修士論文

題目

GPGPUにおける
データ転送自動化フレームワーク
MESI-CUDA

指導教員

大野和彦講師

平成23年度

三重大学大学院 工学研究科 情報工学専攻
計算機アーキテクチャ研究室

道浦 悌 (410M526)

内容梗概

近年，GPU 上で汎用計算を実行する GPGPU が注目されている．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．

Abstract

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 MESI-CUDA programs can match for hand-optimized CUDA programs, automatically generating optimized data transfer code.

目 次

図 目 次

表 目 次

1　はじめに

GPU(Graphics Processing Units) は，画像処理専用のユニットであるが，近年 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，本研究ではデータ転送を自動化するフレームワーク MESI-CUDA（Mie Experimental Shared-memory Interface for CUDA) [5, 6, 7] を開発している．本フレームワークは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．ユーザに対しては，共有メモリ型の GPGPU プログラミングのモデルを提供する．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，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 について解説する．3 章では関連研究を紹介し， 4 章で MESI-CUDA の機能とプログラミングモデルについて説明する． 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．さらに，GPU には分岐予測機などが無く，単純な演算は高速であるが，制御を多数含むプログラムは動作が遅く，処理に適したものと適していないものがある．

従って，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 プログラムを容易に開発することができる．CUDA では，CPU をホスト，GPU をデバイスと呼ぶ．CUDA を用いたサンプルプログラムを図 2.1 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，その関数には修飾子`__device__`か`__global__` が付与される (図 2.1：4, 8 行)．修飾子のついていない関数や`__host__`の修飾子のついた関数はホスト側で実行される．ホスト側のコードから`__global__` の修飾子のついた関数を呼び出すことでカーネルを実行することができる (図 2.1：40, 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う．データ転送関数を図 2.2 に示す．データ転送の種類は 2 種類あり，ホストからデバイスへのデータ転送をする download 転送 (図 2.1：35-39 行) と，デ

```
 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 2.1: CUDA コードの例

```
// 同期式 Download 転送 (Host to Device)
cudaMemcpy(d, h, b, cudaMemcpyHostToDevice);
// 同期式 Readback 転送 (Device to Host)
cudaMemcpy(h, d, b, cudaMemcpyDeviceToHost);
// 非同期式 Download 転送 (Host to Device)
cudaMemcpyAsync(d, h, b, cudaMemcpyHostToDevice, s);
// 非同期式 Readback 転送 (Device to Host)
cudaMemcpyAsync(h, d, b, cudaMemcpyDeviceToHost, s);
```

図 2.2: CUDA におけるデータ転送

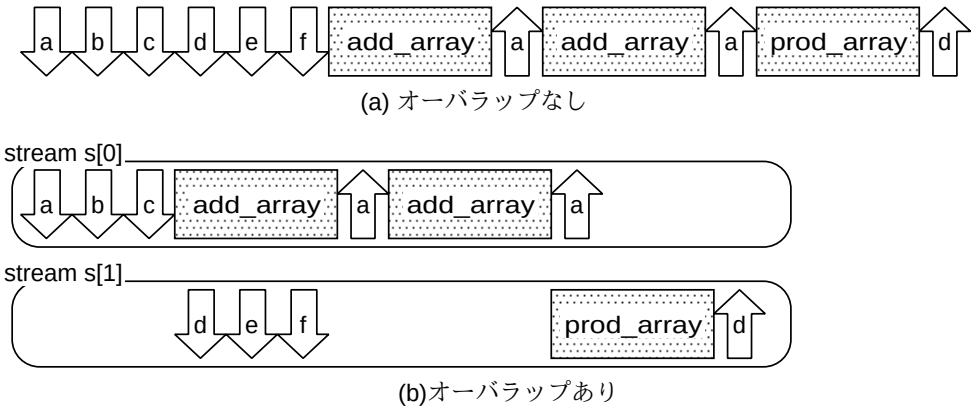

図 2.3: ストリームを用いた転送とカーネルのオーバラップ

バイスからホストへのデータ転送をする readback 転送 (図 2.1：41, 45, 47 行) である．カーネルを実行するためにはカーネルで使用するデータの download 転送が完了している必要があり，カーネル実行後にホストが参照するデータについては readback 転送が完了している必要がある．

また，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 つの方式がある．同期式転送と非同期式転送との処理の流れの違いを図 2.3 に示す．同期式の転送を用いた場合は，プログラムはデータ転送命令発行後，転送完了まで次のコードを実行せずに待機する．非同期式の転送を用いる場合は，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．しかし，非同期式のデータ転送の関数は，データ転送を行うように命令を発行するだけで，転送の完了を保証しない．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　ストリームとは依存関係のある非同期のデータ転送とカーネル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，カーネル処理がそれぞれ登録順に実行されていく．ストリームは生成 (図 2.1：27-29 行) と破棄 (図 2.1：64-65 行) を行う必要がある．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 2.1：35-39, 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 2.1：40, 44, 46 行)．このような指定を行うことで，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 2.1：21-26, 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の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 2.1：15-20, 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，メモリ階層についての支援ツールとして，自動的に各メモリ階層の特性に応じてデータの配置を自動的に行う研究 [10] がなされているが，GPU プログラムを解析して自動で割り当てるため，従来通りの GPU プログラミングを行う必要がある．

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，明示的なロックや排他制御，同期を不要にすることでプログラマの負担を減らしている．

4 MESI-CUDA の機能

4.1 MESI-CUDA 概要

MESI-CUDA フレームワークは, データ転送コードやメモリ確保・解放, ストリーム処理のコードを自動的に生成することで, ユーザの負担を軽減させる．ホストとデバイスへの処理の振り分けやカーネルの記述はユーザ自身が従来の CUDA に準じる形でコーディングを行う．このようなフレームワークにしたのは以下のような理由のためである．

- 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．
- デバイスに依存しプログラミングが困難なのは，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である．
- データの分割や転送は，並列アルゴリズムの本質ではない. しかし，GPGPU では，性能に大きく影響しデバイスへの依存性が高い．そのため，完全に 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，それらを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環境のモデルを採用し，ホスト・デバイス両方よりアクセス可能な共有変数を提供する．よって，ホスト関数・カーネル関数の違いによる変数の使い分けや，データ転送の記述が不要になる．

また，フレームワークで自動的に転送のタイミングやカーネル処理の順序を決定し，最適化を行う．この処理の中で，カーネル処理とデータ転送のオーバラップが可能なようにストリームの割り当てを行う．

図 2.1 の CUDA プログラムと等価な MESI-CUDA プログラムを図 4.4 に示す．カーネル関数に関する記述や，ホスト側での処理は CUDA と同様に行っている．その一方で，共有変数を用いることによって，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．

```
 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 4.4: CUDA コードと等価なフレームワークコード

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 の記述に準拠する．そのため，スレッド数の指定はユーザが行う．
- 変数に対しては共有メモリモデルを採用する．
  - 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．
  - データ転送に関する記述は不要である．

共有変数の宣言方法は，図 4.4，3 行目のように，変数宣言の修飾子として，`__share__`を付与する．

共有変数の値は，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．

4.2.1 本プログラミングモデルの利点

共有変数に対して上記のような一貫性を保証することで，並列処理において必須である同期の記述も不要としている．また，前述のように並列アルゴリズムの本質ではないデータ転送やストリーム処理などの記述が不要であり，簡潔なコーディングが可能である．さらに，デバイス固有のスペックに応じた最適化をフレームワーク内で自動で行うため，コードの移植性を高めている．C 言語に比べて大きく異なる点は，カーネルの記述のみで，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可能である．

4.2.2　本プログラミングモデルの欠点

低レベルな記述をフレームワークで隠蔽しているため，メモリ階層の有効活用や，クリティカルなデータ転送，カーネル実行の記述をユーザが行うことはできない．そのため，実行性能が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，カーネル実行の前にデータ転送命令を挿入すればよい．しかし，それだけでは前述のデータ転送とカーネル処理のオーバラップを行うための解析としては不十分である．そこで，データフロー解析も行う．この解析結果を基に依存関係のないカーネル処理とそのデータ転送を別々のストリームにすることで，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. スケジューリング
3. ストリーム割り当て
4. 同期ポイント解析
5. コード生成
   (a) メモリ確保・解放
   (b) ストリーム生成・破棄
   (c) 転送コード生成

5.1 データフロー解析

データ転送タイミングの決定やストリーム割り当てを行うために，データフロー解析を行う．複雑なフロー解析を行う必要はなく，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に対して解析すればよい．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し，変数の値自体を追跡する必要はない.

__share__の修飾子を付与して宣言された変数を変数表に登録し，これらの変数を基点として解析を行う． 各カーネルのデータフロー解析は, __global__の修飾子が付与された関数を基点として解析する.

ホストのデータフロー解析は main 関数を基点として解析を行う．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

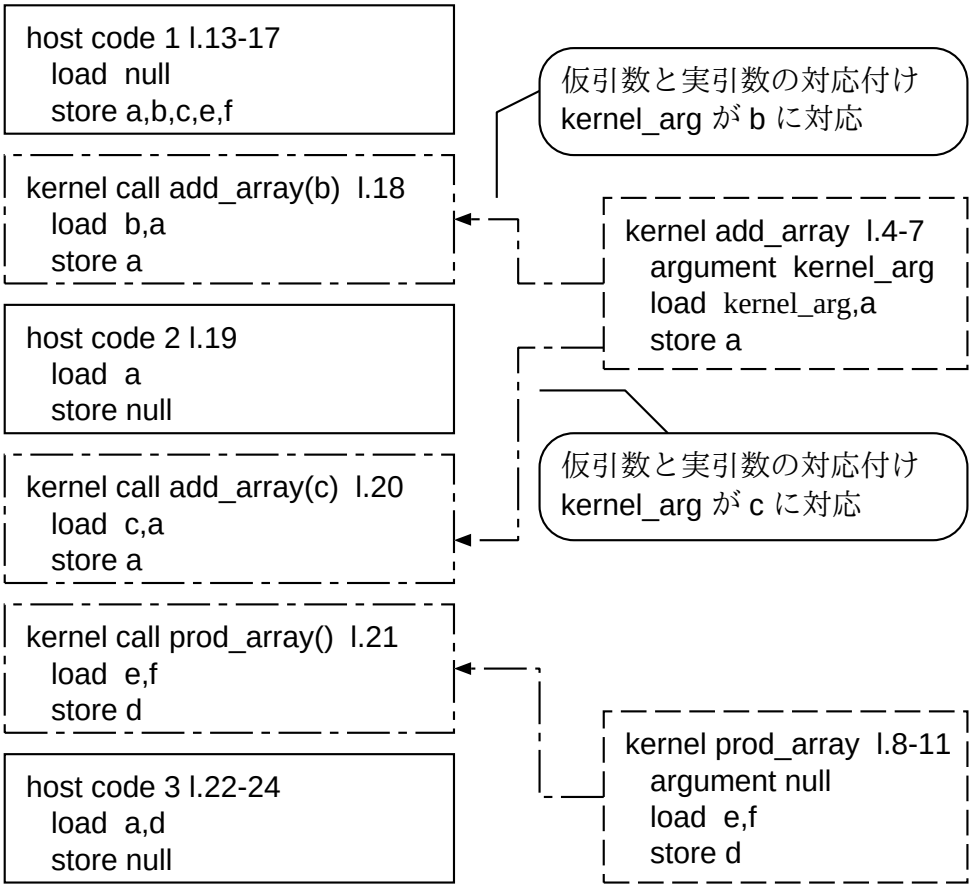

図 5.5: 共有変数の参照・代入の解析結果

るかを解析する．また，カーネルの呼び出し時の仮引数・実引数を対応付けする．図 4.4 における参照・代入の解析結果は図 5.5 のようになる．

次に，必要なデータ転送を解析する方法を示す．

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前のホストコードを参照し，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実行時にデータ転送が完了している可能性が高くなるため，代入直後にデータ転送コードを挿入する．

readback 転送　ホストにおける共有変数の参照についても同様に，参照を行う前にデータの 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で参照が行われているか確認する．参照されている共有変数については，readback 転送が必要となる．ホストにおける共有変数の参照時にはすでに転送が完了していることが望ましいため，カーネル呼び出しの直後に転送コードを挿入する．

これらの解析結果から求められる，図 4.4 に対する，データフローは図 5.6 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

前述のようにデータ転送とカーネル処理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，可能な限り多くのストリームを生成し，データ転送とカーネルを異なるストリームに割り当てることで，データ転送・カーネル処理のオーバラップが効率的に行われ，プログラムの実行効率が上がる．

しかし，ストリームの性質上，同一の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，同一の共有変数にアクセスしているカーネル関数やその変数のデータ転送は，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命令をはさみデータ転送・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，各カーネル・データ転送のストリーム割り当てを行う．共有変数の依存関係があるカーネルとデータ転送の集まりを**カーネルグループ**と呼ぶ．

1. 各カーネルとデータ転送についてカーネルグループ分けを行う．
2. 各カーネルグループ内の各カーネルについて，単一のカーネルでしか使用されていない変数の転送は別ストリームとする．
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，カーネルグループ内に複数のカーネルがあった場合オーバラップできずに最適にならない．そこで，カーネルグループ内の各カーネルで使用されている変数について解析する．各カーネルで単独に使用されている変数については，別にストリームをもうけて，オーバラップを行うことは可能

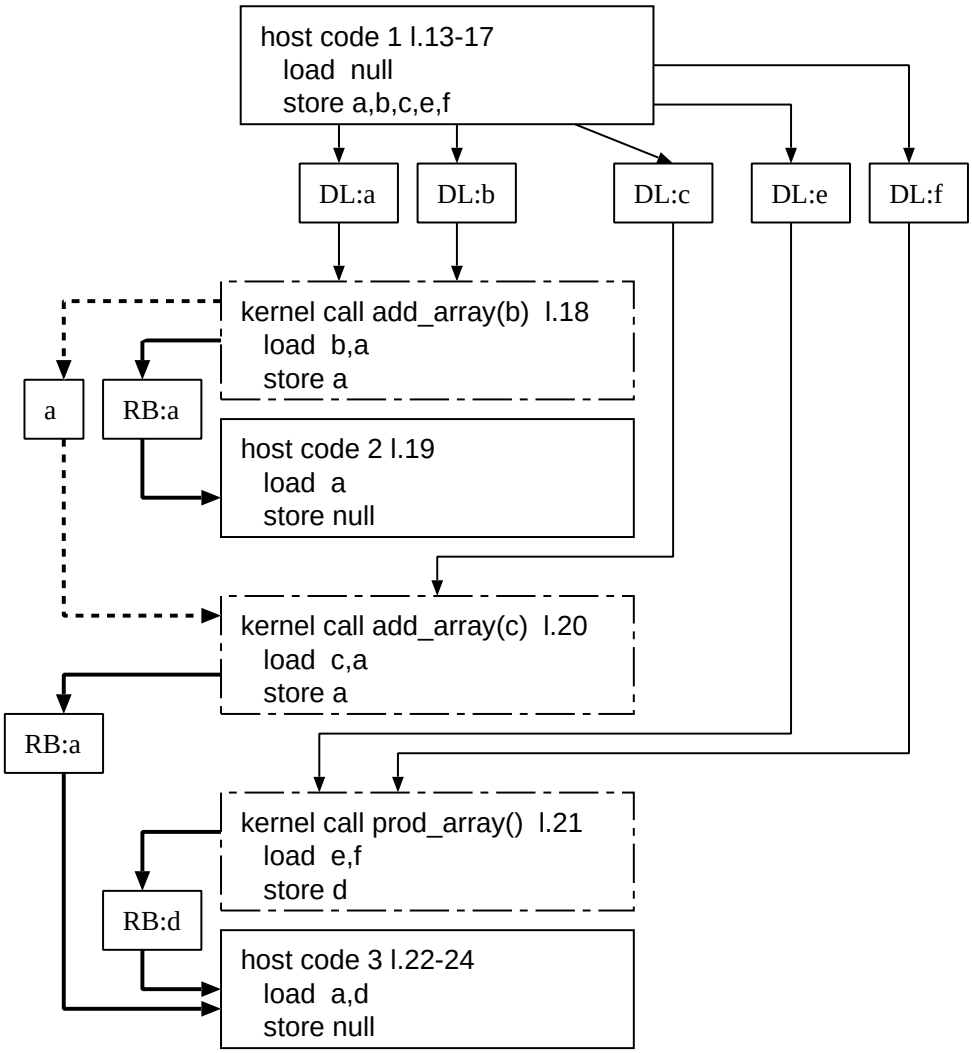
host code 1 l.13-17
load null
store a,b,c,e,f
DL:a
DL:b
DL:c
DL:e
DL:f
kernel call add_array(b) l.18
load b,a
store a
a
RB:a
host code 2 l.19
load a
store null
kernel call add_array(c) l.20
load c,a
store a
RB:a
kernel call prod_array() l.21
load e,f
store d
RB:d
host code 3 l.22-24
load a,d
store null

図 5.6: データフロー

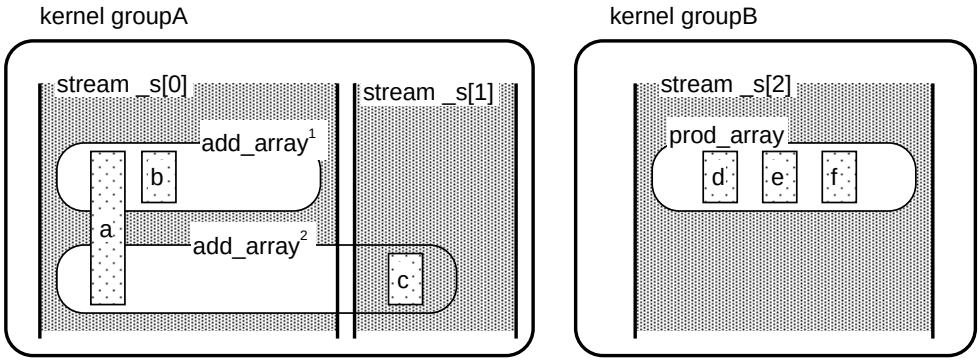

図 5.7: 変数・カーネル・ストリームの関係

である．そのため，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．しかし download 転送において，関連カーネルであるにも関わらずデータ転送を別ストリームで行った場合はストリームの同期を取る必要がある．

また，ホストにおける参照はストリームに所属させることはできないので，readback 転送についてもストリームの同期を取り，参照時にデータ転送を完了させて，古い値を参照しないようにする．

図 4.4 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 5.7 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．カーネルグループごとにストリームを割り当てただけでは本来オーバラップ可能なカーネル add_array(b) の実行と，次に実行されるカーネル add_array(c) で使用する変数 c のデータ転送のオーバラップができない．そこで，変数 c については別のストリームに所属させて転送する．ただし，カーネル add_array(c) は変数 c のストリームとは異なるストリームに所属し，実行される．そのため，カーネル add_array(c) を実行する前に変数 c に割り当てたストリームの同期処理を挿入し，カーネル実行時に変数 c の転送が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，スケジューリングの余地がある．また，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，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．そこで，カーネル呼び出しのコード位置を変えることが必要になる．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の代入がホストで完了した後である．よって，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 4.4 の場合では，カーネル `prod_array()` は，16-17 行の代入後，実行が可能な状態である．しかし，実際には 21 行で呼び出されている．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，カーネル呼び出しが同期待ちで実行されずに効率的ではない．カーネル `prod_array()` の呼び出しを 18 行に移すことでこの問題は解決する．

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．逆に，カーネルの実行が最初にある場合は，転送を早めに持ってくる方が GPU の使用効率は良くなる．

5.4 コード生成

5.4.1 メモリ確保・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* とし，デバイス用変数名は`_dev_`*varname* とする．また，各ホスト用変数，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．ホストコード，カーネル関数内の変数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 4.4 に対して，メモリ確保と解放コードを挿入し，ホストコード，カーネル関数内の変数アクセスをそれぞれホスト用変数，デバイス用変数に対応させ置換したコードを図 5.8 に示す．

```
                  ⋮
 4 __global__ void add_array(int *kernel_arg, int *_dev_a){
 5   int id = blockDim.x*blockIdx.x+threadIdx.x;
 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){
 9   int id = blockDim.x*blockIdx.x+threadIdx.x;
10   _dev_d[id] = _dev_e[id] * _dev_f[id];
11 }
12 int main(){
( 1) int *_host_a, *_dev_a;
( 2) int *_host_b, *_dev_b;
( 3) int *_host_c, *_dev_c;
( 4) int *_host_d, *_dev_d;
( 5) int *_host_e, *_dev_e;
( 6) int *_host_f, *_dev_f;
( 7) cudaMallocHost((void**)&_host_a,N*sizeof(int));
( 8) cudaMallocHost((void**)&_host_b,N*sizeof(int));
( 9) cudaMallocHost((void**)&_host_c,N*sizeof(int));
(10) cudaMallocHost((void**)&_host_d,N*sizeof(int));
(11) cudaMallocHost((void**)&_host_e,N*sizeof(int));
(12) cudaMallocHost((void**)&_host_f,N*sizeof(int));
(13) cudaMalloc((void**)&_dev_a,N*sizeof(int));
(14) cudaMalloc((void**)&_dev_b,N*sizeof(int));
(15) cudaMalloc((void**)&_dev_c,N*sizeof(int));
(16) cudaMalloc((void**)&_dev_d,N*sizeof(int));
(17) cudaMalloc((void**)&_dev_e,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);
( 2) cudaFreeHost(_host_b;
( 3) cudaFreeHost(_host_c);
( 4) cudaFreeHost(_host_d);
( 5) cudaFreeHost(_host_e);
( 6) cudaFreeHost(_host_f);
( 7) cudaFree(_dev_a);
( 8) cudaFree(_dev_b);
( 9) cudaFree(_dev_c);
(10) cudaFree(_dev_d);
(11) cudaFree(_dev_e);
(12) cudaFree(_dev_f);
24   }
```

図 5.8: メモリ確保と解放・変数の置き換え

5.4.2 ストリーム

ストリーム割り当てで決定したストリーム数だけストリームを宣言し，ストリームを生成する．プログラム先頭において cudaStream_t を用いてストリームを必要な数だけ宣言し，cudaStreamCreate 関数を用いて作成する．また，各カーネル呼び出しにおいて，各カーネルが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 データ転送コード

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, cudaMemcpyHostToDevice, 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, cudaMemcpyDeviceToHost, 所属ストリーム);
- **転送同期コード：**
  cudaStreamSynchronize(所属ストリーム);

図 4.4 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 5.9 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 5.9: データ転送コード生成

表 6.1: 実プログラム実行時間 (msec)

	最適化	MESI-CUDA	CPU
レイトレーシング	170.0	190.0	1960.0

表 6.2: 各アルゴリズム実行時間 (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

6 評価

MESI-CUDA の実プログラム評価を行うためにレイトレーシングを用いた．これは，1024x768pixel の画像を生成するもので，1 ラインの画像生成を 1 カーネルとして処理している．評価環境は CPU は Core i7 930，デバイスは TeslaC1060 を用いた．レイトレーシングプログラムについて，手動で最適化したコードと MESI-CUDA で出力したコード，CPU 上で実行される逐次コードのそれぞれで実行時間を計測した．実行時間を表 6.1 に示す．最適化コードと MESI-CUDA の実行時間は，データ転送とレイトレーシング処理の時間であり，画像出力のために用いている OpenGL の初期化などの時間は含まれない．CPU 実行時間はレイトレーシング処理にかかった時間のみで，こちらにも初期化の時間などは含まれていない．MESI-CUDA 出力コードは最適化コードに比べて，20msec の実行時間の増加に抑えることができた．これは，メモリ階層の有効的な使い分けと，配列の分割転送による効率的なカーネル処理・データ転送のオーバラップによる差である．CPU のみで逐次実行した場合に比べては，MESI-CUDA は 10 倍以上の性能を示している．

また，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，差分を取って絶対値の総和を求める SAD，行列積を行うプログラムである．それぞれの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，データ転送とカーネル実行のオーバラップを全く行っていない非最適化 CUDA

コードを用意し，実行時間を計測した．評価環境にはレイトレーシングと同様に TeslaC1060 を用いた．それぞれのプログラムの実行時間を表 6.2 に示す．実行時間が最適化コードと MESI-CUDA で同じである暗号解読，SAD，行列積については，ほぼ同じコードを出力することができた．一方，実行時間が異なるヒストグラム，行列転置については，スケジューリングによる差が生じている．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 0.4 から 6.5 秒改善されている．

これらの二つの評価から，MESI-CUDA はプログラマのコーディングの負担を減らしつつ，最適なコードに近いコードを出力することができたといえる．今回用いた実プログラムは 20msec の実行時間の増加であり，よく用いられるアルゴリズムについてもわずかなオーバヘッドに抑えることができた，従って，MESI-CUDA は幅広いプログラムに対してプログラマが転送コードを記述せずに最適に近いコードを出力することができる．また，CPU 上で実行するよりも MESI-CUDA は 10 倍速い性能を示しており，MESI-CUDA を用いると C 言語ライクなプログラミングでありながら，GPU の高性能を活用することができる．今後の課題として，メモリ階層の使い分けとインデックス解析による配列の分割転送，スケジューリングアルゴリズムを MESI-CUDA に実装し，実行効率の低下を抑えていく．

7　おわりに

本論文では 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．これは，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フレームワークである．また，MESI-CUDA のプログラミングモデルや，データフロー解析，ストリームの割り当て，コード生成を行う方法について示した．実行性能について評価し，最適なコードとほとんど差がない実行時間で動作することも示した．

しかし，現状のフレームワークには問題点もある．配列のインデックス解析を行っておらず，カーネル内で配列の一部しか操作しない場合も，配列全体を転送してしまいオーバヘッドが発生する．また，分岐などの制御に対する解析方法を確立していないため，分岐パス内で参照・代入がある場合はカーネル実行の直近でしかデータ転送を行えない可能性がある．カーネル処理で使用するデータがグローバルメモリ内に収まりきらない場合，転送スケジューリングアルゴリズムが不十分なため，実行できないこともある．これらは今後の課題として改善していく．

謝辞

本研究を行うに当たり日頃ご指導頂きました近藤利夫教授，大野和彦講師，佐々木敬泰助教に深く感謝致します．また，日頃お世話になりました計算機アーキテクチャ研究室の皆様に感謝致します．

参考文献

[1] John D. Owens,David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, Timothy J. Purcell, A Survey of General-Purpose Computation on Graphics Hardware

[2] GPGPU.org, http://gpgpu.org/

[3] nVidia CUDAZone, http://www.nvidiaco.jp/object/cuda_home_new_jp.html

[4] OpenCL, http://www.khronos.org/opencl/

[5] 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 SACSIS2011, 221-222

[6] 道浦 悌, 大野 和彦, 佐々木 敬泰, 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), 1-9, 2011-07-20

[7] Kazuhiko Ohno, Dai Michiura, Masaki Matsumoto, Takahiro Sasaki, Toshio Kondo, A GPGPU Programming Framework based on a Shared-Memory Model, Parallel and Distributed Computing and Systems - 2011

[8] 中村 晃一, 林崎 弘成, 稲葉 真理, 平木 敬, SIMD 型計算機向けループ自動並列化手法, 情報処理学会研究報告 2010-HPC-126(10), 1-8, 2010-07-27

[9] Muthu Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *Compiler Construction*, volume 6011 of *Lecture Notes in Computer Science*, pages 244–263. Springer Berlin / Heidelberg, 2010.

[10] YiYang, PingXiang, JingfeiKong, HuiyangZhou, A GPGPU Compiler for Memory Optimization and Parallelism Management

A サンプルの MESI-CUDA コードに対する出力コード全文

```
#include <stdio.h>
#define N 3200

__global__ void add_array(int *x, int *_dev_a){
  int id = blockDim.x*blockIdx.x+threadIdx.x;
  _dev_a[id] = _dev_a[id] + x[id];
}

__global__ void prod_array(int *_dev_d, int *_dev_e, int *_dev_f){
  int id = blockDim.x*blockIdx.x+threadIdx.x;
  _dev_d[id] = _dev_e[id] * _dev_f[id];
}

int main(){
  int *_host_a, *_dev_a;//__share__ a[N];
  int *_host_b, *_dev_b;//__share__ b[N];
  int *_host_c, *_dev_c;//__share__ c[N];
  int *_host_d, *_dev_d;//__share__ d[N];
  int *_host_e, *_dev_e;//__share__ e[N];
  int *_host_f, *_dev_f;//__share__ f[N];

  //メモリ確保ホスト側
  cudaMallocHost((void**)&_host_a,N*sizeof(int));
  cudaMallocHost((void**)&_host_b,N*sizeof(int));
  cudaMallocHost((void**)&_host_c,N*sizeof(int));
  cudaMallocHost((void**)&_host_d,N*sizeof(int));
  cudaMallocHost((void**)&_host_e,N*sizeof(int));
  cudaMallocHost((void**)&_host_f,N*sizeof(int));

  //メモリ確保デバイス側
  cudaMalloc((void**)&_dev_a,N*sizeof(int));
  cudaMalloc((void**)&_dev_b,N*sizeof(int));
```

```
cudaMalloc((void**)&_dev_c,N*sizeof(int));
cudaMalloc((void**)&_dev_d,N*sizeof(int));
cudaMalloc((void**)&_dev_e,N*sizeof(int));
cudaMalloc((void**)&_dev_f,N*sizeof(int));

//ストリーム生成
cudaStream_t _s[3];
int _i;
for (_i = 0 ; _i < 3 ; _i++){
  cudaStreamCreate(&_s[_i]);
}

//ロードとストリーム_s[0] による download 転送
load_array(_host_a);
cudaMemcpyAsync(_dev_a, _host_a, N*sizeof(int),
                cudaMemcpyHostToDevice, _s[0]);
load_array(_host_b);
cudaMemcpyAsync(_dev_b, _host_b, N*sizeof(int),
                cudaMemcpyHostToDevice, _s[0]);

//ロードと転送用ストリーム_s[1] による download 転送
load_array(_host_c);
cudaMemcpyAsync(_dev_c, _host_c, N*sizeof(int),
                cudaMemcpyHostToDevice, _s[1]);

//ロードとストリーム_s[2] による download 転送
load_array(_host_e);
cudaMemcpyAsync(_dev_e, _host_e, N*sizeof(int),
                cudaMemcpyHostToDevice, _s[2]);
load_array(_host_f);
cudaMemcpyAsync(_dev_f, _host_f, N*sizeof(int),
                cudaMemcpyHostToDevice, _s[2]);

//カーネル実行 a=a+b;
add_array<<<N/32, 32, 0, _s[0]>>>(_dev_b, _dev_a);
```

```
//a[N] の readback 転送
cudaMemcpyAsync(_host_a, _dev_a, N*sizeof(int),
                cudaMemcpyDeviceToHost, _s[0]);
//a[N] の転送完了を保証するための同期
cudaStreamSynchronize(_s[0]);
output_array(_host_a);

//転送用ストリーム_s[1] で送った c[N] の転送完了待ち
cudaStreamSynchronize(_s[1]);
//カーネル実行 a=a+c;
add_array<<<N/32, 32, 0, _s[0]>>>(_dev_c, _dev_a);
 //a[N] の readback 転送
cudaMemcpyAsync(_host_a, _dev_a, N*sizeof(int),
                cudaMemcpyDeviceToHost, _s[0]);

//カーネル実行 d=e*f;
prod_array<<<N/32, 32, 0, _s[2]>>>(_dev_d, _dev_e, _dev_f);
//d[N] の readback 転送
cudaMemcpyAsync(_host_d, _dev_d, N*sizeof(int),
                cudaMemcpyDeviceToHost,_s[2]);

//a[N] の転送完了を保証するための同期
cudaStreamSynchronize(_s[0]);
output_array(_host_a);

//d[N] の転送完了を保証するための同期
cudaStreamSynchronize(_s[2]);
output_array(_host_d);

//ストリーム破棄
for (_i = 0 ; _i < 3 ; _i++){
  cudaStreamDestroy(&_s[_i]);
}

//メモリ解放ホスト側
```

```
  cudaFreeHost(_host_a);
  cudaFreeHost(_host_b);
  cudaFreeHost(_host_c);
  cudaFreeHost(_host_d);
  cudaFreeHost(_host_e);
  cudaFreeHost(_host_f);

  //メモリ解放デバイス側
  cudaFree(_dev_a);
  cudaFree(_dev_b);
  cudaFree(_dev_c);
  cudaFree(_dev_d);
  cudaFree(_dev_e);
  cudaFree(_dev_f);
}
```