Vol.2011-HPC-130 No.17
2011/7/27

GPGPUにおけるデータ転送自動化コンパイラの設計

道浦 悌[†1] 大野和彦[†1]
佐々木敬泰[†1] 近藤利夫[†1]

近年の GPU の演算性能の向上により，GPU 上で汎用演算を実行させる GPGPU への関心が高まっている．また，nVIDIA 社から開発環境である CUDA がリリースされるなど，GPU プログラミングは容易になりつつある．しかし CUDA を用いても，GPGPU では，メインメモリ・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．そこで本論文では，GPGPU におけるデータ転送処理を自動化するコンパイラを提案する．本コンパイラは CUDA コードに対して，データ転送コードを自動的に生成する．これにより，プログラマの開発の負担を軽減させる．また，CUDA はデータ転送のオーバヘッドを低減させるために，カーネルの実行とデータ転送のオーバラップが可能なストリーム機能を用意している．そこで，本コンパイラはこのストリーム機能を適切な箇所で用いることによって，プログラムの実行効率を高めている．本コンパイラの性能を示すために提案する手法に基づく想定出力コードを作成し，最適化したコードと実行時間を比較し，評価を行った．その結果，評価に用いた暗号読解プログラムの場合，おおよそ 1 秒程度のオーバヘッドに留まった．

Automatic generation of data transfer code for GPGPU compiler

Dai Michiura,[†1] Kazuhiko Ohno,[†1] Takahiro Sasaki[†1] and Toshio Kondo[†1]

This paper presents a compiler which automatically generates data transfer code for GPGPU. The user does not need to specify data transfer functions of CUDA, reducing the burden of GPGPU programming. The data transfer time is often large overhead using GPU and decline the performance, sometimes slower than using CPU. However, the data transfer can be overlapped with the execution of kernels on GPU, and our compiler automatically utilize this feature. We hand-compiled a cryptanalysis program using the scheme of our compiler and compared with hand-optimized equivalent code. The difference of performance was approximately 1 second.

1. はじめに

近年 GPU は CPU に比べ性能向上がめざましく，ムーアの法則をしのぐ性能向上[1] を見せている．そのため，GPU に汎用的な演算を行わせる GPGPU[2] への関心が高まっている．また，CUDA[3] や OpenCL[4] などで GPU プログラミングがサポートされており GPGPU への敷居は低くなってきている．しかし，CUDA や OpenCL などを用いてもハードウェアの構成上，プログラマがアーキテクチャにあわせたプログラミングと最適化を行う必要がある．GPGPU は，ホスト (CPU 側) とデバイス (GPU 側) に分かれており，プログラマは明示的にホストメモリ・デバイスメモリ間のデータ転送を行う必要がある．このため，他のプログラミングに比べプログラマの負担が大きい．そこで，我々はこのデータ転送処理に着目し，データ転送の自動化コンパイラを提案している[5]．このコンパイラは，ホストメモリ・デバイスメモリ間のデータの転送コードを自動的に生成，挿入を行うことで，データ転送を自動化し，GPU プログラミングの負担を軽減する．

以下では，2 章で背景として，関連研究と CUDA の概要を紹介する．3 章で本コンパイラの機能の概要について述べる．4 章で本コンパイラの機能を実現させるための手法を説明する．5 章では本コンパイラの有用性を示すために，本コンパイラの提案手法により生成した CUDA コードの評価を行う．最後に，6 章でまとめる．

2. 背景

2.1 関連研究

CUDA を支援する研究は数多くなされている．アプリケーション開発より派生した研究として，超高精細の衛星画像処理を例とした，デバイスメモリに収まりきらないデータを処理する手法に関する研究[6] がある．また，ストリーム処理の支援を行うミドルウェア[7] の方式をとっているものもある．for 文などのループ文に限るが自動で並列化を行う研究[8] や，メモリ階層への最適なデータ配置を自動で行う研究[9] がコンパイラ方式でなされている．我々は，コード生成に適しているコンパイラ形式での研究に着目し，データ転送の自動化をコンパイラで実現する．

†1 三重大学大学院工学研究科
Graduate School of Engineering, Mie University

```
//download 転送 (ホストからデバイスへの転送)
cudaMemcpy(val_dev, val_host, N*sizeof(int), cudaMemcpyHostToDevice);
//readback 転送 (デバイスからホストへの転送)
cudaMemcpy(val_host, val_dev, N*sizeof(int), cudaMemcpyDeviceToHost);

//非同期 download 転送 (ホストからデバイスへの転送)
cudaMemcpyAsync(val_dev, val_host, N*sizeof(int), cudaMemcpyHostToDevice,streamA);
//非同期 readback 転送 (デバイスからホストへの転送)
cudaMemcpyAsync(val_host, val_dev, N*sizeof(int), cudaMemcpyDeviceToHost,streamA);
```

図 1 CUDA におけるデータ転送
Fig. 1 Data transfer functions of CUDA

2.2 CUDA の概要

CUDA の概要と現状の CUDA におけるデータ転送について述べる．CUDA とは nVIDIA 社より提供されている SDK で，GPU プログラムを容易に開発することができる．CUDA は，C 言語ベースで開発されていて，GPU プログラミング用のライブラリを呼び出すことで，GPU 上で動作するプログラムの記述を簡単にしている．CUDA では，CPU をホスト，GPU をデバイスとしている．デバイス上で実行するコードをカーネルとして記述し，ホスト上で実行されるコードと区別されている．カーネルはホスト側より呼び出される．また，CUDA では，プログラマが明示的にホストメモリ・デバイスメモリ間のデータ転送を行う必要がある．データ転送は 2 種類あり，ホストメモリからデバイスメモリへのデータ転送である download 転送と，デバイスメモリからホストメモリへのデータ転送である readback 転送である．

2.2.1 CUDA におけるデータ転送

CUDA におけるデータ転送関数を**図 1** に示す．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転送が完了している必要がある．そのため，効率的にデータ転送を実行し，GPU がデータ転送の完了待ちで，カーネルを実行できない時間をなるべく短くしなければならない．しかし，多数のカーネルが様々なタイミングで実行されるため，高効率なデータ転送を行うプログラムを記述することは難しい．加えて，用いるデバイスによってデバイスメモリの容量も異なるため，デバイスごとに，データ転送の最適化を行う必要があり，コードの移植性にも問題がある．さらに，非同期式のデータ転送を用いることでカーネル実行とデータ転送をオーバラップすることが可能となるが，この非同期式のデータ転送は，CUDA の機能であるストリームを用いなければ実現できない．ストリームとは関連性のある変数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，カーネル実行がそれぞれ登録順に実行されていく．また，実行中のカーネルとデータ転送が異なるストリームに属している場合，カーネル実行とデータ転送を同時に実行することが可能となり，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，データ転送にも細心の注意をはらいプログラミングを行わなければならず，プログラマの負担が増える．

また，データ転送で用いる変数もホスト側とデバイス側のそれぞれを宣言する必要がある．従ってホスト，デバイスどちらで実行させるコードかにより参照する変数を使い分ける必要がある．

2.2.2 CUDA4.0

CUDA4.0 では，Unified Virtual Addressing がサポートされている．今までの CUDA のプログラミングモデルは，メモリ空間をホストと各デバイスの別々のアドレス空間として扱う必要があった．一方，CUDA4.0 からサポートされた，Unified Virtual Addressing を用いることで，プログラマはメモリ空間を単一のアドレス空間として扱うことが可能となった．これにより，図 1 の cudaMemcpy 関数や cudaMemcpyAsync 関数といったメモリ転送関数の第 4 引数である転送の方向を，cudaMemcpyDefault と記述するだけで済むようになった．しかし，プログラマによる明示的なデータ転送は依然として必要である．

3. データ転送自動化コンパイラの機能

本章では提案するコンパイラの機能の概要について述べる．本コンパイラは，CUDA コードに対して，データ転送コードを自動的に生成することで，プログラマが明示的なデータ転送コードを記述しなくてすむようにする．さらに，デバイスに応じて転送を最適化することで，自動的に効率のよいコードを得ることができる．また，本コンパイラではホスト側，デバイス側の両コードで同じ変数をグローバル変数のように扱うことができる．これにより，ホスト側とデバイス側のそれぞれで行っていた変数管理の負担が低減される．CUDA における配列の差分をとるコードを**図 2** に示す．本コンパイラを用いた場合，図 2 のコードと等価なコードは**図 3** のようになる．

3.1 自動転送の形式

本コンパイラは 3 種類の転送形式をサポートする．本コンパイラは download 転送の自動化を行う download 形式，readback 転送の自動化を行う readback 形式，どちらの性質も持つ share 形式の 3 つの形式を提供する．宣言方法は download 形式，readback 形式，share

```
1  #include <stdio.h>
2  #define N 320
3
4  __device__ void dev_sub(int *picA_dev,int *picB_dev){
5      int id=blockDim.x*blockIdx.x+threadIdx.x;
6      picA_dev[id]=picA_dev[id]-picB_dev[id]
7  }
8  __global__ void call_sub(int *picA_dev,int *picB_dev){
9    dev_sub(picA_dev,picB_dev);
10 }
11
12 int main(){
13   int *picA_host;  int *picA_dev;
14   int *picB_host;  int *picB_dev;
15
16   //メモリ確保
17   cudaMallocHost((void**)&picA_host,N*sizeof(int));
18   cudaMalloc((void**)&picA_dev,N*sizeof(int));
19   cudaMallocHost((void**)&picB_host,N*sizeof(int));
20   cudaMalloc((void**)&picB_dev,N*sizeof(int));
21
22   readpic(picA_host); readpic(picB_host);
23
24   //データ転送（download）
25   cudaMemcpy(picA_dev, picA_host, N*sizeof(int), cudaMemcpyHostToDevice);
26   cudaMemcpy(picB_dev, picB_host, N*sizeof(int), cudaMemcpyHostToDevice);
27   //カーネル実行
28   call_sub<<<N/32,32>>>(picA_dev,picB_dev);
29   //データ転送（readback）
30   cudaMemcpy(picA_host, picA_dev, N*sizeof(int), cudaMemcpyDeviceToHost);
31
32   outputpic(picA_host);
33
34   //メモリ解放
35   cudaFree(picA_dev);
36   cudaFree(picB_dev);
37   cudaFreeHost(picA_host);
38   cudaFreeHost(picB_host);
39 }
```

図 2 CUDA オリジナルコード
Fig. 2 Original code for CUDA

```
1  #include <stdio.h>
2  #define N 320
3
4  __device__ void dev_sub(){
5    int id=blockDim.x*blockIdx.x+threadIdx.x;
6    picA_dev[id]=picA_dev[id]-picB_dev[id]
7  }
8
9  __global__ void call_sub(){
11   dev_sub();
11 }
12
13 int main(){
14   __share__ int picA[N];
15   __download__ int picB[N];
16
17   readpic(picA); readpic(picB);
18
19   //カーネル実行
20   call_sub<<<N/32,32>>>();
21
22   outputpic(picA);
23
24 }
```

図 3 本コンパイラを用いたコード
Fig. 3 Code for proposed compiler

```
__download__ int temp;
__readback__ int res;
__share__ int array[256];
```

図 4 本コンパイラにおける変数宣言方法
Fig. 4 Variable declaration for proposed compiler

形式それぞれで，変数宣言時に変数の前に`__download__`，`__readback__`，`__share__`とつけるだけでよい．**図 4** は宣言方法の例である．

download 形式

download 転送のみ自動化を行う．不必要な readback 転送を行わないことをねらいとしている．例えば，カーネルで参照されるだけのデータ，もしくはカーネル実行後，ホスト側が参照しないデータを格納する変数の宣言に用いる．

readback 形式

readback 転送のみ自動化を行う．不必要な download 転送を行わないことをねらいとしている．例えば，カーネルで代入が発生し，ホスト側で参照しか行われない，結果として必要なデータを格納する変数の宣言に用いる．

share 形式

download 転送，readback 転送の自動化を行う．ホスト側，カーネル両方で代入，参照などが発生する変数に対して用いる．どの形式を用いればよいかわからない場合はこの形式を使用すればよい．しかし，不必要なデータ転送が発生し，オーバヘッドとなってしまう可能性がある．

3.2 データ転送

本コンパイラではカーネル実行とデータ転送のオーバラップを実現することで，オーバヘッドの削減を行う．本節では，3 つのカーネルを実行する場合を例にデータ転送の流れを説明する．**図 5** はデータ転送やカーネルの実行を時系列で示したものである．カーネルの実行状況が複雑な場合，一つのカーネルに対し単純に download 転送，カーネル実行，readback 転送を一連の動作としないとデータ転送の管理が煩雑となる．しかし，そのように一連の流れとしてカーネルを実行した場合，図 5 左に示すようにカーネルのデータ転送完了待ち時間が長くなってしまい，GPU にアイドルタイムが発生する．従って，データ転送の最適化を行わないと GPU のアイドルタイムが増大し，GPU で処理を行わせる利点が失われる可能性がある．

本コンパイラでは，図 5 右に示すようにデータ転送とカーネルの実行をオーバラップするようにコードを生成することで，データ転送時間のオーバヘッドを低減する．データ転送とカーネルの実行のオーバラップのためにストリームを用いる．各カーネルが使用するデータを静的解析し，データとカーネルの関係を明らかにする．関連のあるデータとカーネルを一つのストリームでまとめ，依存関係のないデータやカーネルは別ストリームとしてまとめる．それぞれのストリームごとにデータ転送命令を挿入することで自動転送を実現する．また，デバイスメモリの容量などを参照して最適化を行うことで，自動で GPU のスペックに合わせた最適化をプログラムの書き換えなしに行うことができる．

3.3 処理全体の流れ

処理全体の流れを以下に示す．

(1) 変数処理
- 変数表登録
- 変数拡張

(2) ストリーム割り当て

(3) データ転送自動化

(4) 後処理

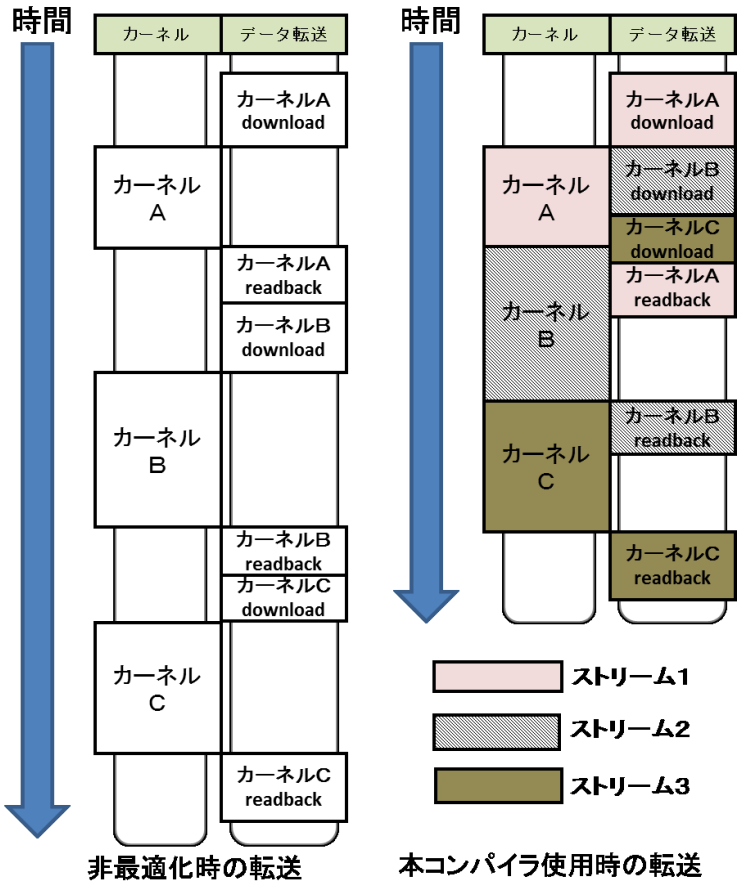

図 5 データ転送の流れ
Fig. 5 Data transfer diagram

download 形式，readback 形式，share 形式それぞれで宣言された変数を変数表に登録し，管理を行う．それぞれの形式で宣言された変数をホストで用いる変数とデバイスで用いる変数に拡張し，メモリを確保する．また，各変数とカーネル間の依存関係を解析し，所属させるストリームを決定する．変数において，download 転送，readback 転送が可能な範囲を解析し，適切な場所にデータ転送命令を挿入する．最後にプログラム終了直前に，メモリ解放，ストリームの破棄などを行う．

4. データ転送自動化コンパイラの設計

本章ではコンパイラの機能を実現するための手法について**図 6** のコードを用いて説明する．最初に，本コンパイラにおけるプログラミングモデルを示す．次に，各処理について，手法を示す．そして，図 6 のコードを用いた具体例で説明する．本コンパイラはデバイスメモリのサイズに応じた対応を行う予定であるが，本論文では，すべてのデータがデバイスメモリ上に格納できる場合の手法を示す．デバイスメモリのサイズに応じた処理の手法は，今後の課題である．

4.1 本コンパイラのプログラミングモデル

本コンパイラは，実行中のカーネルが使用する変数に対して，カーネルのみからしか代入，参照できない．カーネル実行中にホスト側で変数に代入し，デバイスへデータ転送することで，カーネルに影響を与えることはできない．ホスト側で変数に代入を行う必要がある場合は，カーネル呼び出し前に行わなければならないものとする．ホスト側で変数の参照を行う必要がある場合は，カーネル呼び出し後に行われなければならない．呼び出されたカーネルの実行は，呼び出し後のホストのコードの実行と並列に行われるが，ホスト側で変数を参照した時点で，自動的に同期を行う．つまり，対応するカーネルがまだ実行中であれば，そのカーネルの終了までホスト側の実行は中断される．

4.1.1 本プログラミングモデルの利点

上記のように，本プログラミングモデルにより，データに対して一貫性を保証することで，プログラマによる明示的な同期命令を不要とする．また，カーネルのみが並列で処理されるため，ホスト側の動作は，カーネルを C 言語における関数呼び出しと同等と見なすことができる．そのため，C 言語のようなプログラミングモデルとしてプログラミングが可能である．

4.1.2 本プログラミングモデルの欠点

一方，実際には，カーネルの処理結果を参照するまではホスト側の実行も並列に行われ

```
1  #include <stdio.h>
2  #define N 320
3
4  __device__ void dev_sub1(){
5  int id=blockDim.x*blockIdx.x+threadIdx.x;
6    picA[id]=picA[id]-picB[i];
7  }
8  __device__ void dev_sub2(){
9  int id=blockDim.x*blockIdx.x+threadIdx.x;
10   picA[id]=picA[id]-picC[i];
11 }
12 __device__ void dev_sub3(){
13 int id=blockDim.x*blockIdx.x+threadIdx.x;
14   picD[id]=picE[id]-picF[i];
15 }
16 __global__ void call_sub1(){dev_sub1();}
17 __global__ void call_sub2(){dev_sub2();}
18 __global__ void call_sub3(){dev_sub3();}
19
20 int main(){
21   __share__ int picA[N];
22   __download__ int picB[N],picC[N];
23   __readback__ char picD[N];
24   __download__ char picE[N], picF[N];
25
26   readpic(picA); readpic(picB); readpic(picC);
27
28   readpic(picE); readpic(picF);
29
30   call_sub1<<<N/32,32>>>();
31
32   outputpic(picA);
33   readpic(picA);
34
35   call_sub2<<<N/32,32>>>();
36
37   call_sub3<<<N/32,32>>>();
38
39   outputpic(picA);
40
41   outputpic(picD);
42
43 }
```

図 6 本コンパイラにおける複数カーネルプログラムの例
Fig. 6 Example code for proposed compiler

表 1　変数表
Table 1　Variable table

	__share__ int picA	__download__ int picB	__readback__ char picD
プログラマが宣言した変数名	picA[N]	picB[N]	picD[N]
転送形式	share	download	readback
変数の型	int	int	char
変数サイズ	320*sizeof(int)	320*sizeof(int)	320*sizeof(char)
ホスト側変数名	*_host_picA	*_host_picB	*_host_picD
ホスト側変数アドレス	_host_picA	_host_picB	_host_picD
デバイス側変数名	*_dev_picA	*_dev_picB	*_dev_picD
デバイス側変数アドレス	_dev_picA	_dev_picB	_dev_picD
所属ストリーム	(_stream[0])	(_stream[0])	(_stream[1])

る．しかし，カーネル実行中にホスト側で変数へのアクセスができないため，ホスト上での実行と，デバイス側での実行のオーバラップができない可能性があり，オーバヘッドが発生する場合がある．また，カーネル実行途中でホスト側において変数のアクセスができないため，カーネルの途中でホスト側から変数の値の更新ができず，このような値の更新が必要なカーネルの記述ができない．そういったアルゴリズムを記述する場合には，カーネルを分割して記述することで実現する必要がある．

4.2 変数処理

download 形式，readback 形式，share 形式それぞれで宣言された変数を変数表に登録する．図 6 のコードを処理した場合の変数表を**表 1** に示す．

最初に，各転送形式で宣言された変数について，変数表への登録を行う．変数宣言の形式を参照し，プログラマが宣言した変数名，転送形式，変数の型，変数のサイズを登録する．ホスト側変数名は”_host_+プログラマが宣言した変数名”とし，デバイス側変数名は”_dev_+プログラマが宣言した変数名”とする．ただし，所属ストリームの項目はストリーム割り当て処理で決定するので，この時点では未設定である．表 1 の括弧内はストリーム割り当て処理で登録される内容である．変数表登録後は，変数宣言の置き換えを行う．各形式で宣言された変数を，ホスト側変数名，デバイス側変数名で置き換え，それぞれ cudaMallocHost 関数，cudaMalloc 関数を用いてメモリ領域を確保する．次に，本コンパイラは，ホスト側で動作するプログラムの変数とデバイス側で動作するプログラムどちらでも同じ変数の使用を許している．そのため，プログラマが宣言した変数名で参照されている変数を置き換える必要がある．ホスト側で動作するプログラムのプログラマが宣言した変数名はホスト側変数名で置き換え，デバイス側で動作するカーネルのプログラマが宣言した変数名はデバイス側変数名で置き換える．さらに，各関数で必要になる変数について各関数に引数を付与する．プログラム終了直前に，ホスト側変数は cudaFreeHost 関数，デバイス側変数は cudaFree 関数を用いて，メモリを解放する．

図 6 の [21-24] 行目の変数に対しての変数処理を例に説明する．[21-24] 行目の変数を変数表に登録する．所属ストリームの解析はストリーム処理で行う．picA[N]，picB[N]，picD[N] の変数表は表 1 のようになる．次に，各変数を変数表のホスト側変数名，デバイス側変数名の通り宣言するように [21-24] 行目を置き換える．宣言直後の [25] 行目で，メモリ確保を行うため，各ホスト側変数では cudaMallocHost 関数，各デバイス側変数では cudaMalloc 関数を用いたメモリ確保命令を挿入する．また，ホストプログラムでの参照，カーネルでの参照はそれぞれホスト側変数名，デバイス側変数名で行う必要がある．そのため，[26,28,32-33,39,41] 行目のホストプログラムにおける各変数名をホスト側変数名に置き換える．同様に，[6,10,14] 行目のカーネルにおける各変数名をデバイス側変数名に置き換える．最後に，各カーネル間でのデバイス側変数のアドレスなど必要な変数を関数間で受け渡す必要がある．[4-7,16,30][8-11,17,35][12-15,18,37] 行目のような関数，カーネル間で必要な引数を付与する． 一連の処理後のコードは**図 7** のようになる．

4.3 ストリーム割り当て

カーネル実行とデータ転送のオーバラップのため，ストリームの割り当てを行う．オーバラップのためには，実行中のカーネルが所属しているストリームとデータ転送命令を行う変数が所属しているストリームが別でなければならない．そのため，なるべく独立した多くのストリームを作る方が，データ転送の効率がよい．理想はカーネルの数と同じ数のストリームを作ることである．しかし，各ストリーム内では登録順に実行されていくが，それぞれ別ストリームに属しているカーネル，データ転送はどのような順で処理されていくかは不明である．依存関係のあるカーネルや変数は同一のストリームに属させないと意図しない結果になる場合がある．そのため，カーネル間やホストでの参照，書き込みなどの依存関係を解析し関連のあるカーネル，変数を一つのストリームにまとめる．別の依存関係のないカーネルや変数は別のストリームにまとめる．このようにしてなるべく多くのストリームを作成する．以下の手順でストリーム割り当てを行う．

(1)　各カーネルが使用している変数を解析する
(2)　カーネル間での変数依存関係を解析する
(3)　(2) の解析の結果を用いて依存関係のあるカーネル，変数を一つのグループとする
(4)　他のカーネルと依存関係のないカーネルは単一カーネルとそのカーネルが使用してい

```
4  __device__ void dev_sub1(int *_dev_picA,int *_dev_picB){//引数付与
5  int id=blockDim.x*blockIdx.x+threadIdx.x;
6    _dev_picA[id]=_dev_picA[id]-_dev_picB[i];//変数名の置き換え
7  }

16 __global__ void call_sub1(int *_dev_picA,int *_dev_picB){
                  dev_sub1(_dev_picA,_dev_picB);}//引数付与

20 int main(){
21   int *_host_picA,*_dev_picA;//変数宣言の置き換え
22   int *_host_picB,*_dev_picB;//変数宣言の置き換え

25-1   cudaMallocHost((void**)&_host_picA,N*sizeof(int));//メモリ確保命令の挿入
25-2   cudaMalloc((void**)&_dev_picA,N*sizeof(int));
25-3   cudaMallocHost((void**)&_host_picB,N*sizeof(int));
25-4   cudaMalloc((void**)&_dev_picB,N*sizeof(int));

30   call_sub1<<<N/32,32>>>(_dev_picA,_dev_picB);//引数付与
```

図 7 変数処理後のコード (抜粋)
Fig. 7 Code after variable replacement

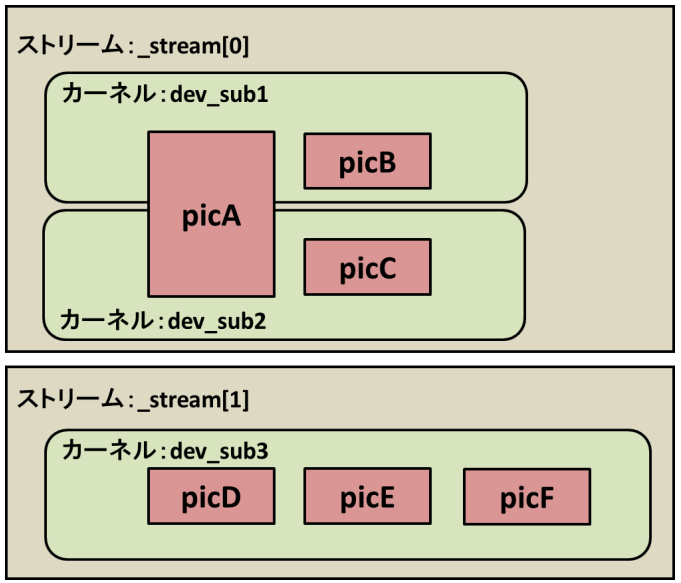

図 8 変数・カーネル・ストリームの関係
Fig. 8 Relations among variables, kernels and streams

る変数で一つのグループとする

(5) 各グループにストリーム名を割り振り，所属ストリームとして変数表に登録する

所属ストリームを解析した後，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．ストリーム名は配列で `_stream[N]` とする．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．

図 6 のコードを例に，変数に対してのストリーム割り当て処理を示す．変数，カーネル，ストリームの関係は**図 8** のようになる．変数 picA は [4-7] 行目のカーネル `dev_sub1`，[8-11] 行目のカーネル `dev_sub2` で使用されており，この `dev_sub1`，`dev_sub2` のカーネル間には依存関係がある．そのため，この 2 つのカーネルは同一のストリーム `_stream[0]` に所属する．また，カーネル `dev_sub1` で参照されている picB，カーネル `dev_sub2` に用いられている picC も同一のストリーム `_stream[0]` に所属する．`_stream[0]` 所属の変数の各変数表の所属ストリームの項目を `_stream[0]` として更新する．それとは別に，カーネル `dev_sub3` で参照されている変数 picD，picE，picF はすべてのカーネルにおいて，カーネル `dev_sub3` 以外では参照されていない．そのため，カーネル `dev_sub3`，picD，picE，picF を独立したストリーム `_stream[1]` としてまとめる．同様に，変数 picD，picE，picF の変数表の所属ストリームを `_stream[1]` として更新する．すべての変数，カーネルの依存関係を解析すると，必要なストリームの数がわかる．今回の例では，ストリーム数は 2 となるため，プログラム先頭においてストリームを `cudaStream_t` を用いて 2 つ宣言し，cudaStreamCreate 関数を用いて作成する．また，[30,35] 行目の `call_sub1`，`call_sub2` のカーネル呼び出しに `_stream[0]` 上で，[37] 行目の `call_sub3` のカーネル呼び出しに `_stream[1]` 上で呼び出すようにする．最後に，プログラム終了直前で cudaStreamDestroy 関数を用いて，この 2 つのストリームを破棄する．

4.4 データ転送自動化

各変数に対して，自動でデータ転送コードを生成することで自動転送を実現する．download 形式の変数に対しては download 転送のみを自動生成する．readback 形式の変数に対

しては readback 転送のみ自動生成する．share 形式の変数は，どちらの転送も自動生成を行う．各変数に対してデータ転送コードの挿入が必要である行を割り出す必要がある．以下にデータ転送コードが必要である行を示す．

- **download 転送条件**：
  転送条件 [i]:変数が使用されているカーネルの呼び出しからソースコートをさかのぼり探索し，ホストプログラムで変数の代入を行った直後の行
- **readback 転送条件**：
  転送条件 [ii]:変数が使用されているカーネル呼び出し後のコードで，ホスト側で参照がある場合，カーネル呼び出し直後の行

上記条件の箇所に挿入されるデータ転送命令を以下に示す．データ転送コードの各引数は転送する変数の変数表から参照される．

- **download 転送コード**：
  転送コード [a]: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**：
  転送コード [b]: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);

さらに，readback 転送においては，非同期式データ転送を用いているため，データ転送コードを発行してもホスト側での参照までにデータ転送が完了している保証がない．そのため，readback 転送コード挿入後，cudaStreamSynchronize 関数を用いて，同期処理を行う必要がある．同期コードの挿入条件と，挿入する同期コードを以下に示す．

- **readback 転送同期条件**：
  同期条件 [iii]:readback 転送命令挿入後の直近のホスト側での参照を行っている直前の行
- **readback 転送同期コード**：
  同期コード [c]:cudaStreamSynchronize(所属ストリーム);

各変数について上記の条件で変数表の転送形式に応じて適用する．download 形式の変数は，download 転送の転送条件 [i] で転送コード [a] を適用する．readback 形式の変数は，readback 転送の転送条件 [ii] で転送コード [b] と，同期条件 [iii]，同期コード [c] を適用する．share 形式の変数には，転送条件 [i] で転送コード [a]，転送条件 [ii] で転送コード [b]，同期条件 [iii]，同期コード [c] を適用する．

図 6 を例に説明する． 変数 picA は share 形式のため転送条件 [i]，[ii]，転送コード [a]，[b]，同期条件 [iii]，同期コード [c] が適用される．変数 picA は [27] 行目で転送条件 [i]，[31] 行目で転送条件 [ii]，[34] 行目で転送条件 [i]，[36] 行目で転送条件 [ii] が適合する．[27,34] 行目で転送コード [a]，[31,36] 行目で転送コード [b] に従いデータ転送コードを挿入する．また，[31,38] 行目で，同期条件 [iii] に適合する．そのため，[31,38] 行目に同期コード [c] を挿入する．[31] 行目には転送コード [b] がすでに挿入されているが，同期コードはこの転送コードの後ろに挿入する．変数 picA に対して挿入されたデータ転送コード，同期コードを**図 9** に示す．変数 picB は download 形式のため，転送条件 [i]，転送コード [a] が適用される．[27] 行目で転送条件 [i] に適合し，転送コード [a] を挿入する．同様に，変数 picC は download 形式で [27] 行目で転送条件 [i] に適合する．[27] 行目に，転送コード [a] を挿入する．download 形式の picE，picF は [29] 行目で転送条件 [i] に適合し，転送コード [a] を挿入する．readback 形式の picD は，転送条件 [ii]，転送コード [b]，同期条件 [iii]，同期コード [c] で，コード挿入を行うと [38] 行目で転送条件 [ii] に適合，転送コード [b] を挿入する．また，[40] 行目で同期条件 [iii] に適合し，同期コード [c] を挿入する．

```
27-1 //download 転送
27-2 cudaMemcpyAsync(_dev_picA, _host_picA, 320*sizeof(int), cudaMemcpyHostToDevice, _stream[0]);

31-1 //readback 転送
31-2 cudaMemcpyAsync(_host_picA, _dev_picA, 320*sizeof(int), cudaMemcpyDeviceToHost, _stream[0]);
31-3 //同期ポイント (_stream[0] のみの完了待ち)
31-4 cudaStreamSynchronize(_stream[0]);

34-1 //download 転送
34-2 cudaMemcpyAsync(_dev_picA, _host_picA, 320*sizeof(int), cudaMemcpyHostToDevice, _stream[0]);

36-1 //readback 転送
36-2 cudaMemcpyAsync(_host_picA, _dev_picA, 320*sizeof(int), cudaMemcpyDeviceToHost, _stream[0]);

38-1 //同期ポイント (_stream[0] のみの完了待ち)
38-2 cudaStreamSynchronize(_stream[0]);
```

図 9 変数 picA に対して挿入されたデータ転送命令コード (抜粋)
Fig. 9 Data transfer code inserted by proposed compiler

今回の例では，picA，picB，picC の download 転送コードが [27] 行目にそれぞれ挿入され，なおかつ同一のストリームに所属する．このような場合データ転送の順序のスケジューリングの余地がある．picA，picB は，カーネル `dev_sub1` の実行のためにデータ転送完了が必要である．picA，picC は，カーネル `dev_sub2` の実行のためにデータ転送完了が必要

表 2 実行時間 (sec)
Table 2 Execution time (sec)

デバイス	想定コード	最適コード	非最適コード
TeslaC2050	100.78	99.13	107.14
TeslaC1060	116.32	115.40	141.71

である．しかし，カーネル `dev_sub1` が先に実行されるため，picA，picB のデータ転送が優先されるべきである．本コンパイラはこのようなスケジューリング機能も実装する必要があるが，手法の設計は今後の課題である．

5. 予備実験

本コンパイラの有用性を示すために，本論文の手法に基づく出力コードを手動で作成し実行時間を計測した．手法を示していないデバイスメモリ量に応じた処理とデータ転送タイミングのスケジューリングは想定コードには盛り込んでいない．マッチングによる暗号解読のプログラムについて，想定出力コード，手動で最適化したコード，非最適化コードの 3 種類を用意し，2 種類の GPU デバイス上で実行時間の比較を行った．想定出力コードと最適コードとの差は，最適コードはデータ転送のスケジューリングを行っており，ストリーム番号 0 を用いた排他的転送による高速化を行っている点であり，想定出力コードと非最適化コードとの差は，カーネル実行とデータ転送のオーバラップの有無である．数字 5 桁の暗号を 1 億 7920 万個解読した場合の実行時間を**表 2** に示す．

結果から本コンパイラを用いることにより，手動で最適化したコードと 1 秒程度の差で，なおかつ，非最適化コードよりも高速で実行可能であることがわかる．従って，本コンパイラを用いると低負担で GPU プログラミングが可能で，かつ高速に実行できる．

6. おわりに

本論文では GPGPU におけるデータ転送自動化コンパイラの設計として，転送を自動化するための手法を示した．また，現状の手法における性能の見積もりを予備実験を通して行った．今後は，全データがデバイスメモリ上におさまらない場合のアルゴリズムの考案と転送タイミングのスケジューラを開発していき，提案したコンパイラの実装を進める．また，最適コードとの比較を行い実行性能について評価する．

参考文献

1) John D. Owens,David Luebke,Naga Govindaraju,Mark Harris,Jens Krger,Aaron E. Lefohn,Timothy J. PurcellA Survey of General-Purpose Computation on Graphics Hardware
2) GPGPU.org,http://gpgpu.org/
3) nVidia CUDA Zone,
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
6) HiroyukiSato,ShigekiTakase,Atsuo,Ozaki,ToshioWakayama,SPEED-UP OF SAR IMAGE FORMATION PROCESSING USING GRAPHICS PROCESSING UNITS
7) 中川 進太, 伊野 文彦, 萩原 兼一,CUDA プログラムにおいてストリーム処理を支援するミドルウェア
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
9) YiYang,PingXiang,JingfeiKong,HuiyangZhou,A GPGPU Compiler for Memory Optimization and Parallelism Management