一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
CPSY2015-23(2015-08)

データ転送の発行順序制御によるGPUプログラムの高速化

小野　和馬† 　竹嶌　良† 　津邑　公暁†

† 名古屋工業大学

あらまし　理論演算性能の高い GPU に汎用計算を行わせる GPGPU が注目を集めている．このような GPU プログラミング用に，NVIDIA 社は並列計算アーキテクチャモデルとして CUDA を提供している．CUDA では，データ並列な処理を並行実行させるハードウェアサポートに加え，処理順依存のない，異なる関数を並行実行させるサポートも備えている．しかし，関数の効率的な並行実行のためには，プログラマは関数の実行順を適切に制御しなくてはならない．この問題を緩和する手法の１つとして，Kernel Reordering が提案されている．しかしこの手法では制御対象にデータ転送を含めておらず，関数のみの実行順序を制御するため，本来並行実行可能なデータ転送と関数とが逐次実行される可能性がある．そこで本稿では，Kernel Reordering を改良し，関数だけではなくそれに付随するデータ転送も含めて実行順序を制御する手法を提案する．

キーワード　GPGPU, CUDA, Concurrent Kernel Execution

An Effective Scheduling of Data Transfer for GPU Applications

Kazuma ONO†, Ryo TAKESHIMA†, and Tomoaki TSUMURA†

† Nagoya Institute of Technology

Abstract　Utilizing a GPU to perform general purpose computation is called GPGPU. The high theoretical performance of GPU draws attention to GPGPU. For the GPU programming, CUDA supplies an architecture model for parallel processing. A CUDA-enabled GPU has hardware supports for executing different functions concurrently. For using the hardware supports efficiently, developers should troublesomely schedule the execution order of functions appropriately. Kernel Reordering is one of the studies for supporting the function-scheduling. However, Kernel Reordering does not handle data transfer, and data transfer and function execution are serialized even if they have no dependency each other. To address this problem, we improve Kernel Reordering to schedule not only functions but also data transfer.

Key words　GPGPU, CUDA, Concurrent Kernel Execution

1. は じ め に

GPU（**Graphics Processing Units**）は画像処理に特化したプロセッサであり，広域なメモリバンド幅や，高いベクトル演算性能を持つ．また，CPU と比べコア数が多く，並列処理に適したプロセッサである．このような特徴を持つ GPU に汎用計算を行わせる **GPGPU**（**General Purpose computing on Graphics Processing Units**）が注目を集めている．このような GPU プログラミングのために，NVIDIA 社製の GPU を対象とした並列計算アーキテクチャモデルとして，**CUDA**（**Compute Unified Device Architecture**）[1] が開発されている．CUDA では**カーネル関数**として定義した処理を GPU が持つ大量のコアを用いて並行実行させることで，高速な処理を実現する．ただし，主にデータ並列性を持つ処理が GPU による高速化の恩恵を受けることができ，複数タスクの並列実行は困難であった．これは，Fermi [2] より前の構造を採用した GPU では，ハードウェアの制約上，異なるカーネル関数を並行実行できなかったためである．そこで，GPU 上でタスク並列な処理を実行するためのハードウェアサポートが登場した．このサポートの１つに，Fermi 以降のアーキテクチャを採用した GPU に搭載されている **Concurrent Kernel Execution** がある．ただし，異なるカーネル関数同士を並行実行させることによって性能向上を図るためには，プログラマがカーネル関数の実行順序を適切にスケジューリングする必要がある．しかし，特に関数の実行順序が不定となるマルチスレッドアプリケーションにおいて，プログラマが適切にスケジューリングを行うことは難しい．この問題に対し，カーネル関数の実行順序を制御する手法である，Kernel Reordering が提案されている．しかし，この手法はデータ転送をスケジューリング対象としていないため，データ転送とカーネル関数が逐次実行されることで，実行速度が低下してしまう可能性がある．そこで本稿ではデータ転送もスケジューリング対象とするよう Kernel Reordering

This article is a technical report without peer review, and its polished and/or extended version may be published elsewhere.
Copyright ©2015　by　IEICE

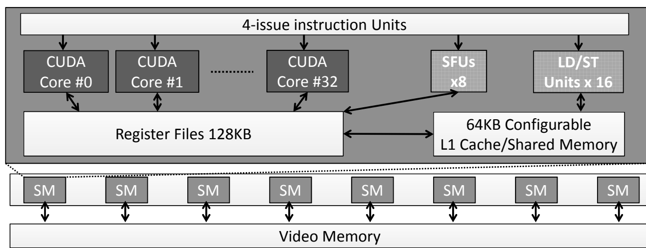

図 1 GPU のアーキテクチャ概略図

を改良し，データ転送とカーネル関数を並行実行させることでGPU プログラムの高速化を目指す.

2. 研 究 背 景

本章では CUDA および，GPU 上における処理の実行順序を制御する既存研究について述べる.

2.1 CUDA

CUDA は，NVIDIA 社製の GPU を対象とした並列計算アーキテクチャモデルである．CUDA では GPU 上で実行する処理をカーネル関数として定義し，このカーネル関数を多数のスレッドを用いて並行実行する．ただし，プログラマはカーネル関数を呼び出す際に，GPU 上にどのようにスレッド群を生成し用いるかを示す実行構成を定義する必要がある.

ここで，NVIDIA 社製の一般的な GPU アーキテクチャの例を**図 1** に示す．GPU は階層的な構造を持ち，内部には **SM**（**Streaming Multiprocessor**）と呼ばれる複数のプロセッサが存在する．さらに SM 内には **CUDA Core** と呼ばれる演算器やロード・ストアユニットなどのユニットが多数存在する．CUDA では，CUDA Core 上の多数のスレッドに同一の処理を割り当て並行実行する，**SIMT**（**Single Instruction, Multiple Thread**）型の実行方式を採用している．この多数のスレッドを用いた並行実行により，CUDA は高い演算性能を実現できる.

なお，このように GPU 上で多数のスレッドを用いてデータを処理するためには，そのデータを CPU のホスト・メモリから，GPU のデバイス・メモリへと転送する必要がある．ここで，データ転送とカーネル関数実行との間に処理順依存が無い場合，これらは並行実行可能である．この並行実行は，CUDA が提供するストリームを用いることで実現可能となる．ストリームとは，GPU での処理の実行順序をプログラマが定義できる枠組みである．プログラマはデータ転送やカーネル関数の呼出し時に特定のストリームを指定することで，処理の発行順序を制御することができる．ここで，同一のストリームに発行された処理同士は，必ず発行された順に逐次的に実行される．つまり，同一のストリームに発行された処理は，CUDA によって処理順依存があるとみなされる．これに対し，異なるストリームに発行された処理は，CUDA によって処理順依存が無いとみなされ，並行実行される．ただし，プログラマがストリームを指定しなかった場合，全ての処理が単一のストリームへと発行され逐次実行されてしまう．このことから，カーネル関数とデータ転送とを並行実行させるためには，それぞれの発行先ストリームをプログラマが明示的に指定する必要がある．そして，ストリームへと発行された処理は GPU 内のハードウェア・ワークキューへと集約され，そこから各 SM へと発行される.

さらに，Fermi 以降のアーキテクチャを搭載した GPU では，

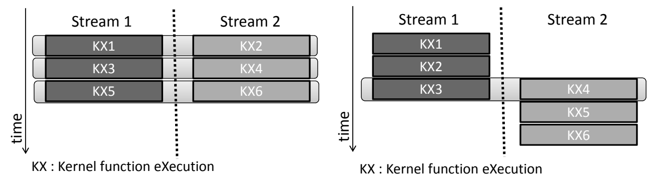

図 2 関数が並行実行される様子（偽の依存関係なし）

図 3 関数が並行実行されない様子（偽の依存関係あり）

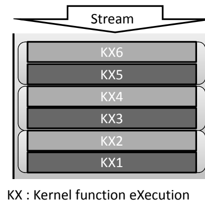

図 4 ワークキューへの処理発行（偽の依存関係なし）

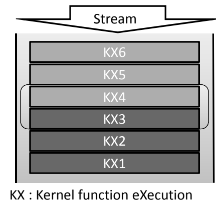

図 5 ワークキューへの処理発行（偽の依存関係あり）

あるカーネル関数の実行中に別の新たなカーネル関数の実行を可能とする **Concurrent Kernel Execution** と呼ばれるハードウェアサポートを搭載しており，カーネル関数同士を並行実行することができる．しかし，異なるストリームに対してカーネル関数呼出しを発行したとしても，カーネル関数同士が並行実行されないことがある．これはカーネル関数呼出しの発行順序によっては，異なるストリームに対して発行されたカーネル関数呼出し同士が依存関係を持っているかのように CUDA に誤認されてしまうためであり，このような依存関係を本稿では偽の依存関係と呼ぶ.

ここで，カーネル関数呼出しが適切にストリームへと発行され関数が並行実行される様子を**図 2** に，カーネル関数呼出しをストリームへと発行する際に偽の依存関係が発生してしまい関数が十分に並行実行されない様子を**図 3** にそれぞれ示す．なお，図中の KX はカーネル関数の実行（Kernel function eXecution）を，KX の添字はストリームへと発行された順番をそれぞれ表す．図 2 に示すように異なるストリームへカーネル関数呼出しが交互に発行される場合，KX1 と KX2，KX3 と KX4，KX5 と KX6 がそれぞれ並行実行される．これに対して，図 3 に示すように同じストリームへカーネル関数呼出しが続けて発行される場合，連続して発行されたカーネル関数のうち，最後の KX3 のみが別のストリームのカーネル関数と並行実行される．これはストリームに発行された処理が 1 つのハードウェア・ワークキューに集約された後，処理が SM に発行されることに起因する．ここで，図 2 および図 3 で示した順番で発行された処理が集約された際のハードウェア・ワークキュー内の様子を，**図 4** および**図 5** にそれぞれ示す．なお，図中の枠で囲まれた処理はそれらが連続して発行され，並行実行されることを表す．図 2 に示すように，異なるストリームへ処理が交互に発行された場合，図 4 に示すようにワークキュー内にも異なるストリームの処理が交互に登録される．そのため，KX1 と KX2，KX3 と KX4，KX5 と KX6 の 3 組がそれぞれ並行実行される．一方で図 3 に示すように，同じストリームへ処理が連続して発行された場合，図 5 に示すようにワークキュー内にも同じストリームの処理が連続して登録される．そして，同じストリームの処理は逐次的に実行されるため，図 5 に示すように KX3 と KX4 の 1 組のみが並行実行される．このように

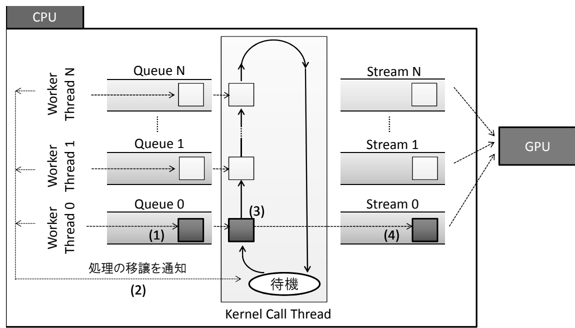

図 6 Kernel Reordering の動作モデル

偽の依存関係が発生する場合，関数が十分に並行実行されなくなり，プログラム本来の性能が引き出せない.

2.2 Kernel Reordering

2.1 節で述べたように，ストリームに対する処理の発行順序によっては偽の依存関係が発生し，プログラムの性能に影響を与える．そのため，プログラマは偽の依存関係が発生しないように十分に注意を払う必要がある．しかしカーネル関数呼出しを，常に偽の依存関係が発生しないようにストリームへと発行することは難しい．特に，実行時に処理の発行順序が定まらないマルチスレッドアプリケーションの場合，適切にスケジューリングすることは非常に困難である．そこで，プログラマが適切な発行順序を考慮せずとも，同じストリームへカーネル関数呼出しが連続して発行されることを防ぐ手法として，**Kernel Reordering** [3] が提案されている．Kernel Reordering では，アプリケーション自体の処理を行う複数のスレッドである **Worker Thread** のほかに，各 Worker Thread に対応した複数の **Queue** と，カーネル関数呼出しの発行を担当する専用の **Kernel Call Thread** を CPU 上に用意する．なお，各 Worker Thread および Queue は，識別用の ID を持つ．ここで，**図 6** に Kernel Reordering の動作モデルを示す．まず，Kernel Call Thread は Worker Thread から通知を受けるまで待機状態となる．それに対して各 Worker Thread は，アプリケーションを実行していき，カーネル関数呼出しの存在を検知すると，自身の Queue にこのカーネル関数呼出しを登録し (1)，そのことを Kernel Call Thread へ通知する (2)．通知を受けた Kernel Call Thread が Queue からカーネル関数呼出しを取り出すことで，Worker Thread の代わりに カーネル関数呼出しをストリームへと発行する．すなわち，Worker Thread による上記の一連の動作によって，Kernel Call Thread にカーネル関数呼出しの発行権限を移譲する．そして関数呼出しの発行権限を移譲した後，Worker Thread はカーネル関数の実行結果が必要になるまで後続処理を継続し，その後 Kernel Call Thread からのカーネル関数の実行完了の通知を待つ．一方で Kernel Call Thread は，カーネル関数呼出しの発行権限の移譲を受け，自身の待機状態を解除し，Queue からカーネル関数呼出しを取り出して (3)，ストリームに発行する (4)．その後 Kernel Call Thread は，Queue からカーネル関数呼出しを取り出したことを，その Queue に対応する Worker Thread に通知する．ここで，図 6 の (3) において，Kernel Call Thread はスレッド ID を用いてラウンドロビン方式で Queue を選択し，カーネル関数呼出しを取り出す．これにより，同じストリームへ連続してカーネル関数呼出しが発行されることを防ぎ，偽の依存関係の発生を抑制できる.

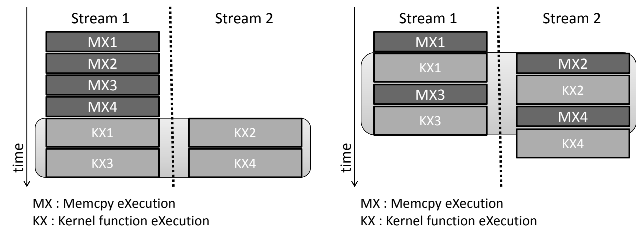

図 7 Kernel Reordering を用いても並行実行されない状態

図 8 理想的に並行実行される状態

3. データ転送処理の発行順序制御

本章では，既存手法である Kernel Reordering の問題点および，それを解決する提案手法について述べる.

3.1 Kernel Reordering の問題点

2.2 節で述べた Kernel Reordering はカーネル関数呼出しのみをスケジューリングの対象としており，カーネル関数は全てのデータ転送の完了後に初めて実行される．そのため，本来ならば並行実行可能なデータ転送とカーネル関数とが逐次実行されてしまう可能性がある．ここで，**図 7** に Kernel Reordering を用いた際の処理の実行状態を，**図 8** にデータ転送とカーネル関数が並行実行される理想的な状態を，それぞれ示す．なお，図中の MX はデータ転送の実行（Memcpy eXecution）を表している．また，MX や KX の添字は MX と KX の対応関係を表しており，KX で必要なデータが同じ添字を持つ MX で転送されることを示す．図 7 に示すように，Kernel Reordering を用いたとしてもデータ転送とカーネル関数は逐次実行されてしまう．これは，Kernel Reordering がカーネル関数呼出しのみをスケジューリングの対象としているためである．しかしながら，データ転送とカーネル関数は並行実行可能であるため，図 8 に示すように実行されるのが理想である．この問題点を解消するため，本稿ではカーネル関数とデータ転送が並行実行されるよう Kernel Reordering を拡張した，**事前転送方式**と**直前転送方式**の 2 つの手法を提案する．これらは，カーネル関数呼出しが Queue に登録されてから Kernel Call Thread によってストリームへと発行されるまでのストリームの空き時間の長さに応じて，それぞれ適する状況が異なる．なお，この時間の長さは直前にストリームで実行される処理や OS の割り込み処理によって変化する.

3.2 カーネル関数呼出しとデータ転送処理の発行順序制御

本節では，提案する事前転送方式と直前転送方式の動作モデルについて述べる.

3.2.1 事前転送方式

事前転送方式は，データ転送を可能な限り早期に実行することで，OS の割り込み処理が発生したとしてもストリームの空き時間を減らすことを目的とする方式である．この方式では，カーネル関数呼出しを Queue に登録する直前に，Worker Thread がデータ転送処理をストリームに発行する.

ここで，事前転送方式を用いる場合の動作モデルを**図 9** に示す．この図は，2 つの Worker Thread と 1 つの Kernel Call Thread が，Queue を介して 2 つのストリーム Stream1, Stream2 にデータ転送処理とカーネル関数呼出しを発行している様子を表している．なお，図中の KI は Worker Thread がカーネル関数呼出しを Queue に登録し Kernel Call Thread に

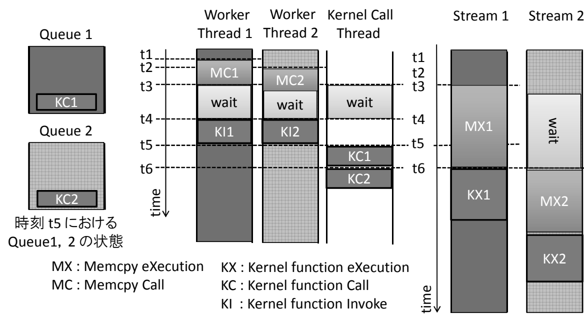

図 9　事前転送方式の動作モデル

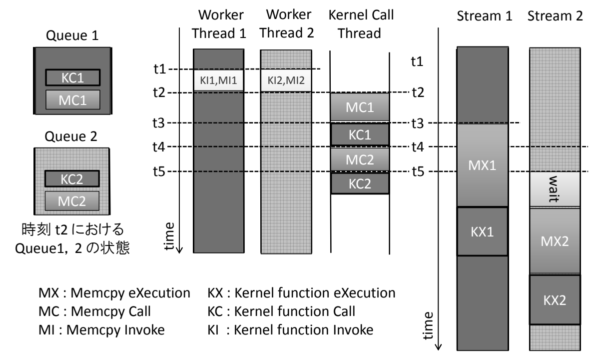

図 10　直前転送方式の動作モデル

発行権限を移譲する（Kernel function Invoke）ことを，KC はカーネル関数呼出し（Kernel function Call）を，MC はデータ転送の呼出し処理（Memcpy Call）をそれぞれ表している．この方式では，データ転送とカーネル関数を並行実行させつつストリームの空き時間を減らすために，Worker Thread がデータ転送処理をなるべく早くストリームへと発行する．そのため，Kernel Reordering のようにカーネル関数呼出しのスケジューリング前にデータを一括して転送するのではなく，Worker Thread がカーネル関数呼出しを Queue に登録する段階で，対応するデータ転送処理をストリームへと発行する．このような動作を実現するため，まず Worker Thread1 は Stream1 に対して MC1 を発行し（t1），Worker Thread2 は Stream2 に対して MC2 を発行する（t2）．その後に，OS の割り込み処理のため Worker Thread と Kernel Call Thread が動作を中断したとしても（t3〜t4），最も早く発行される MX1 が，Stream1 で実行開始できる（t3）．なお，直後に Stream2 へ MX2 も発行されるが，GPU はデータバスを 1 つしか持たないため，MX2 は MX1 の完了を待ち，発行される（t6）．そして，OS の割り込み処理が完了した後，Worker Thread1 は KC1 に対応する KI1 を行い，Worker Thread2 は KC2 に対応する KI2 を行う（t4）．このようにデータ転送処理を発行する Worker Thread に対し，Kernel Call Thread はカーネル関数呼出しをストリームへと発行する．そのため，Kernel Call Thread は，Worker Thread1，Worker Thread2 から KI1，KI2 を行ったという通知を受け取ったあと，Queue からカーネル関数呼出しを取り出し，各 Stream に対して KC1，KC2 を発行する（t5, t6）．ここで Kernel Call Thread は，2.2 節で述べた Kernel Reordering と同様に，カーネル関数呼出しを取り出す Queue をラウンドロビン方式で選択する．そして，先にストリームに発行された MX1 が完了してから（t6），Stream1 において KX1 が，Stream2 において MX2 がそれぞれ実行開始され，並行実行される．このように事前転送方式は，後述する直前転送方式よりもストリームの空き時間を減らすことができる．

3.2.2　直前転送方式

直前に何らかの処理を発行したストリームと異なるストリームへ処理を発行した場合には，ストリームの切替えが生じ，多少のオーバヘッドが伴う．そのオーバヘッドは，データ転送にかかる時間が短い場合，無視できる大きさではない．直前転送方式は，カーネル関数の実行直前にデータを転送することで，データ転送と関数の実行を連続させ，ストリームの切替えを減らすことを目的とする方式である．この方式は，事前転送方式と異なり Worker Thread がデータ転送処理も Queue に登録しておくことで，Kernel Call Thread にデータ転送処理の発行も任せる．そして，Kernel Call Thread がデータ転送処理と関数呼出しを連続してストリームへ発行することで，それらの実行を必ず連続させることが可能となる．

ここで図 **10** に，図 9 と同じ形式で，直前転送方式の動作モデルを示す．なお図中の MI は，Worker Thread がデータ転送処理を Queue に登録し Kernel Call Thread に発行権限を移譲する（Memcpy Invoke）ことを示す．この直前転送方式では事前転送方式とは異なり，Worker Thread はデータ転送処理を発行せず，データ転送処理とカーネル関数呼出しの発行権限を Kernel Call Thread に移譲する．そのため，Worker Thread1 および Worker Thread2 は，それぞれ MC1，KC1 と MC2，KC2 に対応する KI1，MI1 と KI2，MI2 を行う（t1）．一方，Kernel Call Thread はラウンドロビン方式で選択した Queue からデータ転送処理とカーネル関数呼出しを連続して取り出し，ストリームに発行する．そのため，Stream1 に対して MC1 と KC1 を続けて発行し（t2，t3），Stream2 に対して MC2 と KC2 を続けて発行する（t4，t5）．このように直前転送方式では，同じストリームへのデータ転送処理とカーネル関数呼出しの発行を連続させることによって，ストリーム切替えによるオーバヘッドを抑制できる．

4.　提案手法の実装

本章では，提案手法の実装について述べる．

4.1　Queue に登録する要素クラスの設計

既存手法である Kernel Reordering では，カーネル関数呼出しを Queue に登録する際に，登録する関数に与える引数も併せて登録する必要があるため，Queue に登録される要素が関数ポインタと引数とを保持できるよう実装されている．しかし 3.2 節で示したように，提案手法ではカーネル関数呼出しだけではなく，データ転送処理も Queue に登録する．そのため，データ転送に必要なホスト・メモリおよびデバイス・メモリ上のアドレスや，転送するデータのサイズも Queue の要素が保持できる必要がある．そこで，データ転送処理が登録できるよう Kernel Reordering の Queue を拡張する．具体的には，スレッド ID や実行構成などカーネル関数呼び出しに共通する情報を管理するための抽象クラスを設計し，これを基にして，データ転送処理を扱うためのメソッドやメンバ変数を持つ派生クラスを設計した．そして，この派生クラスを Queue の要素とすることで，データ転送処理とカーネル関数呼出しを共に扱える Queue を実装した．この派生クラスのインスタンスでは，データ転送処理に必要なアドレスやデータサイズ，関数の引数，関数ポインタを保持する．さらに，関数の実行構成とスレッド ID の Queue

```
1  ...
2  for(i=0; i<x; i++)
3    cudaMemcpy(h_mem[i], ...);
4  #pragma parallel omp thr(8){
5    for(i=0; i<x; i++){
6      queue[i].enqueue(kernel, id);
7      //発行権限を移譲
8      //処理結果が帰ってくるまで待機
9  } }
10 ...
11
12 void KernelCallThr(){
13   while(true) {
14     //通知を受け取るまで待機
15     callKernel(queue[i]);
16     ...
17
18
19 } }
```

図 11 Kernel Reordering を用いる場合の実装例

```
1  ...
2  #pragma parallel omp thr(8){
3    for(i; i<x; i++){
4      if( 事前転送方式 )
5        cudaMemcpy(h_mem[i], ...);
6      queue_ex[i].enqueue;
7      //発行権限を移譲
8      //処理結果が帰ってくるまで待機
9  } }
10 ...
11
12 void KernelCallThr(){
13   while(true) {
14     //通知を受け取るまで待機
15     if( 直前転送方式 )
16       cudaMemcpy(h_mem[i], ...);
17     queue_ex[i].callKernel;
18     ...
19 } }
```

図 12 提案手法を用いる場合の実装例

への登録，関数呼び出しとデータ転送処理の Queue への登録，カーネル関数呼出しとデータ転送処理のストリームへの発行を行うメソッドをそれぞれ用意する．

ここで，3.2.1 項および 3.2.2 項で述べたそれぞれの提案手法で，Worker Thread および Kernel Call Thread が派生クラスのメソッドをどう用いるのかについて述べる．事前転送方式では 3.2.1 項で示したとおり，データ転送を可能な限り早期に開始する．そのため，関数呼び出しを Queue に登録するメソッドの呼出し直前に，データ転送処理を呼び出すことで，事前転送方式を実装できる．直前転送方式では 3.2.2 項で示したとおり，データ転送をカーネル関数の実行直前に開始する．そのため，カーネル関数呼出しをストリームへ発行するメソッドの呼出し直前に，データ転送処理を呼び出すことで，直前転送方式を実装できる．

4.2 アプリケーションへの適用

本節では，4.1 節で述べた Queue の要素クラスを用いて，実際にアプリケーションに提案手法を適用する際の，Kernel Reordering からの変更点について述べる．ここで，**図 11** に Kernel Reordering を用いてスケジューリングする場合，**図 12** に提案手法を用いてスケジューリングする場合の，Worker Thread と Kernel Call Thread の記述例を示す．なお，両図の 1〜10 行目はデータ転送と Worker Thread の動作に対応するコードの一部を示しており，12 行目以下は Kernel Call Thread の動作に対応するコードの一部を示している．Kernel Reordering では 2.2 節で示したように，関数の呼出し前に全てのデータ転送を完了させる（図 11，2〜3 行目）．その後，Worker Thread を生成し（4 行目），各スレッドはカーネル関数呼出しを Queue に登録する（6 行目）．そして，Kernel Call Thread が Queue に登録されたカーネル関数呼出しを取り出し，ストリームへと発行する（15 行目）．これに対し，提案手法ではデータ転送も含めてスケジューリングするため，先に Worker Thread を生成する（図 12，2 行目）．そして事前転送方式を用いる場合は，可能な限りデータ転送を早期に行うために，Worker Thread がデータを転送する（5 行目）．一方直前転送方式を用いる場合は，データ転送とカーネル関数の実行を連続させるために，Kernel Call Thread がデータを転送する（16 行目）．

表 1 評価環境

OS	CentOS 6.4
CPU	Intel Core i7-4770
GPU	GTX TITAN
Generation	Kepler2
CUDA version	6.5.0

表 2 ベンチマークのパラメータ

Benchmark	sqrt	cutcp	himeno
Worker Thread 数	16	8	8
関数呼出しの数	128	26	128
問題サイズ	4MB	Large	Middle

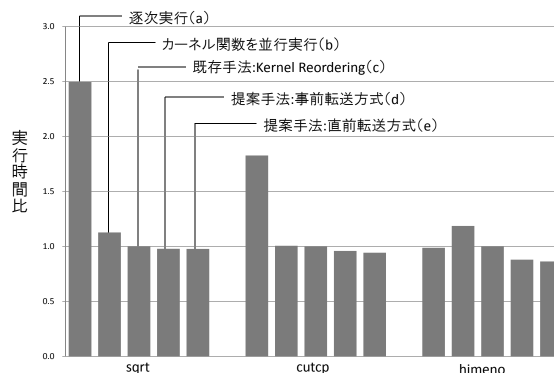

図 13 評価結果（Kepler2）

5. 評　　価

提案手法の有効性を確かめるために，Kernel Reordering と提案手法を用いてスケジューリングしたプログラムの実行速度を比較し，各手法の評価結果について考察した．

5.1 評価環境

評価環境を**表 1** に示す．評価に用いる GPU には，NVIDIA 社の Geforce シリーズから，Kepler2 アーキテクチャを採用した GTX TITAN を使用した．また評価には，既存手法である Kernel Reordering が評価に用いている平方根計算を行わせるサンプルプログラム（sqrt）と，Parboil ベンチマークから cutcp，加えて姫野ベンチマーク（himeno）の 3 つのプログラムを用いた．各プログラムの実行時のパラメータを**表 2** に示す．なお，ベンチマークパラメータのうち関数呼出しの数とは，プログラム内でカーネル関数を呼び出す回数を示す．

5.2 評価結果

評価結果を**図 13** に示す．縦軸は実行時間を表しており，既存手法である Kernel Reordering を 1 として正規化している．図中のグラフは，それぞれのベンチマークプログラムの実行時間を 5 本のバーで表しており，左から順に（a）カーネル関数を逐次実行したプログラム,（b）OpenMP を用いてカーネル関数を並行実行するようにスケジューリングしたプログラム,（c）既存手法である Kernel Reordering を用いてスケジューリングしたプログラム,（d）提案手法のうち事前転送方式でスケジューリングしたプログラム,（e）直前転送方式でスケジューリングしたプログラムとなっている．このグラフから，事前転送方式と直前転送方式のいずれの方式でも，既存手法と比べて性能が向上していることがわかる．データ転送とカーネル関数が並行実行されることで，既存手法と比較して事前転送方式では平均約 6.03% ，最大約 12.0% ，直前転送方式では平均 7.18% ，最

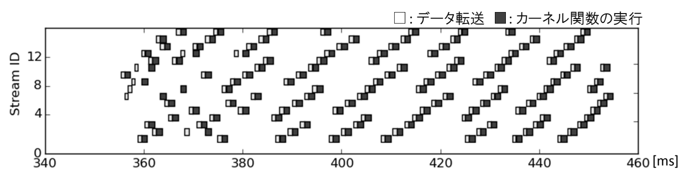

図 14 平方根プログラムのプロファイル結果（事前転送方式）

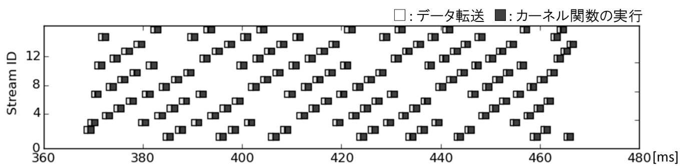

図 15 平方根プログラムのプロファイル結果（直前転送方式）

大約 13.6% それぞれ実行時間が削減できた．

5.3 考　　察

提案手法において，関数とデータ転送の並行実行により実行時間が削減されたが，直前転送方式のほうがより実行時間を削減できている．この原因を調べるために，sqrt 実行時の様子をプロファイラを用いて解析した．提案手法である事前転送方式，および直前転送方式のそれぞれで実行した際のプロファイル結果を**図 14** および**図 15** に示す．図中の縦軸は各 StreamID を，横軸は実行時間を示しており，実行時間の単位はミリ秒である．また凡例は白色がデータの転送を，黒色がカーネル関数の実行をそれぞれ示している．図 14 では，プログラムの開始直後において，データ転送完了からカーネル関数実行開始までにタイムラグが発生していることが分かる．これは事前転送方式が，カーネル関数呼出しを取り出す Queue をラウンドロビン方式で選択しており，転送が完了したデータから優先して処理していないためである．そのため，関数の実行開始まで不必要な待機が発生している．それに対し，図 15 では全てのデータ転送とカーネル関数の実行が連続しており，関数が実行開始まで不必要な待機をしていない．そのため，直前転送方式を用いることで事前転送方式よりも実行時間が削減できたと考えられる．

6. 関 連 研 究

2.1 節で述べた偽の依存関係を解消するために，NVIDIA 社は Hyper-Q [4] と呼ばれるハードウェアサポートを Kepler2 以降の GPU アーキテクチャに提供している．Hyper-Q の登場により，Kepler [5] までのアーキテクチャには 1 つしか搭載されていなかったハードウェア・ワークキューを複数搭載するようになり．複数のストリームに発行された処理を同時にワークキューへと発行できるようになった．しかし Hyper-Q は，データ転送の発行順序をスケジューリングしないため，データ転送とカーネル関数の並行実行が十分に行われない．これに対し本提案手法は，データ転送とカーネル関数が並行実行されるようスケジューリングするため，Hyper-Q に対して優位性を持つと考えられる．

また，複数の GPU を用いて，それぞれの GPU に異なる処理を行わせることで，複数タスクの並列実行を高速化させる研究も行われている．この研究の 1 つとして，マルチ GPU 環境向けのプログラミングフレームワークである GPU-Chariot [6] が提案されている．このフレームワークは，マルチ GPU 環境でデータ転送とカーネル関数が並列実行される期間を増大させるものである．また，この手法では，処理の実行時間がその処理対象であるデータのサイズに強く依存すると仮定し，実行時にデータサイズを考慮して処理の実行順序を決めるアウトオブオーダ方式のスケジューリングを採用している．しかし，この手法では 1 つの GPU で 1 つのカーネル関数のみを実行するため，小規模なカーネル関数を実行する場合には GPU リソースを十分に活用できない可能性がある．一方，本提案手法では関数どうしを並行実行するため，小規模な関数を実行する場合も GPU リソースをより有効活用できる．

7. お わ り に

本稿では，GPU アプリケーションにおけるデータ転送処理とカーネル関数呼出しの発行順序を制御する，事前転送方式と直前転送方式の 2 種類の方式を提案した．いくつかのベンチマークプログラムを用いて提案手法を評価したところ，既存手法である Kernel Reordering と比較して，事前転送方式では平均約 6.03% ，最大約 12.0% ，直前転送方式では平均 7.18% ，最大約 13.6% それぞれ実行時間が削減できた．

今後の課題としては，まず Queue からカーネル関数呼出しを取り出す順序を改良することが挙げられる．現在の実装では，既存の Kernel Reordering と同様に，カーネル関数呼出しを取り出す Queue をラウンドロビン方式で選択している．しかし，直前転送方式においては，データ転送とカーネル関数の実行が連続するため，先に Queue から取り出される処理によって転送されるデータのサイズが大きい場合，転送が終わるまで，他のデータを用いるカーネル関数の実行開始が遅れてしまう可能性がある．そこで，Queue の ID 順ではなく，データのサイズ順に転送を開始する Queue を選択することで，この問題を緩和できると考えられる．さらに，データ転送処理の発行タイミングが異なる 2 つの方式を動的に切り替えることが課題として挙げられる．OS による割込み処理が多く発生する場合，ストリームの空き時間が多く発生するため，事前転送方式が適しており，割込み処理が発生しない場合，ストリームの空き時間はあまり発生しないため，直前転送方式が適している．よって，割込み処理の発生状況に応じて 2 つの方式を動的に切り替えることで，更なる性能向上が見込めると考えられる．

文　　献

[1] CUDA C Programming Guide, NVIDIA Corp. 7.0 edition, Mar. 2015.
[2] Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture: Fermi, NVIDIA Corp. 1.1 edition, 2009.
[3] F. Wende, F. Cordes, and T. Steinke, "On Improving the Performance of Multi-threaded CUDA Applications with Concurrent Kernel Execution by Kernel Reordering," Proc. Application Accelerators in Higph Performance Computing (SAAHPC 2012) , pp.75–83, IEEE, 2012.
[4] T. Bradley, Hyper-Q Example, NVIDIA Corp. 1.0 edition, Aug. 2012.
[5] Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110, NVIDIA Corp. 1.0 edition, 2012.
[6] F. Ino, S. Nakagawa, and K. Hagihara, "GPU-Chariot: A Programming Framework for Stream Applications Running on Multi-GPU Systems," IEICE Transactions on Information and Systems, vol.E96-D, no.12, pp.2604–2616, Dec. 2013.