])d_temp, (double(*)[N])d_power,
                            cap, rx, ry, rz);
42     copy<<<grid, BS>>>((double(*)[N])d_temp, (double(*)[N])d_result);
43   }
44   cudaMemcpy((double *)h_result, d_result, S, cudaMemcpyDeviceToHost);
45   write_output(h_result);
46   cudaFree(d_power);
47   cudaFree(d_temp);
48   cudaFree(d_result);
49 }
```

図 2.2: ステップシミュレーションを実行する CUDA コード

2.2 マルチ GPU

複数デバイスの並列動作が可能であるマルチ GPU 環境においては, 個々のデバイスを適切に利用することで更なる高性能化が見込める. CUDA では, データ転送やカーネル起動の対象デバイスをコード上で切り替えることでマルチ GPU 対応が可能である. しかし, 高性能の実現には無駄なアイドル時間や冗長なデータ転送を削減するコードを記述する必要があり, チューニングにかかるユーザの負担が大きい問題がある. また, そのチューニングパラメータは実行環境の各デバイス性能に強く依存するため, コード移植性の低さが問題である.

3 MESI-CUDA

3.1 MESI-CUDA 概要

MESI-CUDA は CUDA より簡単に GPGPU プログラムを記述できるプログラミングフレームワークである. MESI-CUDA プログラミングモデルを図 3.3 に示す. MESI-CUDA は仮想共有メモリモデルを採用しており, ユーザは図 3.3 に示すホスト・デバイス両方からアクセス可能な仮想共有メモリを利用できる. また, 個々のデバイスを隠蔽した単一ホスト・単一デバイスの実行モデルを提供し, ホスト・デバイスへの処理割当てやカーネルの記述は CUDA と同様に記述する. MESI-CUDA 処理系はこのプログラミングモデルを維持したまま実行環境に応じて自動最適化を行う.

図 2.2 の CUDA コードと等価な MESI-CUDA コードを図 3.4 に示す. `__global__`修飾子を付けて宣言する配列変数を仮想共有変数と呼ぶ (図 3.4:3 行). 仮想共有変数はホスト・デバイス双方からの参照及びカーネル関数への引数渡しが可能で, デバイスメモリの確保・開放やデータ転送の記述は不要となる (図 3.4:28-29 行, 32-33 行, 35 行). MESI-CUDA 処理系は各仮想共有変数のメモリアクセスを解析し, その解析情報を元に必要なデータ転送を行う CUDA コードを自動生成する. また処理系は論理スレッドマッピングにより, 各スレッドの計算資源への割り当てを自動的に行う (図 3.4:5-6 行, 18-19 行).

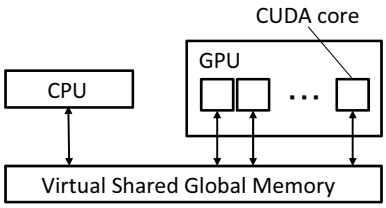

図 3.3: MESI-CUDA プログラミングモデル

3.2 動的タスクスケジューリング機構

動的タスクスケジューリング機構はマルチ GPU 対応に向けて MESI-CUDA 処理系に導入した機構である [11]. 本機構は CUDA カーネルのスレッドブロックにおけるメモリアクセスの性質を利用してカーネルの分割実行を実現する. 図 3.5 に分割スレッドブロックのメモリアクセス例を示す. 一般的なカーネルの生成スレッド群において, 同一カーネルのスレッドブロック間にはデータ依存関係が存在しない. ゆえに, 1 回のカーネル起動をスレッドブロック単位で複数回に分けて起動してもプログラムの意味は変化しない. そこで, 図 3.5 に示すような分割スレッドブロックとそれに付随する部分メモリデータを任意のデバイスに割り当てることで, 独立デバイス上で分割カーネルの実行が可能となる.

本機構はカーネル起動時のスレッドブロック及びメモリデータを動的に分割し, それらを複数デバイスの計算資源にスケジューリングすることで実行効率を向上させる. これらは予め実装したランタイムルーチンをコンパイラが解析情報と共にユーザコードへ挿入することで実現する. MESI-CUDA 処理系内において, ユーザが記述するカーネル起動をジョブ, ジョブを分割したものをタスクと呼ぶ. タスクはジョブを任意のスレッドブロック数で分割した実行単位であり, 単一デバイス上で独立に実行できる必要がある. そこでコンパイラは解析情報を元に, タスクに割り当てられたスレッドブロック情報を引数として必要な download/readback 転送と分割カーネル起動を行う関数 (タスク関数) をジョブ毎に生成する. ランタイムはこのタスク関数を実行するデバイスを負荷状況や転送データに応じて適切に選択することで, マルチ GPU 環境におけるタスクスケジューリングが可能となる.

```
 1 #define N 8192
 2 #define STEP_ITERATION 1000
 3 __global__ double g_power[N][N], g_temp[N][N], g_result[N][N];
 4 __global__ void stencil(double result[][N], double temp[][N], double power[][N],
                          double Cap, double Rx, double Ry, double Rz){
 5   int x = lthreadIdx.x;
 6   int y = lthreadIdx.y;
 7   int n = (y > 0) ? y-1 : y;
 8   int s = (y < N - 1) ? y+1 : y;
 9   int w = (x > 0) ? x-1 : x;
10   int e = (x < N - 1) ? x+1 : x;
11   double tx = (temp[y][w] + temp[y][e] - 2.0 * temp[y][x]) * Rx;
12   double ty = (temp[n][x] + temp[s][x] - 2.0 * temp[y][x]) * Ry;
13   double tz = (AMB_TEMP - temp[y][x]) * Rz;
14   double delta = Cap * (power[y][x] + tx + ty + tz);
15   result[y][x] = temp[y][x] + delta;
16 }
17 __global__ void copy(double temp[][N], double result[][N]){
18   int x = lthreadIdx.x;
19   int y = lthreadIdx.y;
20   temp[y][x] = result[y][x];
21 }
22 void read_input(double data[][N]){...}
23 void write_output(double data[][N]){...}
24 void set_scala(double *Cap, double *Rx, double *Ry, double *Rz){...}
25 int main(int argc, char **argv){
26   int i;
27   double cap, rx, ry, rz;
28   read_input(g_power);
29   read_input(g_temp);
30   set_scala(&cap, &rx, &ry, &rz);
31   for (i = 0; i < STEP_ITERATION; i++){
32     stencil<[<N, N>]>(g_result, g_temp, g_power, cap, rx, ry, rz);
33     copy<[<N, N>]>(g_temp, g_result);
34   }
35   write_output(g_result);
36 }
```

図 3.4: 図 2.2 の CUDA コードと等価な MESI-CUDA コード

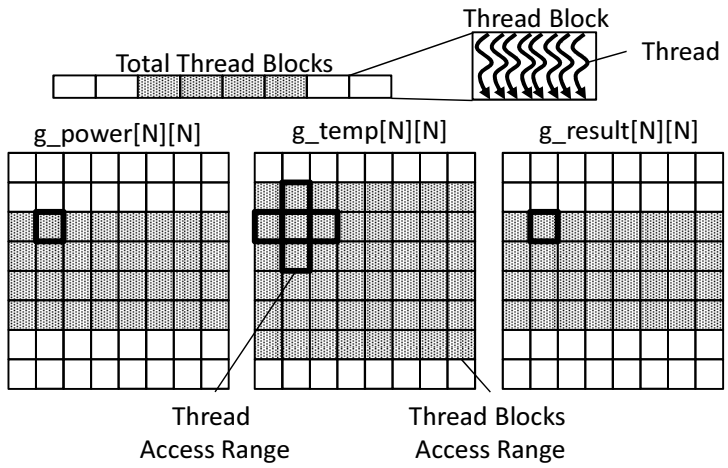

図 3.5: 分割スレッドブロックのメモリアクセス例

3.3 従来の処理系の問題点

従来のタスク関数は安全なデバイス独立実行を保証するために, download/readback 転送をタスク実行毎に行う. これによりタスク実行時は常にデータ転送が発生するため, アプリケーションによっては無視できない通信オーバヘッドが発生する. 我々は MESI-CUDA 処理系の改良を進め, 各デバイスへの転送データをキャッシュ化して再利用し, さらにそのキャッシュ利用率を高めるスケジューリングを実装することでデータ転送量の削減を行った [12]. しかし, 手動最適化した CUDA プログラムと同等の性能を実現するには, 更なる処理系の改良によりデータ転送量を最小化する必要がある.

図 3.6 に示すステンシル計算のタスク実行を具体例として, 処理系が解決すべき問題を説明する. ステンシル計算は図 2.2 及び図 3.4 のコード例にも見られる計算パターンで, ステップシミュレーションに多く利用されている. ステンシル計算においては, 各スレッド間でメモリ読み込み範囲が排他的にならない性質が存在する. そのためステンシル計算を行うカーネルのジョブを複数デバイスでタスク実行する場合, 図 3.6 の点線矢印で示す範囲の download 転送を各デバイスで行う必要がある. タスク s, t の各 download 転送のメモリ範囲は重複領域が存在するため, 図 3.6 の斜線部で示された自身とは別のタスクにより書き込まれるメモリ領域 (袖領域) が各デバイス上で生じる. この時, 各タスク自身が書き込んだメモリ範囲を readback 転送するのではなく, 図 3.6 の実線矢印で示す袖領域の交

換をタスク間で行う方が最適である. しかし従来の動的タスクスケジューリング機構では, 袖領域交換のように局所的な部分メモリデータを各デバイスに分散させる転送を自動的に行うことができない.

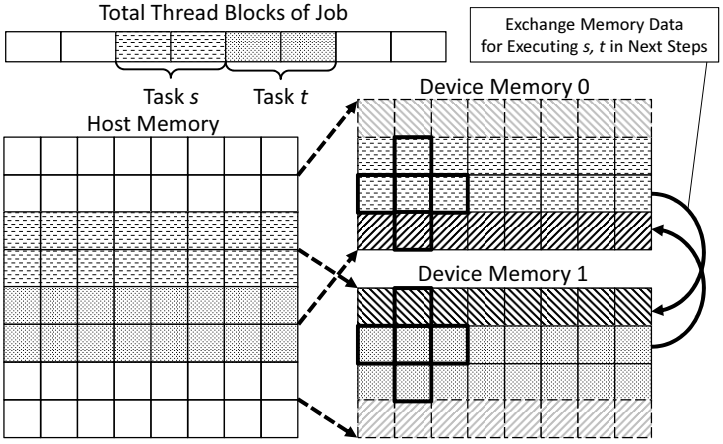

図 3.6: ステンシル計算のタスク実行例

4 提案手法

4.1 概要

動的タスクスケジューリング機構の機能拡張により, メモリアクセスに応じた仮想共有メモリの最適分割に基づくメモリ管理手法を提案する. すべての仮想共有変数及びジョブに対しタスク間のメモリアクセス範囲の重複を網羅的に検出し, その結果得られる個々の重複範囲の粒度で仮想共有メモリを分割する. そして, その分割メモリ領域単位でホスト・各デバイス間の分散状況を管理することで, タスク実行に必要なメモリ領域内における部分メモリデータのキャッシュ化が可能となる.

従来のメモリ管理手法はタスク実行に必要なメモリ領域単位でデータ転送が行われる都合上, 転送時の固定長データサイズでしかメモリ管理ができない. それに対し, 提案手法はメモリ管理情報からタスク実行対象のデバイス上に必要なメモリデータを集約する形式でデータ転送を行うため, 袖領域交換のような転送量が最小となるデータ転送を実現できる.

4.2 メモリ管理手法

仮想共有メモリの最適分割に基づくメモリ管理手法の概要を図 4.7 に示す．ある仮想共有変数を参照する任意の 2 タスクがそれぞれ実行に必要な連続メモリ領域を u, v とする．u, v は図 4.7 に示すように, 仮想共有変数に対応するホスト変数の先頭要素へのポインタとバイト数のパラメータ組で表現する．また, 仮想共有変数に対するメモリ書き込み/読み込みの有無を示す属性 (Read/Write 属性) をタスク (ジョブ) は保持する．この時, u, v の先頭要素へのポインタとバイト数をそれぞれ u_{start_ptr}, u_{size}, v_{start_ptr}, v_{size} とすると, u, v が重複する連続メモリ領域 $u \cap v$ は以下の積集合演算で求めることができる．

$$[u_{start\_ptr}, u_{start\_ptr} + u_{size}] \cap [v_{start\_ptr}, v_{start\_ptr} + v_{size}]$$

そして, $u \cap v$ のメモリ領域を u と v から独立したデータセグメントとして使用する．これにより, ホストメモリ・各デバイスメモリ間のデータ分散状況を任意のサイズに分割したメモリ領域単位で管理できる．タスク実行時は 1 つ以上のデータセグメントを任意のデバイスメモリ上に集約して分割カーネルを起動する．この際, Read/Write 属性に応じて各データセグメントのキャッシュ有効化/無効化を適切に行う．

2 次元ステンシル計算に本手法を適用した場合, 袖領域と袖領域交換時に自タスクが他タスクへ差し出すメモリ領域のそれぞれ 1 行分をデータセグメントとして使用する．袖領域交換時に自タスクが差し出すメモリ領域のデータセグメントが自タスクの実行デバイスで有効であることは自明である．しかし, 袖領域のデータセグメントは他タスクにより更新されるため, 自タスクの実行デバイス上に存在するデータセグメントは無効化する．そのため, 自タスクを同一実行デバイスで再度実行するにあたり, 実行に必要なデータセグメントをすべて有効化する必要が生じる．この時, 無効なデータセグメントを有効化する挙動が袖領域交換の動作と一致する．

4.3 コンパイラ

コンパイラとランタイムの組み合わせで実現している動的タスクスケジューリング機構において, source-to-source コンパイラは以下の作業を担当する．

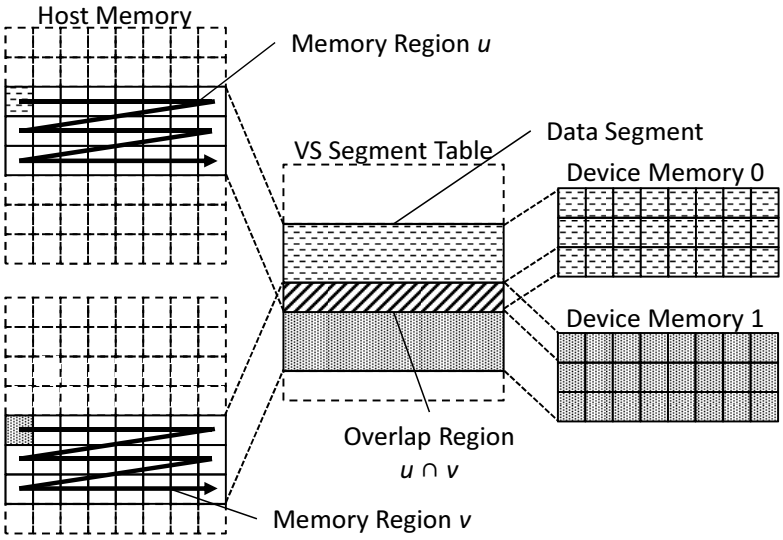

図 4.7: 仮想共有メモリの最適分割に基づくキャッシュ管理の概要

- カーネル及び仮想共有変数の静的解析
- タスク関数生成
- スケジューラ起動やジョブ生成等のランタイムルーチン挿入

本機構の機能拡張において, コンパイラは実行時に決定されるタスクの粒度に応じてホスト変数の先頭要素へのポインタとバイト数のパラメータ組をランタイムに提供する必要がある. これは, 従来の配列インデックス解析による単位スレッドブロック当たりのアクセス範囲解析手法の応用で実現可能である [13]. しかし重複領域を求めるために 1 次元の連続したメモリ領域を必要とする関係上, 解析で得られたアクセス範囲がメモリ不連続である場合はそのアクセス範囲を内包する連続メモリ領域に拡大してパラメータ組を算出する. その結果, 冗長なデータ転送や最悪の場合配列全体をデータセグメントとして管理することになり. カーネルのメモリアクセスによっては転送オーバヘッドが増加する可能性がある.

一方で, 1 次元の連続メモリ領域ではなくグリッド状に分割することで効率のよいメモリ管理ができる場合を想定できる. しかし, 周囲 8 近傍や周囲 27 近傍のみを参照する一般的な 2 次元・3 次元ステンシル計算の場合, グリッド状にメモリを分割すると袖領域が増加するため VS セグメントテーブルの生成・管理が複雑になる. またステンシル計算のメモリアク

セスは典型的なコアレッシングアクセスであるため, GPU の特性上連続メモリ領域である方が高速なメモリアクセスを期待できる [6]. 従って, 本研究はステンシル計算の実行効率が向上する可能性が高い 1 次元連続メモリ領域を扱う方式を選択する.

また従来のスケジューリング機構はタスク関数実行時に readback 転送が行われるが, 提案手法のメモリ管理方式では readback 転送を行わない. そのため, ホストでの仮想共有変数の参照直前にランタイムへ readback 転送を要求するランタイムルーチンを新たに挿入する必要がある. これは, 仮想共有変数に対するホストのアクセス範囲をユーザコードから解析することで実現が可能である.

また本手法の実現にあたり, 各ジョブが参照する仮想共有変数に対して Read/Write 属性を新たに付与する必要がある. これは, カーネル関数における仮想共有変数の配列参照を抽出することで解析可能であり, ジョブ生成のランタイムルーチンに解析情報を挿入することで実現する.

5 コード生成機構の実装

5.1 CUDA コード本体

図 3.4 のコードから本手法により生成される CUDA コードを図 5.8 に示す. まず main 関数開始時にスケジューラを起動し (図 5.8:16 行), ユーザコード上に出現する仮想共有変数及びジョブをランタイムへ登録する (図 5.8:20-38 行). 仮想共有変数の登録には対応するホスト変数とメモリサイズの他, ランタイムでの最適化に必要な解析情報として読み込み専用 (READONLY), 書き込み専用 (WRITEONLY), 読み書き両方発生 (READWRITE) のいずれかを指定する (図 5.8:20-22 行). ジョブは `job_t` データ構造にタスク関数の関数ポインタ, 参照する仮想共有変数, 総スレッドブロック数を格納してランタイムルーチンで登録する (図 5.8:23-38 行). この際, 仮想共有変数に対する Read/Write 属性をカーネル関数内の代入文において, 右辺に出現する場合は READ, 左辺に出現する場合は WRITE, 両辺に出現する場合は READWRITE を指定する (図 5.8:26-28 行, 33-34 行).

次にユーザが記述したカーネル起動をジョブ生成コードに置き換える (図 5.8:43-50 行). ジョブ生成でカーネル起動に必要なスカラ値を `job_t` データ構造に格納し (図 5.8:43-46 行), ランタイムルーチンでランタイム

にジョブ実行許可を通知する (図 5.8:47 行, 49 行). この際, 実行許可を通知したジョブはランタイムによりユーザコードとは別のホストスレッド上で非同期実行されるため, ジョブ間の依存関係が破綻しないようにランタイムと同期をとる必要がある (図 5.8:48 行, 50 行).

最後に, ホストでの仮想共有変数の参照直前にランタイムへ readback 転送を要求するランタイムルーチンを挿入する (図 5.8:52 行). この時, ホストが必要なメモリ領域のみをホスト変数の先頭要素ポインタとオフセットで指定することで, ランタイムに最小限の readback 転送を実行させる.

5.2 カーネル関数及びタスク関数

図 3.4 から生成されるカーネル関数及びタスク関数を図 5.9 に示す. コード変換機構は分割カーネル起動を実現するために, コード変換したカーネル関数 (図 5.9:1-13 行, 14-18 行) とタスク関数であるカーネル起動のラッパー関数 (`execute` 関数) をカーネル関数毎に生成する (図 5.9:19-25 行, 40-44 行). また提案手法のための拡張として, タスクの粒度に応じてホスト変数の先頭要素へのポインタとバイト数のパラメータ組を取得するタスク関数 (`access_range` 関数) を新たに生成する (図 5.9:26-39 行, 45-53 行). ランタイムはジョブが保持する総スレッドブロック数を分割することでタスクを生成する. 従って, タスクに割り当てられるスレッドブロック範囲は開始スレッドブロック数 (`start_block`) とスレッドブロックの数 (`block_num`) で一意に定まる. よってタスク関数は, 開始スレッドブロック数とスレッドブロックの数を引数に挙動を変化させる関数を生成すれば良い.

`access_range` 関数はタスク間におけるメモリアクセス範囲の重複検出のためにランタイムが利用する. このタスク関数を生成するためには参照する仮想共有変数に対する単位スレッドブロック当たりのアクセス範囲を解析で求める必要がある. 単位スレッドブロック当たりのアクセス範囲はユーザが記述するカーネル関数の配列インデックス解析で求めることができる. しかし制約条件として, 配列のインデックス式はビルドイン変数及び定数回ループ変数で表される一次式でなければならない. 実際の動作として, 引数の `vs_arg_id` で対応する仮想共有変数名の処理に分岐し, 解析で得られた単位スレッドブロック当たりのアクセス範囲と `start_block`, `block_num` を用いて `start_ptr` と `size` の値を求める.

execute 関数はまず_scheduler.get_ptr 関数により, ランタイムが VS セグメントテーブルにより管理する実行に必要なデバイスメモリを取得する (図 5.9:20-22 行, 41-42 行). そして, start_block, block_num を用いて分割カーネルを起動する (図 5.9:24 行, 43 行). カーネル関数が第 1 引数に start_block を指定するようコード変換しているのは, 分割スレッドブロックの起動に最小限必要なメモリ領域でカーネルを起動するためである. カーネル関数は必要最小限のメモリ領域でも安全に動作するよう, 物理スレッドマッピングや配列インデックスの条件式を指定スレッドブロックに応じて調整する (図 5.9:2-7 行, 15-16 行).

6 性能評価

提案手法のメモリ管理方式による自動最適化の有効性をベンチマークプログラムを用いて評価する. 実験に使用したベンチマークは Rodinia ベンチマーク [4] から選択した表 6.1 に示す 2 つのステンシル計算を行うアプリケーションである. また, 実験に用いたマルチ GPU 環境を表 6.2 に示す. hotspot 及び srad_v2 はどちらもステップシミュレーションであり, ループ内でステンシル計算のカーネルを起動する. このようなカーネル起動をマルチ GPU 環境で分割実行する場合, データ転送量が最小となるのはループの前後でそれぞれ download/readback 転送を 1 回だけ行い, ループ内は袖領域のみを転送する挙動である. またほぼ同性能のデバイスが 2 つの場合, 袖領域の数が最小で転送オーバヘッドが最小になる単純 2 分割が最も妥当な分割粒度である. 今回の実験で比較対象となる手動最適化した各ベンチマークの CUDA プログラム (opt-CUDA) は前述のマルチ GPU 最適化を施す. それに対し提案手法により生成される各ベンチマークの CUDA プログラム (MESI-CUDA) はジョブ分割数を 2 で設定し, 分割粒度は opt-CUDA と同じ条件にする. 表 6.2 に示す 2 台のマルチ GPU 環境を用いて, 各 GPU 単体及びマルチ GPU で各ベンチマークの opt-CUDA と MESI-CUDA の実行時間を比較した.

評価環境 0 における実行時間を表 6.3, 評価環境 1 における実行時間を表 6.4 に示す. 表 6.3, 表 6.4 に示すように, 手動最適化した CUDA コードと提案手法により生成した CUDA コードでは実行時間にほぼ差がないことがわかる. これは, 手動最適化した CUDA コードにおける袖領域交換を含む一連の最適挙動を提案手法により自動生成した CUDA コードが実現しているためと考えられる.

```
 1 #define N 8192
 2 #define STEP_ITERATION 1000
 3 #define _B 256
 4 double *_h_power, *_h_temp, *_h_result;
 5 vs_arg_t _s_power, _s_temp, _s_result;
 6 job_t *_j_stencil, *_j_copy;
 7 scheduler_t _scheduler;
 8 __global__ void stencil(int _b, double result[][N], double temp[][N], double power[][N],
                          double Cap, double Rx, double Ry, double Rz){...}
 9 __global__ void copy(int _b, double temp[][N], double result[][N]){...}
10 void read_input(double data[][N]){...}
11 void write_output(double data[][N]){...}
12 void set_scala(double *Cap, double *Rx, double *Ry, double *Rz){...}
13 int main(int argc, char **argv){
14   int i;
15   double cap, rx, ry, rz;
16   _scheduler.init();
17   cudaMallocHost(&_h_power, S);
18   cudaMallocHost(&_h_temp, S);
19   cudaMallocHost(&_h_result, S);
20   _scheduler.regist_vs_arg(&_s_power, _h_power, N*N, sizeof(double), READONLY);
21   _scheduler.regist_vs_arg(&_s_temp, _h_temp, N*N, sizeof(double), READWRITE);
22   _scheduler.regist_vs_arg(&_s_result, _h_result, N*N, sizeof(double), READWRITE);
23   _j_stencil = _scheduler.create_job();
24   _j_stencil->exe_func = _exe_stencil_0;
25   _j_stencil->access_range_func = _ar_stencil_0;
26   _j_stencil->vs_args.push_back(std::make_pair(&_s_result, WRITE));
27   _j_stencil->vs_args.push_back(std::make_pair(&_s_temp, READ));
28   _j_stencil->vs_args.push_back(std::make_pair(&_s_power, READ));
29   _j_stencil->block_num = N*N / _B;
30   _j_copy = _scheduler.create_job();
31   _j_copy->exe_func = _exe_copy_0;
32   _j_copy->access_range_func = _ar_copy_0;
33   _j_copy->vs_args.push_back(std::make_pair(&_s_temp, WRITE));
34   _j_copy->vs_args.push_back(std::make_pair(&_s_result, READ));
35   _j_copy->blockNum = N*N / _B;
36   _scheduler.submit_job(_j_stencil);
37   _scheduler.submit_job(_j_copy);
38   _scheduler.fin_submit_job();
39   read_input((double(*)[N])_h_power);
40   read_input((double(*)[N])_h_temp);
41   set_scala(&cap, &rx, &ry, &rz);
42   for (i = 0; i < STEP_ITERATION; i++){
43     _j_stencil->args[0].d = Cap;
44     _j_stencil->args[1].d = Rx;
45     _j_stencil->args[2].d = Ry;
46     _j_stencil->args[3].d = Rz;
47     _scheduler.ignite_job(_j_stencil);
48     _scheduler.synchronize();
49     _scheduler.ignite_job(_j_copy);
50     _scheduler.synchronize();
51   }
52   _scheduler.read_vs_arg(s_result, h_result, N*N*sizeof(double));
53   write_output((double(*)[N])_h_result);
54   cudaFreeHost(_h_power);
55   cudaFreeHost(_h_temp);
56   cudaFreeHost(_h_result);
57   _scheduler.terminate();
58 }
```

図 5.8: 図 3.4 から生成される CUDA コード

```
 1 __global__ void stencil(int _b, double result[][N], double temp[][N], double power[][N],
                           double Cap, double Rx, double Ry, double Rz){
 2   int x = blockIdx.x % (N / _B) * _B + threadIdx.x;
 3   int y = blockIdx.x / (N / _B);
 4   int n = (y + _b / (N / _B) > 0) ? y-1 : y;
 5   int s = (y + _b / (N / _B) < N - 1) ? y+1 : y;
 6   int w = (x > 0) ? x-1 : x;
 7   int e = (x < N - 1) ? x+1 : x;
 8   double tx = (temp[y][w] + temp[y][e] - 2.0 * temp[y][x]) * Rx;
 9   double ty = (temp[n][x] + temp[s][x] - 2.0 * temp[y][x]) * Ry;
10   double tz = (AMB_TEMP - temp[y][x]) * Rz;
11   double delta = Cap * (power[y][x] + tx + ty + tz);
12   result[y][x] = temp[y][x] + delta;
13 }
14 __global__ void copy(int _b, double temp[][N], double result[][N]){
15   int x = blockIdx.x % (N / _B) * _B + threadIdx.x;
16   int y = blockIdx.x / (N / _B);
17   temp[y][x] = result[y][x];
18 }
19 void _exe_stencil_0(task_t *task, int dev){
20   double *d_result = (double *)_scheduler.get_ptr(task, _s_result, dev);
21   double *d_temp = (double *)_scheduler.get_ptr(task, _s_temp, dev);
22   double *d_power = (double *)_scheduler.get_ptr(task, _s_power, dev);
23   d_temp = (task->start_block == 0) ? d_temp : d_temp + N;
24   stencil<<<task->block_num, _B, 0, 0>>>
       (task->start_block, (double(*)[N])d_result, (double(*)[N])d_temp, (double(*)[N])d_power,
       task->parent_job->args[0].d, task->parent_job->args[1].d, task->parent_job->args[2].d, task->parent_job->args[3].d);
25 }
26 void _ar_stencil_0(int vs_arg_id, int start_block, int block_num, void **start_ptr, size_t *size){
27   if (vs_arg_id == _s_power.id){
28     *size = block_num * _B * sizeof(double);
29     *start_ptr = &_h_power[start_block * _B];
30   }else if (vs_arg_id == _s_result.id){
31     *size = block_num * _B * sizeof(double);
32     *start_ptr = &_h_result[start_block * _B];
33   }else if (vs_arg_id == _s_temp.id){
34     *size = block_num * _B * sizeof(double);
35     *size = start_block == 0 ? *size : *size + sizeof(double) * N;
36     *size = (start_block + block_num) == N*N / _B ? *size : *size + sizeof(double) * N;
37     *start_ptr = start_block == 0 ? &_h_temp[start_block * _B] : &_h_temp[start_block * _B - N];
38   }
39 }
40 void _exe_copy_0(task_t *task, int dev){
41   double *d_temp = (double *)_scheduler.get_ptr(task, _s_temp, dev);
42   double *d_result = (double *)_scheduler.get_ptr(task, _s_result, dev);
43   copy<<<task->block_num, _B, 0, 0>>>(task->start_block, (double(*)[N])d_temp, (double(*)[N])d_result);
44 }
45 void _ar_copy_0(int vs_arg_id, int start_block, int block_num, void **start_ptr, size_t *size){
46   if (vs_arg_id == _s_temp.id){
47     *size = block_num * _B * sizeof(double);
48     *start_ptr = &_h_temp[start_block * _B];
49   }else if (vs_arg_id == _s_result.id){
50     *size = block_num * _B * sizeof(double);
51     *start_ptr = &_h_result[start_block * _B];
52   }
53 }
```

図 5.9: 図 3.4 から生成されるカーネル関数及びタスク関数

この結果を受けて, MESI-CUDA 処理系は自動最適化のために解析情報を用いるタスク分割の必要性が生じた. 従来の MESI-CUDA は個々の GPU 性能が不均一なマルチ GPU 環境を想定し, ランタイムが動的負荷分散を効率よく行うためにタスク数をある程度確保するタスク分割を行う. しかし, ステンシル計算のようにタスク間でデータ転送を必要とする場合, 不均一なマルチ GPU 環境においてもタスク数を抑えた方が最適となる. 従って, 本研究に加えカーネルのメモリアクセスパターンや計算負荷を解析情報としてマルチ GPU 環境に応じて自動最適化する機構を今後提案する必要がある.

表 6.1: 評価ベンチマーク

アプリケーション	概要
hotspot	2 次元過渡熱シミュレーション (1000steps)
srad_v2	2 次元エコー画像のノイズ除去 (1000steps)

表 6.2: 評価環境

	評価環境 0	評価環境 1
GPU 0	Geforce GTX 960(2GB MEM)	Tesla K80(24GB MEM)[1]
GPU 1	Geforce GTX 950(2GB MEM)	
CPU	Xeon E5-2620 2.1GHz	Xeon E5-2630 2.4GHz × 2
host memory	16 GB	32 GB

7 関連研究

マルチ GPU 環境について, 個々のデバイスやデバイスメモリを隠蔽することで複数デバイス制御や分散メモリデータの管理等を自動的に行う研究が盛んに行われている. OpenACC[5] は逐次コード内の並列化したい処理に対し簡単な指示文を挿入するだけで GPGPU プログラミングを可能

[1]Tesla K80 は GK210 を 2 基搭載する GPU ボードである. そのため, このデバイス単体でマルチ GPU 環境の構築が可能である.

表 6.3: 評価環境 0 における実行時間 (秒)

	GTX960		GTX950		GTX960+GTX950	
size	opt-CUDA	MESI-CUDA	opt-CUDA	MESI-CUDA	opt-CUDA	MESI-CUDA
hotspot						
4096^2	7.876	8.903	8.694	9.754	4.625	5.004
8192^2	31.458	32.917	34.408	35.810	17.696	18.582
srad_v2						
2048^2	9.091	10.633	11.332	12.928	6.319	6.840
4096^2	35.675	37.023	44.448	45.861	23.898	24.220

表 6.4: 評価環境 1 における実行時間 (秒)

	K80(1GPU 利用)		K80(2GPU 利用)	
size	opt-CUDA	MESI-CUDA	opt-CUDA	MESI-CUDA
hotspot				
4096^2	4.277	5.019	2.478	2.915
8192^2	16.667	17.130	8.844	9.331
srad_v2				
2048^2	3.251	4.382	2.182	3.165
4096^2	11.914	13.088	6.985	7.485

にしており, そこから更に複数の GPU を自動利用する研究が行われている [8]. OpenACC によりユーザは CUDA や GPU アーキテクチャの知識を習得することなくマルチ GPU を扱える一方で, 高性能を実現するためには指示文を駆使した細かなチューニングを施す必要がある. MESI-CUDA は記述の容易さでは OpenACC に劣るが, 並列化を行うカーネルの記述は CUDA と同様にユーザが行うため, 高速なコードを生成しやすい利点がある.

AMGE[7] は MESI-CUDA と同様にコンパイラとランタイムを組み合わせたフレームワークであり, 実行時に配列やカーネルを自動分割し複数の GPU でカーネル起動を行う. AMGE のメモリ分割は BLOCK, BLOCK-CYCLIC といった複数の分割パターンの中から実行時に最適な分割方式を選択する方式である. それに対し, MESI-CUDA は解析で得られるカーネルのメモリアクセス範囲に応じて最適なメモリ分割を実行時に行うため任意のサイズに分割可能であり, 袖領域などに対応できる.

またステンシル計算について, マルチ GPU 環境だけでなく様々な分散

並列環境で効率良く袖領域を扱う研究が行われている [9]. 今後これらの研究成果を MESI-CUDA に適用することで更なる性能向上を実現できる可能性がある.

8 終わりに

本研究では MESI-CUDA 処理系の動的タスクスケジューリング機構の機能拡張により, 各カーネルのメモリアクセスに応じた仮想共有メモリの最適分割に基づくメモリ管理手法を提案した. コンパイラはメモリ管理手法を実現するために, ランタイムが要求する解析情報を新たなコード生成機構によって提供した. 性能評価の結果, 本手法を用いることで従来手法では性能低下が発生したステンシル計算をマルチ GPU 環境で効率よく実行するコードを自動生成する処理系を実現した. 今後の課題として, より多くのアプリケーションで処理系の実行効率を評価するとともに, さらなる実行効率の向上を実現する手法を考案する必要がある.

謝辞

本研究を行うに辺り, 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致します.

参考文献

[1] *GPGPU.org: General-Purpose computation on Graphics Processing Units*, http://www.gpgpu.org/, (2017.2.6).

[2] *NVIDIA Developer CUDA Zone*, http://developer.nvidia.com/category/zone/cuda-zone, (2017.2.6).

[3] *OpenCL - The open standard for parallel programming of heterogeneous systems*, http://www.khronos.org/opencl/, (2017.2.6).

[4] *Rodinia Benchmark Suite*, http://lava.cs.virginia.edu/Rodinia/download_links.htm 2017.2.6.

[5] *OpenACC*, http://www.openacc-standard.org/, (2017.2.6).

[6] *CUDA C Best Practices Guide*,NVIDIA Corporation,September,(2016).

[7] Javier Cabezas, Llus Vilanova, Isaac Gelado, Thomas B Jablin, Nacho Navarro, Wen-mei W Hwu, *Automatic parallelization of kernels in shared-memory multi-gpu nodes*,Proceedings of the 29th ACM on International Conference on Supercomputing,3-13,(2015).

[8] Toshiya Komoda, Shinobu Miwa, Hiroshi Nakamura, Naoya Maruyama, *Integrating multi-GPU execution in an OpenACC compiler*,Parallel Processing (ICPP), 2013 42nd International Conference on,260-269,(2013).

[9] Markus Wittmann, Georg Hager, Gerhard Wellein, *Multicore-aware parallel temporal blocking of stencil codes for shared and distributed memory*,Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on,1-7,(2010).

[10] Kazuhiko Ohno, Tomoharu Kamiya, Takanori Maruyama, Masaki Matsumoto, *Automatic Optimization of Thread Mapping for a GPGPU Programming Framework*,2014 Second International Symposium on Computing and Networking (CANDAR'14),198-204,(2014).

[11] 山本 怜, 大野 和彦, *GPGPU* フレームワーク *MESI-CUDA* のマルチ *GPU* 環境への対応, 情報処理学会論文誌プログラミング (PRO),Vol.9,No.1,12-12,(2016).

[12] 田中 宏明, 山本 怜, 大野 和彦, *GPGPU*フレームワーク *MESI-CUDA* におけるデータ再利用性を高めるスケジューラ, 情処研報 2016-HPC-155,1-7,(2016).

[13] Kazuhiko Ohno, Rei Yamamoto, Hiroaki Tanaka, *Dynamic Task Scheduling Scheme for a GPGPU Programming Framework*,International Journal of Networking and Computing,Vol.6,No.2,290-308,(2016).