GPGPU におけるデータ転送を自動化する MESI-CUDA の提案

道 浦 悌[†,☆] 大 野 和 彦[†] 松 本 真 樹[†]
佐々木 敬 泰[†] 近 藤 利 夫[†]

近年，GPU 上で汎用計算を実行する GPGPU が注目されている．また，CUDA や OpenCL などの開発環境がリリースされ，GPU プログラミングは容易になりつつある．しかし，これらの環境では，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マが明示的に記述する必要がある．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 を提案している．本論文では，MESI-CUDA のプログラミングモデルを示し，データ転送とカーネル処理のオーバラップ実現のためのデータフロー解析とストリーム割り当て手法を述べる．MESI-CUDA の性能を示すために，手動で最適化した CUDA プログラムと MESI-CUDA の出力プログラムで実行時間を比較して，評価を行った．その結果，実行時間にほとんど差が無く，ほぼ最適に近いコードを得ることができた．

MESI-CUDA: Automatic generation of data transfer code for GPGPU

Dai Michiura,[†,☆] Kazuhiko Ohno,[†] Masaki Matsumoto,[†]
Takahiro Sasaki[†] and Toshio Kondo[†]

The performance of Graphics Processing Units (GPU) is improving rapidly. Thus, General Purpose computation on Graphics Processing Units (GPGPU) is expected as an important method for high-performance computing. Although programming frameworks, such as CUDA and OpenCL, are provided, they require explicit specification of memory allocations and data transfers. Therefore, we are developing a new programming framework *MESI-CUDA*, which hides such low-level description from the user. In this paper, we present the programming model of MESI-CUDA and show the detail of data flow analysis and stream allocation to overlap data transfers and kernel executions. The evaluation result shows that the performance of MESI-CUDA programs is close to hand-optimized CUDA programs, nevertheless the data transfer code is automatically generated and optimized.

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[1]．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)[2] への関心が高まっている．また，CUDA[3] や OpenCL[4] といった GPGPU プログラム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，プログラマは細かな最適化が可能であるが，プログラミングの難易度は高い．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ならない．これらの最適化は高度なプログラミングを必要とする．一方，デバイスによってメモリ容量が異なるため，コードの移植性が低いものとなっている．

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)~7)] を開発している．本フレームワークは，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．ユーザに対しては，共有メモリ型の GPGPU プログラミングモデルを提供する．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になる．さらに，データ転送と GPU 上での計算のオーバ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．本稿では，

† 三重大学大学院工学研究科
Graduate School of Engineering, Mie University
☆ 現在，ジェイアール東海情報システム株式会社
Presently with JR TOKAI Information Systems Company

© 2012 Information Processing Society of Japan

MESI-CUDA フレームワークのプログラミングモデルや記述方法，フレームワーク内部のデータフロー解析方法やストリーム割り当て手法を示す．

以下，２章では背景として GPU アーキテクチャと CUDA の概要を述べる．３章では関連研究を紹介し，4 章で MESI-CUDA の機能とプログラミングモデルについて説明する．５章ではデータフロー解析やコード生成などの MESI-CUDA の内部処理手法を述べる．6 章で，MESI-CUDA の出力する CUDA プログラムと手動で最適化した CUDA プログラムとの性能比較の評価結果を示す．最後に 7 章でまとめを行う．

2. 背　　景

2.1 GPU アーキテクチャ

GPU の基本的なアーキテクチャは，多数のコアがグローバルメモリを共有している構造である．しかし，メモリは複雑に階層化されており，それぞれの用途ごとに使い分ける必要がある．また，コアは一定数でまとめられており，そのユニットごとにレジスタやシェアードメモリ，ローカルメモリを有する．読み込み専用だが高速なメモリであるコンスタントメモリやテクスチャメモリもあり処理に合わせて用いる．

GPU の性能を最大限に引き出すためにはこれらのメモリ階層を使い分ける必要があり，プログラマの負担が大きい．さらに，GPU は現在も性能が向上し続けており，デバイスによってコア数や各メモリサイズなどのスペックが異なる．このため，特定のデバイス上でプログラムを最適化しても，他のデバイスでは高性能を発揮できないことも多い．つまり，コードの移植性にも大きな問題を抱えている．

2.2 CUDA

CUDA は nVIDIA 社より提供されている GPGPU 用の SDK であり，C 言語を拡張した文法とライブラリ関数を用いて GPU プログラムを開発することができる．CUDA では，CPU をホスト，GPU をデバイスと呼ぶ．CUDA のサンプルプログラムを**図 1** 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，修飾子`__device__` か`__global__` が付与される (図 1：4, 8 行)．修飾子`__host__`あるいは修飾子のない関数は，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カーネルを実行することができる (図 1：40, 44, 46 行)．このときに作成するスレッド数を指定する．

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

```
 1 #include <stdio.h>
 2 #define N 12800
 3 #define SIZE (N*sizeof(int))
 4 __global__ void add_array(int *kernel_arg, int *a){
 5   int id = blockDim.x*blockIdx.x+threadIdx.x;
 6   a[id] = a[id] + kernel_arg[id];
 7 }
 8 __global__ void prod_array(int *d, int *e, int *f){
 9   int id = blockDim.x*blockIdx.x+threadIdx.x;
10   d[id] = e[id] * f[id];
11 }
12 int main(){
13   int *ha, *hb, *hc, *hd, *he, *hf;//ホスト用変数
14   int *da, *db, *dc, *dd, *de, *df;//デバイス用変数
15   cudaMallocHost((void**)&ha, SIZE);
          :
20   cudaMallocHost((void**)&hf, SIZE);
21   cudaMalloc((void**)&da, SIZE);
          :
26   cudaMalloc((void**)&df, SIZE);
27   cudaStream_t st[2];
28   cudaStreamCreate(&st[0]);
29   cudaStreamCreate(&st[1]);
30   load_array(ha);
31   load_array(hb);
32   load_array(hc);
33   load_array(he);
34   load_array(hf);
35   cudaMemcpyAsync(da, ha, SIZE,
                     cudaMemcpyHostToDevice, st[0]);
36   cudaMemcpyAsync(db, hb, SIZE,
                     cudaMemcpyHostToDevice, st[0]);
37   cudaMemcpyAsync(dc, hc, SIZE,
                     cudaMemcpyHostToDevice, st[0]);
38   cudaMemcpyAsync(de, he, SIZE,
                     cudaMemcpyHostToDevice, st[1]);
39   cudaMemcpyAsync(df, hf, SIZE,
                     cudaMemcpyHostToDevice, st[1]);
40   add_array<<<N/32, 32, 0, st[0]>>>(db, da);
41   cudaMemcpyAsync(ha, da, SIZE,
                     cudaMemcpyDeviceToHost, st[0]);
42   cudaStreamSynchronize(st[0]);//ha の転送完了待ち
43   output_array(ha);
44   add_array<<<N/32, 32, 0, st[0]>>>(dc, da);
45   cudaMemcpyAsync(ha, da, SIZE,
                     cudaMemcpyDeviceToHost, st[0]);
46   prod_array<<<N/32, 32, 0, st[1]>>>(dd, de, df);
47   cudaMemcpyAsync(hd, dd, SIZE,
                     cudaMemcpyDeviceToHost,st[1]);
48   cudaStreamSynchronize(st[0]);//ha の転送完了待ち
49   output_array(ha);
50   cudaStreamSynchronize(st[1]);//hd の転送完了待ち
51   output_array(hd);
52   cudaFreeHost(ha);
          :
57   cudaFreeHost(hf);
58   cudaFree(da);
          :
63   cudaFree(df);
64   cudaStreamDestroy(st[0]);
65   cudaStreamDestroy(st[1]);
66 }
```

図 1 CUDA コードの例
Fig. 1 Sample Program using CUDA

© 2012 Information Processing Society of Japan

う．データ転送には，ホストからデバイスへのデータ転送をする download 転送 (図 1：35-39 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 1：41, 45, 47 行) の 2 種類がある．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．

データ転送には `cudaMemcpy` 関数で行う同期式と，`cudaMemcpyAsync` 関数で行う非同期式の 2 方式がある．前者を用いた場合は，転送関数呼び出し後，転送完了までブロックする．後者を用いた場合は，データ転送を発行するだけでブロックせずに呼び出し元に復帰する．このためデータ転送とカーネルの同時実行が可能になり実行性能の向上が見込めるが，転送完了のタイミングは保証されない．また，以下で説明するストリームを用いなければならない．

ストリーム

ストリームは依存関係のある非同期のデータ転送やカーネル実行を結びつけるためのもので，各ストリーム上ではデータ転送やカーネル実行が登録順に実行されていく．ストリームは明示的な生成 (図 1：27-29 行) と破棄 (図 1：64-65 行) が必要であり，データ転送にストリームを割り当てるためには，第 5 引数に所属させるストリームを与える (図 1：35-39, 41, 45, 47 行)．カーネルにストリームを割り当てるためには，カーネル呼び出し時にスレッド数と同時に指定する (図 1：40, 44, 46 行)．実行中のカーネルの所属ストリームとデータ転送の所属ストリームが異なっている場合，カーネル実行とデータ転送が同時に実行され，これらのオーバラップが実現できる．従って，より高効率なデータ転送を実現するためにはストリームの管理を行う必要があり，プログラマの負担が増える．

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1:21-26, 58-63 行)．さらに，ストリームを用いてデータ転送とカーネル実行をオーバラップする場合，ホスト側で使用する変数に対しても `cudaMallocHost`, `cudaFreeHost` 関数を用いてメモリ確保・解放を行う必要がある (図 1：15-20, 52-57 行)．

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[8),9)] が多くなされており，定型的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．しかし，定型的でない逐次処理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，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[10)] も行われている．しかし，この手法は GPU プログラムを解析してデータを割り当てるため，従来通りの GPU プログラミングを行う必要がある．

MESI-CUDA フレームワークは並列処理こそ記述する必要があるが，共有メモリ型プログラミングモデルを採用した上で，明示的なロックや排他制御，同期を不要にしている．このため，特定の分野に限らず GPU プログラミング全般において，プログラマの負担を減らしつつ高性能を達成できると考えている．

4. MESI-CUDA の機能

4.1 MESI-CUDA 概要

MESI-CUDA フレームワークは, データ転送やメモリ確保・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関数の記述は, 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．この方針は以下の理由による．

- CUDA では，ホストとデバイスへの処理の割り当てを，ホスト側コードとデバイス用カーネル関数の記述で行う．これは，比較的単純でわかりやすいモデルであり, デバイスへの依存性も低い．
- カーネル関数を実行する順序・タイミングのスケジューリング，デバイスメモリの容量による転送可能なデータ量の管理などは，デバイスに依存しプログラミングが困難である．
- データ転送は性能に大きく影響する一方，デバイスへの依存性が高い．そのため，MESI-CUDA に任せることでユーザの負担を大きく減らせる．また，フレームワーク内で処理することで処理系の自由度が上がり，最適化の余地も増える．

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．そのために，仮想的な共有メモリモデルを採用し，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．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，転送のタイミングやカーネル実行の順序を自動的に最適化し，

ⓒ 2012 Information Processing Society of Japan

```
1 #include <stdio.h>
2 #define N 12800
3 __share__ int a[N], b[N], c[N], d[N], e[N], f[N];
4 __global__ void add_array(int *kernel_arg){
5   int id = blockDim.x*blockIdx.x+threadIdx.x;
6   a[id] = a[id] + kernel_arg[id];
7 }
8 __global__ void prod_array(){
9   int id = blockDim.x*blockIdx.x+threadIdx.x;
10  d[id] = e[id] * f[id];
11 }
12 int main(){
13  load_array(a);
14  load_array(b);
15  load_array(c);
16  load_array(e);
17  load_array(f);
18  add_array<<<N/32, 32>>>(b);
19  output_array(a);
20  add_array<<<N/32, 32>>>(c);
21  prod_array<<<N/32, 32>>>();
22  output_array(a);
23  output_array(d);
24 }
```

図 2 CUDA コードと等価な MESI-CUDA コード
Fig. 2 Sample Program using MESI-CUDA

カーネル実行とデータ転送のオーバラップが可能になるようにストリームの割り当てを行う．

図 1 の CUDA プログラムと等価な MESI-CUDA プログラムを**図 2** に示す．カーネル関数やホスト側の計算処理は CUDA と同様であるが，共有変数を用いることによって，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．

本フレームワークでは，MESI-CUDA コンパイラ `mecc` により，ユーザが記述したプログラムをコンパイルする．`mecc` は MESI-CUDA から CUDA へのトランスレータとして実現されており，内部で CUDA コンパイラ `nvcc` を呼び出してトランスレート結果をコンパイルすることで，実行形式を得る．

4.2 プログラミングモデル

本フレームワークのプログラミングモデルは以下の通りである．

- ホストプログラムは従来通り逐次処理を行う．
- カーネル関数の記述・呼び出しは CUDA の記述に準拠し，スレッド数の指定はユーザが行う．
- 共有メモリモデルを採用し，共有変数はホスト・デバイスどちらからもアクセスが可能である．

共有変数は，修飾子として`__share__`を付与して宣言する (図 2：3 行)．カーネル呼び出し前にホストで共有変数に対して代入が発生していた場合，カーネルでは代入後の値が参照される．逆に，カーネル呼び出し後にホストで共有変数を参照した場合は，カーネルの処理がすべて完了した後の値が得られる．また，カーネル関数の呼び出しは，以降いつ実行しても良いという実行許可であり，実際の実行タイミングはフレームワークにより決定される．これにより，データ転送とカーネル関数実行を MESI-CUDA 側でスケジューリングすることができ，最適化が可能になる．

CUDA と同様，ホスト側でのカーネル関数呼び出しはカーネル実行の完了を待たずに呼び出し元に復帰するため，基本的にホスト側とデバイス側は並列実行が可能である．ただし，実行中のカーネル内で書き込まれる変数をホスト上で参照した場合は，上記モデルを保証するため，カーネルの実行終了までホスト側の処理がブロックする．

4.2.1 本プログラミングモデルの特徴

共有変数ベースのプログラミングにより，データ転送やストリーム処理などの煩雑な記述が不要になり，暗黙の同期モデルを採用することで明示的な同期記述も必要ない．このため簡潔なコーディングが可能であり，カーネル記述を特殊な関数と見なせば C 言語ライクなコーディングができる．さらに，デバイス固有のスペックに依存するコードをフレームワーク内で自動生成・最適化するため，ユーザの負担軽減と共にプログラムの移植性を高めることができる．

一方で，複雑なメモリ構造をフレームワークで隠蔽しているため，各種のメモリを用途に合わせて使い分けたり，クリティカルなデータ転送のタイミングや粒度を調整したりといった，ユーザによる手動チューニングを行うことはできない．そのため，実行性能は処理系の最適化能力に大きく依存する．

5. MESI-CUDA の設計

カーネル関数はユーザが明示的に記述するため，フレームワークは共有変数に対して，実際にアクセスを行えるようにコードを生成すればよい．そのためには，各カーネルで使用されている共有変数を解析し，カーネル実行の前後にデータ転送コードを挿入すればよい．さらに，データフロー解析を行い，互いに依存関係のないカーネル実行とデータ転送を別々のストリームに割り当てることで，両者のオーバラップを実現する．

フレームワークの処理全体の流れを以下に示す．

(1) データフロー解析
(2) ストリーム割り当て
(3) 同期ポイント解析
(4) スケジューリング
(5) コード生成

© 2012 Information Processing Society of Japan

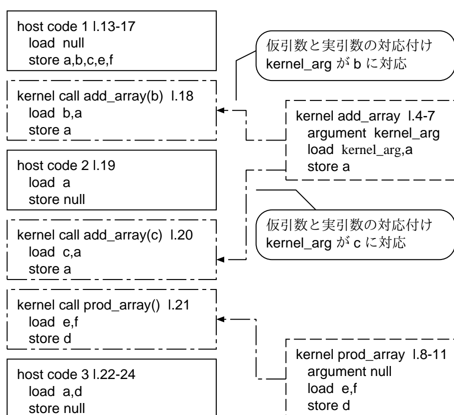

図 3 共有変数の参照・代入の解析結果
Fig. 3 Result of Access Analysis for Shared Variables

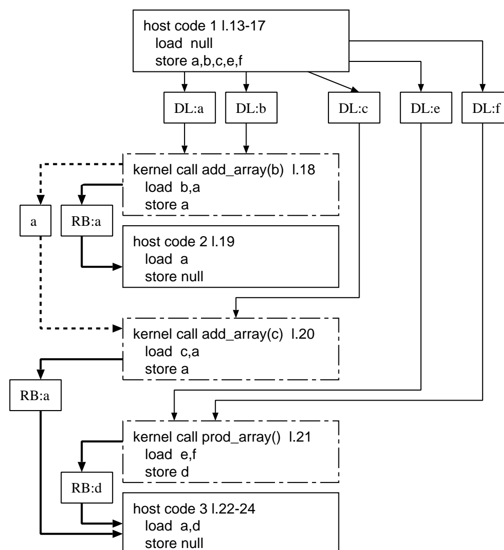

図 4 データフロー解析の結果
Fig. 4 Result of Dataflow Analysis

(a) メモリの確保・解放
(b) ストリームの生成・破棄
(c) データ転送

5.1 データフロー解析

データ転送タイミングやストリーム割り当てを決定するため，データフロー解析を行う．共有変数として宣言された変数およびそれらと依存関係のある変数のみ対象とし，ホスト・デバイスでのそれらの変数の参照・代入の有無と変数間の依存関係を解析する．全変数の解析や変数の値を追跡する必要はない．

最初に修飾子`__share__`を付与して宣言された変数を変数表に登録し，これらが依存する非共有変数を追加していく．データフロー解析は，まず修飾子`__global__`が付与された各関数を基点としてカーネル単位の解析を行う．続いて`main`関数を基点としてホスト上の解析を行い，共有変数への参照・代入がカーネル呼び出しに対してどの位置で行われているかを調べ，カーネル呼び出し時の実引数と仮引数を対応付けする． 図 2 における参照・代入の解析結果は**図 3** のようになる．

必要なデータ転送は，次のようにして求められる．

download 転送

各カーネルで参照されている共有変数は，カーネル呼び出しの前にデータの download 転送が必要である．そのため，各カーネル呼び出し以前のホストコードから，共有変数に代入している行を探す．データ転送は早期に発行した方がカーネル呼び出し時にデータ転送が完了している可能性が高くなるため，カーネル呼び出し直近の代入直後にデータ転送コードを挿入する．

readback 転送

カーネル内で更新される共有変数がホスト側で参照される場合，データの readback 転送が必要である．そこで，カーネル呼び出し後のホストコードから，これらの共有変数を参照している行を探す．ホストでの共有変数参照時に転送が完了している可能性を高めるため，カーネル呼び出しの直後に転送コードを挿入する．

図 2 に対するデータフロー解析結果は，**図 4** のようになる．

5.2 ストリーム割り当て・同期ポイント解析

データ転送とカーネル実行のオーバラップを実現するためには，両者の所属するストリームが異なっていなければならない．そのため，すべてのカーネル実行と関連するデータ転送をそれぞれ異なるストリームに割り当てれば，最大限のオーバラップが期待できる．

しかし，同一ストリーム内での処理は登録順に実行されるが，異なるストリーム間の実行順序は実行時に決定される．そのため，ある共有変数のデータ転送と，それにアクセスしているカーネル実行は，すべて同一のストリームに所属させるか，必要に応じてストリームの同期処理を挿入してデータ転送／カーネル実行の完了を保証する必要がある．そこで，データフローの解析結果を基に，以下の手順でストリーム割り当てを行う．以下，共有変数の依存関係があるカーネル実行とデータ転送の集まりを，**カーネルグループ**と呼ぶ．

(1) 各カーネル実行とデータ転送をカーネルグループにクラスタリングする．
(2) 各カーネルグループにストリームを割り当てる．このとき，単一のカーネルしかアクセスしない変数がある場合は，その転送に新しいストリームを割り当てる．

ⓒ 2012 Information Processing Society of Japan

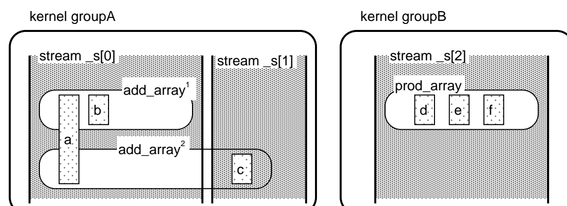

図 5　変数・カーネル・ストリームの関係
Fig. 5　Stream Assignment for Program in Fig. 2

(3)　必要なストリーム数を記録する．

(2) について，カーネルグループごとのストリーム割り当てでは，グループ内のカーネルに関してはオーバラップを行えず，最適にならない．しかし，各カーネルが単独で使用している変数については，ストリームを分けることでオーバラップが可能である．そこで，その変数のデータ転送だけで使用するストリームを割り当てる．ただし，このようなストリームの分割を行った場合は，download 転送の場合もストリームの同期を取る必要がある．また，ホスト側での参照はストリームに所属させることはできないので，readback 転送の場合は常に同期処理を行い，参照までにデータ転送を完了させる必要がある．

図 2 の場合，変数とカーネルの関係およびストリームの割り当ては**図 5** のようになる．カーネルグループ A は共有変数 a，b，c にアクセスし，カーネルグループ B は共有変数 d，e，f にアクセスする．

さらにこれらのカーネルグループ内の詳細なストリーム割り当てを行う．カーネルグループ A はカーネル関数呼び出しが 2 つある．add_array(b) は a，b へのアクセスであり，add_array(c) は a，c へのアクセスである．カーネル add_array(b) の実行と，次に実行されるカーネル add_array(c) で参照する変数 c のデータ転送は本来オーバラップ可能であるが，カーネルグループごとにストリームを割り当てただけでは実現できない．そこで，変数 c は別のストリーム_s[1] に所属させる．また，カーネル add_array(c) を実行する前に_s[1] の同期処理を挿入することで，所属ストリームの異なる c のデータ転送がカーネル実行時までに完了していることを保証する．カーネルグループ B については単一のカーネル呼び出しのみであるので，すべて同一のストリーム上で処理する．

5.3　スケジューリング

異なるカーネルグループであれば，カーネルの実行順序はどのような順番で実行しても問題ない．そのため，ホスト上の依存関係の範囲内でスケジューリングを行う余地がある．

ユーザに対して同期も隠蔽するフレームワークの性質上，ユーザは同期の位置を意識したプログラミングを行わない．それゆえ，カーネル呼び出しの前にストリームの同期が挿入され，実行可能なカーネル呼び出しが発行されずに同期待ちとなることがある．そのような場合は，カーネル呼び出しのコード位置を変える必要がある．カーネルの呼び出しが可能となるのは，そのカーネルで参照している共有変数すべてに対し，ホスト側での代入が完了した後である．この制約の範囲内で各カーネル呼び出しをできる限り前方に移すことにより，実行効率の良いコードを得ることができる．

図 2 の例では，カーネル prod_array() は，16-17 行の代入後はいつでも実行可能である．しかし，実際には 21 行で呼び出されており．さらに 19 行のホストの共有変数参照で転送待ちの同期が発生する．そのため，同期完了までカーネル呼び出しが実行されず，効率的ではない．カーネル prod_array() の呼び出しを 18 行直後に移動することで，この問題を解決できる．

また，データ転送のタイミングにもスケジューリングの余地がある．変数を使用しているカーネルの実行が後であるほど、転送は後回しにしても良い．したがって，複数の変数の転送が同時期に可能となる場合，それらを参照する各カーネル呼び出しの位置によってデータ転送コードの順序を決定することで，GPU の使用効率が良くなる．

5.4　コード生成

5.4.1　メモリの確保・解放

__share__で宣言された共有変数について，本来必要なホスト用変数，デバイス用変数に拡張する．変数表を参照して，共有変数の変数名 *varname* について，ホスト用変数名は_host_*varname*，デバイス用変数名は_dev_*varname* とする．また，各ホスト／デバイス用変数についてメモリ確保と解放を行うコードを挿入する．さらに，ホストやカーネル関数のコードは共有変数で記述されているため，これらの変数をそれぞれホスト用変数，デバイス用変数に置換する．

図 2 のコードに対し，メモリ確保・解放コードを挿入し，ホストやカーネル関数内の変数アクセスをそれぞれ対応するホスト／デバイス用変数に置換したコードを**図 6** に示す．行番号として括弧が付与された行が，挿入されているコードであり，括弧内の数字の順に，括弧なしの数字の行番号の直後に挿入される．

5.4.2　ストリームの生成・破棄

プログラムの先頭で，ストリーム割り当てで決定した個数のストリームを宣言し，cudaStreamCreate 関数によりストリームを生成する．また，各カーネル関

ⓒ 2012 Information Processing Society of Japan

```
          ⋮
 4 __global__ void add_array(int *kernel_arg, int *_dev_a){
 6   _dev_a[id] = _dev_a[id] + kernel_arg[id];
 7 }
 8 __global__ void prod_array(int *_dev_d, int *_dev_e,
                             int *_dev_f){
10   _dev_d[id] = _dev_e[id] * _dev_f[id];
11 }
12 int main(){
( 1) int *_host_a, *_dev_a;
          ⋮
( 6) int *_host_f, *_dev_f;
( 7) cudaMallocHost((void**)&_host_a,N*sizeof(int));
          ⋮
(12) cudaMallocHost((void**)&_host_f,N*sizeof(int));
(13) cudaMalloc((void**)&_dev_a,N*sizeof(int));
          ⋮
(18) cudaMalloc((void**)&_dev_f,N*sizeof(int));
13   load_array(_host_a);
          ⋮
17   load_array(_host_f);
18   add_array<<<N/32, 32>>>(_dev_b, _dev_a);
19   output_array(_host_a);
20   add_array<<<N/32, 32>>>(_dev_c, _dev_a);
21   prod_array<<<N/32, 32>>>(_dev_d, _dev_e, _dev_f);
22   output_array(_host_a);
23   output_array(_host_d);
( 1) cudaFreeHost(_host_a);
          ⋮
( 6) cudaFreeHost(_host_f);
( 7) cudaFree(_dev_a);
          ⋮
(12) cudaFree(_dev_f);
24   }
```

図 6 メモリ確保と解放・変数の置き換え
Fig. 6 Memory Allocation/Deallocation Code

数とデータ転送関数の呼び出しにおいて，所属するストリーム上で実行するように引数を与える．最後に，プログラム終了直前に cudaStreamDestroy 関数を用いてストリームを破棄する．

5.4.3 データ転送

データフロー解析で決定した位置にデータ転送コードや同期命令を挿入する．挿入されるコードは以下の通りである．

- **download 転送コード：**
  cudaMemcpyAsync(デバイス側変数アドレス, ホスト側変数アドレス, 変数サイズ, cudaMemcpy-HostToDevice, 所属ストリーム);
- **readback 転送コード：**
  cudaMemcpyAsync(ホスト側変数アドレス, デバイス側変数アドレス, 変数サイズ, cudaMemcpy-DeviceToHost, 所属ストリーム);
- **転送同期コード：**
  cudaStreamSynchronize(所属ストリーム);

図 2 に対して，ストリーム処理とデータ転送を挿入したコードを**図 7** に示す．

```
          ⋮
12 int main(){
          ⋮
(19) cudaStream_t _s[3];
(20) int _i;
(21) for (_i = 0 ; _i < 3 ; _i++)
(22)   cudaStreamCreate(&_s[_i]);
13   load_array(_host_a);
( 1) cudaMemcpyAsync(_dev_a, _host_a, N*sizeof(int),
                     cudaMemcpyHostToDevice, _s[0]);
14   load_array(_host_b);
( 1) cudaMemcpyAsync(_dev_b, _host_b, N*sizeof(int),
                     cudaMemcpyHostToDevice, _s[0]);
15   load_array(_host_c);
( 1) cudaMemcpyAsync(_dev_c, _host_c, N*sizeof(int),
                     cudaMemcpyHostToDevice, _s[1]);
16   load_array(_host_e);
( 1) cudaMemcpyAsync(_dev_e, _host_e, N*sizeof(int),
                     cudaMemcpyHostToDevice, _s[2]);
17   load_array(_host_f);
( 1) cudaMemcpyAsync(_dev_f, _host_f, N*sizeof(int),
                     cudaMemcpyHostToDevice, _s[2]);
18   add_array<<<N/32, 32, 0, _s[0]>>>(_dev_b, _dev_a);
( 1) cudaMemcpyAsync(_host_a, _dev_a, N*sizeof(int),
                     cudaMemcpyDeviceToHost, _s[0]);
( 2) cudaStreamSynchronize(_s[0]);
19   output_array(_host_a);
( 1) /*転送用ストリーム_s[1] で送った_dev_c の転送完了待ち*/
( 2) cudaStreamSynchronize(_s[1]);
20   add_array<<<N/32, 32, 0, _s[0]>>>(_dev_c, _dev_a);
( 1)  cudaMemcpyAsync(_host_a, _dev_a, N*sizeof(int),
                     cudaMemcpyDeviceToHost, _s[0]);
21   prod_array<<<N/32, 32, 0, _s[2]>>>
               (_dev_d, _dev_e, _dev_f);
( 1) cudaMemcpyAsync(_host_d, _dev_d, N*sizeof(int),
                     cudaMemcpyDeviceToHost,_s[2]);
( 2) cudaStreamSynchronize(_s[0]);
22   output_array(_host_a);
( 1) cudaStreamSynchronize(_s[2]);
23   output_array(_host_d);
          ⋮
(13) for (_i = 0 ; _i < 3 ; _i++)
(14)   cudaStreamDestroy(&_s[_i]);
24   }
```

図 7 データ転送コード生成
Fig. 7 Data Transfer Code

6. 評　　価

GPGPU でよく用いられるアルゴリズムについて MESI-CUDA の評価を行った．評価したプログラムは暗号解読，ヒストグラム算出，行列の転置，差分絶対値和を求める SAD，行列積である．各プログラムについて，プログラマがチューニングした CUDA コード，MESI-CUDA フレームワークの出力コード，非最適化 CUDA コードを用意し，実行時間を計測した．

最適化コードは転送・カーネル処理のタイミングを

© 2012 Information Processing Society of Japan

表 1 各アルゴリズム実行時間 (sec)
Table 1 Execution Time (sec)

	最適化	MESI-CUDA	非最適化
暗号解読	23.5	23.5	30.0
ヒストグラム	4.1	4.2	4.6
行列転置	11.4	11.7	14.3
SAD	127.1	127.1	127.8
行列積	165.9	165.9	166.3

最適化しているが，GPU 上のメモリはグローバルメモリしか使用せず，メモリ階層の有効活用は行っていない．非最適コードは転送とカーネルのオーバラップを行っていないコードである．つまり，本評価はデータ転送部分の最適化の評価となる．評価環境は Tesla C1060 を搭載した PC (core i7 930, 6GB Memory) を用いた．各プログラムの実行時間を表 1 に示す．

暗号解読，SAD，行列積については，MESI-CUDA でも最適化 CUDA コードとほぼ同じコードが出力され，同じ実行性能を得ることができた．一方，ヒストグラムと行列転置については，スケジューリングによる差が生じたため，最適化 CUDA コードより若干実行時間が増加している．しかし，すべてのプログラムにおいて最適化コードとほぼ遜色のない実行性能を示しており，非最適化コードに比べ 0.2 %から 21.7 %改善されている．したがってデータ転送に関しては，MESI-CUDA フレームワークはプログラマのコーディングの負担を減らす一方で，データ転送については手動最適化に近いコードを出力できたといえる．

7. おわりに

本稿では，GPGPU におけるデータ転送を自動化するフレームワーク MESI-CUDA を提案した．MESI-CUDA は，共有メモリ型のプログラミングモデルを採用することにより，GPGPU プログラミングにおいて不可欠であったプログラマによるデータ転送の記述を不要とし，自動でデータ転送コードを生成する．また，プログラミングモデルと静的解析・コード生成手法を述べた．さらに実行性能を評価し，データ転送については手動で最適化したコードと実行時間にほとんど差がないことも示した．

しかし，現状の最適化・コード生成手法および実装には不十分な点があり，プログラムによっては手動最適化コードより大きく性能が低下する可能性がある．

データ転送については，配列全体をアトミックに扱っているため，部分的な更新でも全体を転送してしまう，巨大な配列を段階的に処理できず扱える大きさがグローバルメモリのサイズに制約される，といった問題がある．これらの問題については，今後配列アクセスのインデックス解析を導入し，より効率的なコード生成の実現を計画している．

また，現状では GPU のメモリ階層を考慮した最適化は実現できていないため，メモリ周りを十分に手動チューニングした CUDA プログラムと同等の性能は得られない．現在の実装でも，カーネル関数内で明示的にグローバルメモリとのコピーを行うことにより，シェアードメモリの利用は可能である．しかし，テクスチャメモリ・コンスタントメモリについてはホスト上のスケジューリング結果との整合性が保証できないため，MESI-CUDA のトランスレート結果に対して手動で最適化を行ってから `nvcc` によりコンパイルするといった作業が必要となる．今後は，GPU のメモリ階層に対して共有変数を適切にマッピングすることで，メモリ使用方法の自動最適化を実現し，こうした低レベルな記述なしに高い実行性能が得られる処理系を目指す．

参考文献

1) J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.
2) GPGPU.org: General-Purpose computation on Graphics Processing Units. `http://www.gpgpu.org/`.
3) NVIDIA Developer CUDA Zone. `http://developer.nvidia.com/category/zone/cuda-zone`.
4) OpenCL - the open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`.
5) 道浦 悌, 大野 和彦, 佐々木 敬泰, and 近藤 利夫. GPGPU におけるデータ自動転送化コンパイラの提案. **先進的計算基盤システムシンポジウム** *SACSIS2011*, pages 221–222, 2011.
6) 道浦 悌, 大野 和彦, 佐々木 敬泰, and 近藤 利夫. GPGPU におけるデータ転送自動化コンパイラの設計. **情報処理学会研究報告** *2011-HPC-130(17)*, pages 1–9, 2011.
7) K. Ohno, D. Michiura, M. Matsumoto, T. Sasaki, and T. Kondo. A GPGPU Programming Framework based on a Shared-Memory Model. In *Parallel and Distributed Computing and Systems - 2011*, pages 310–318, 2011.
8) 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)*, pages 1–8, 2010.

ⓒ 2012 Information Processing Society of Japan

9) M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In Rajiv Gupta, editor, *Compiler Construction*, volume 6011 of *Lecture Notes in Computer Science*, pages 244–263. Springer Berlin / Heidelberg, 2010.
10) Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU compiler for memory optimization and parallelism management. *SIGPLAN Not.*, 45:86–97, 2010.

ⓒ 2012 Information Processing Society of Japan