Vol.2009-HPC-121 No.23
2009/8/5

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

中川進太[†1]　伊野文彦[†1]　萩原兼一[†1]

本稿では，GPU（Graphics Processing Unit）におけるストリーム処理を支援するためのミドルウェアについて述べる．我々のミドルウェアは，CUDA（Compute Unified Device Architecture）プログラムを対象として，GPUにおける計算をCPU・GPU間におけるデータ転送とオーバラップする際の作業負担を軽減する．オーバラップの効果を最大化するために，ミドルウェアはカーネル実行やデータ転送などのAPI関数を実行時に適切な順序で呼び出す．開発者はCUDAが提供する関数をミドルウェアのものに置き換えることでオーバラップを実現できる．評価実験では，実際のアプリケーションにミドルウェアを適用し，実行時間を14%ほど削減できた．また，テクスチャを使うアプリケーションについては，オーバラップによる隠蔽効果がないことを確認した．

A Middleware for Assisting Stream Processing in CUDA Programs

Shinta Nakagawa,[†1] Fumihiko Ino[†1]
and Kenichi Hagihara[†1]

This paper presents a middleware for achieving stream processing on the graphics processing unit (GPU). Our middleware works for compute unified device architecture (CUDA) programs. It is designed to reduce development efforts needed for overlapping GPU computation with data transfer between the CPU and the GPU. To maximize the efficiency of this overlap, the middleware dynamically determines the appropriate order of API executions, such as the kernel invocation and the data transfer functions. Using our middleware, developers can achieve overlap by replacing CUDA functions with our functions. In experiments, we apply the middleware to practical applications. As a result, it reduces the execution time by 14%. We also find that there is no overlapping effects for texture-based applications.

1. はじめに

GPU（Graphics Processing Unit）はグラフィクス処理用の演算装置であり，CPUと比較して高い浮動小数点演算性能を有する．そこで，GPUの性能を汎用計算に活用するGPGPU[1)]（General Purpose computation on GPU）が注目されている．現在，nVIDIA社はC言語に対する拡張であるCUDA[2)]（Compute Unified Device Architecture）を提供しており，これを用いてGPU上で動作するプログラムを記述できる．

CUDAでは，計算に必要な入力データをGPU側のデバイスメモリに転送した上で，GPU上で計算を実行する．計算終了後，結果を主記憶に転送する．CUDAプログラムでは，主記憶からデバイスメモリへのデータ転送，GPU上での計算およびデバイスメモリから主記憶へのデータ転送を繰り返し逐次実行する．逐次実行する一連の命令をストリームと呼び，CUDAプログラムでは複数のストリームを並行に実行できる．このとき，あるストリームの計算を，異なるストリームのデータ転送とオーバラップできる．しかし，ストリーム処理には以下に述べる制約が存在する．まず，データ転送は異なるストリーム間であっても命令の発行順に逐次実行される．同様に，計算命令も逐次実行される．したがって，命令の発行順序により，計算をデータ転送とオーバラップできない例が存在する．

CUDAプログラムでのオーバラップにおける問題点の一つとして，開発者の労力が大きいことが挙げられる．これは，以下の2点が原因である．第一に，ストリーム処理における制約を守ってオーバラップを実現できる順序で命令を発行する必要がある．第二に，複数のストリームを並行に実行するために，アクセス先のメモリ領域をストリームごとに変更する必要がある．

そこで本研究では，オーバラップの実現に伴う開発者の負担を軽減することを目的として，開発者の作業を部分的に自動化するミドルウェアを開発する．本研究が対象とするアプリケーションでは，カーネル実行およびそれに伴う入出力データの転送からなるタスクを複数個実行する．ただし，タスク間にはデータ依存がないものとし，現在はタスク間で実行時間の差が小さいことを前提としている．ミドルウェアは異なるタスク間でオーバラップを実現するために，命令バッファに命令を蓄えた上で，命令を動的に並び替える．また，各ストリームにおいて均等な数の命令を実行することにより負荷分散を図る．さらに，複数のデバ

†1 大阪大学大学院情報科学研究科コンピュータサイエンス専攻
Department of Computer Science, Graduate School of Information Science and Technology, Osaka University

イスメモリ領域を仮想的に単一のメモリ領域として扱えるようにする．この目的のために，命令の実行時にアクセス先のメモリ領域を変更する．なお，ミドルウェアは C++言語のクラスとして実装する．開発者は，CUDA が提供する関数をミドルウェアが提供する関数へと置き換えることによりオーバラップを実現できる．

以下では，2 章で CUDA プログラムにおけるストリーム処理について述べる．次に，3 章でオーバラップの実現手法およびミドルウェアの実装について述べる．4 章ではミドルウェアを用いて実アプリケーションにストリーム処理を適用し，実験を実施した結果について報告する．最後に，5 章においてミドルウェアの有用性および今後の課題についてまとめる．

2. CUDA プログラムにおけるストリーム処理

CUDA では GPU をデバイス，CPU をホストとし，デバイスはコプロセッサとして動作する．デバイスで実行するコードはカーネル関数として記述し，ホスト側のコードから呼び出す．GPU はグラフィクスカード上のデバイスメモリと接続され，ホストはデバイスメモリとの間で入出力データを転送する．なお，GPU 上での計算に必要なデータ転送のうち，ホストからデバイスへのデータ転送をダウンロード，デバイスからホストへのデータ転送をリードバックと呼ぶ．本稿では，計算とはカーネルの実行を指す．また，デバイス上のメモリ領域のうち，計算に用いるために確保する領域をデバイスバッファと呼ぶ．さらに，カーネルの実行に必要な入力データのダウンロード，カーネルの実行および計算結果のリードバックをタスクと呼ぶ．

CUDA では命令を非同期に実行できる．このとき，2 つ以上のストリームを用いることにより，異なるタスクの命令を並行に実行できる．ストリーム処理を適用しない CUDA プログラムでは，すべてのタスクに含まれる命令を逐次実行する．**図 1** にこのときの命令実行の例を示す．この例ではダウンロード，カーネル実行およびリードバックを 2 回繰り返している．図 1 において点線で区切っている部分がそれぞれ 1 つのタスクである．

一方，ストリーム処理を適用して命令を並行に実行すると，異なるストリーム間でカーネル実行をデータ転送とオーバラップできる．**図 2** にこのときの命令実行の例を示す．矩形中の算用数字はデータ転送命令の発行順序を表し，ローマ数字はカーネル実行命令の発行順序を表す．命令の発行順序は開発者が CUDA の関数を呼び出す順序により決まる．この例では，あるタスクの命令をストリーム 1 において，別のタスクの命令をストリーム 2 において実行している．これにより，カーネル i の実行をダウンロード 2 と，リードバック 3 をカーネル ii の実行とオーバラップしている．ただし，ストリーム処理の制約により命令

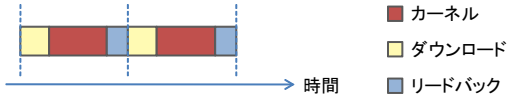

図 1 複数のストリームを用いない命令実行の例

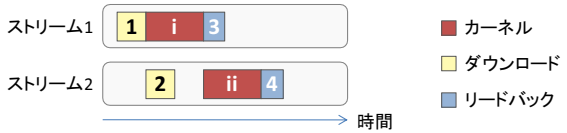

図 2 ストリーム処理により命令を並行に実行する例

をオーバラップできない例が存在する．

ここで，あるアプリケーションに対してストリーム処理を適用することにより，データ転送を隠蔽できるとする．このとき，実行時間の理論値 T は以下のように求められる．1 つのタスクがダウンロード，カーネル実行およびリードバックからなるとき，それぞれに要する時間を T_D，T_K および T_R とする．初回および最後のデータ転送はカーネル実行とオーバラップできないため，タスク数 m に対して次の式が成り立つ．

$$T = T_D + mT_K + T_R$$

なお，CUDA はバージョン 2.2 から，GPU・ホスト間でのゼロコピー通信[2] をサポートしている．ホストのメモリ空間をデバイスメモリ空間上にマップすることにより，GPU からホストに対して直接入出力できる．この際のデータ転送は自動的にカーネル実行とオーバラップされる．ただし，この機能は特定のメモリアクセスパターンにおいてのみ効果がある．

3. ミドルウェアを用いたオーバラップの実現

3.1 節では，オーバラップを実現するための手法について詳細に述べる．また，3.2 節では，3.1 節で述べる手法の，ミドルウェアにおける実装について述べる．さらに，3.3 節では，ミドルウェアを用いてアプリケーションにストリーム処理を適用する際に必要となる，プログラムに対する修正箇所について述べる．

開発者はミドルウェアに対してそれぞれのタスクの命令を発行する．この際に命令が属するタスクを指定することにより，ミドルウェアがタスクに関する情報を得られる．一方，ミドルウェアは制約を守ってオーバラップを実現できるように命令を動的に並び替える．並び

替えは開発者による命令の発行と非同期に行う．したがって，並び替えの時点ですべての情報が得られているとは限らない．それゆえ，ある時点で得られた情報のみを基に命令を並び替える．この目的のために，命令を一時的に蓄える命令バッファを用いる．

また，複数のストリームを用いて命令を並行に実行するためには，アクセス先のデバイスバッファを，命令を実行するストリームごとに変更する必要がある．そこで，開発者がアクセス先を明示的に変更する必要がないように，命令の実行時にアクセス先を選択する．実際のアクセス先に関わらず，開発者は 1 つの仮想バッファのみを用いる．

3.1 オーバラップの実現手法

ミドルウェアは，すべてのタスクが完了するまで以下を繰り返す．

(1) タスクを命令バッファに追加する
(2) 命令バッファ中のタスクをストリームに割り当てる
(3) ストリームに割り当てられているタスクから命令を発行する
(4) 実行が完了したタスクを削除する

開発者は並び替えと非同期に命令を発行する．したがって，3.2 節で述べる同期命令により，すべてのタスクが完了したと判断する．

まず，タスクを命令バッファに追加する．また，N 個のストリームが存在するとき，すべてのストリームに対してタスクを割り当てる．タスクの数が N に満たない場合は，新たなタスクを命令バッファに追加した時点で，タスクを割り当てていないストリームに対して，追加したタスクを割り当てる．なお，1 つのストリームに対して同時に割り当てるタスクは 1 つである．この理由は，1 つのストリームでは同じデバイスバッファを用いることから，あるストリームにおいて実行中のタスクが完了するまで，同じストリームでは別のタスクを実行できないためである．ここで，タスク間で実行時間の差が小さいため，負荷分散を実現するためにはタスクを各ストリームへと均等に割り当てればよい．すべての命令の実行が完了したタスクは命令バッファから削除し，別のタスクをストリームに割り当てる．

次に，GPU に対する命令の発行について述べる．I を命令の種類の集合とするとき，I はカーネル実行およびデータ転送からなる．すべての $i_1 \in I$ について，以下の方法でタスクの命令を発行する．

(1) i_1 をいずれかのストリームにおいて実行している場合は次の i_1 に移る．
(2) i_1 に対してオーバラップできる命令の種類を $i_2 \in I$ とする
(3) i_2 を実行していないすべてのストリームを巡回的に走査し，割り当てられているタスクの先頭にある命令を取得する
(4) 取得した命令の種類が i_1 であればその命令を発行して次の i_1 に移る．命令の種類が i_1 でない場合は引き続きストリームを走査する

なお，i_1 がカーネル実行のとき i_2 はデータ転送である．一方，i_1 がデータ転送のとき i_2 はカーネル実行である．このようにして，新たに発行する命令を，実行中の命令に対してオーバラップできる．

最後に，仮想バッファについて述べる．それぞれのストリームに対して存在するデバイスバッファを実バッファと呼ぶ．1 つの仮想バッファには N 個の実バッファが対応する．開発者がデバイスバッファを確保する際，ミドルウェアはすべてのストリームに対する実バッファを確保する．次に，確保した実バッファが対応する仮想バッファへのポインタを開発者に返す．一方，開発者が命令を発行する際には，ミドルウェアに仮想バッファへのポインタを渡す．命令をストリーム $k(1 \leq k \leq N)$ において実行するとき，仮想バッファへのポインタを，ストリーム k に対する実バッファへのポインタに置き換える．このようにして，開発者が明示することなく，アクセス先のデバイスバッファをストリームごとに変更できる．

3.2 ミドルウェアの実装

図 3 にミドルウェアの実装をシーケンス図[3] を用いて示す．図中の点線で描いた矩形がミドルウェアである．右向きの実線で描いた矢印は命令の呼び出し関係を表し，矢印の種類により同期実行および非同期実行を区別する．また，左向きの点線で描いた矢印は同期の完了を表す．さらに，垂直に描いた矢印は実行の流れを表す．ミドルウェアは，インタフェース関数および制御スレッドとして実装する．プログラムはミドルウェアに指示を出すためにインタフェース関数を呼び出す．インタフェース関数内では，制御スレッドに対して指示を出す．一方，制御スレッドは GPU に対して命令を発行する．制御スレッドをプログラムから独立して子スレッドとして実行することにより，制御スレッドにおける命令の並び替えおよび発行の完了を待つことなく，プログラムが任意に命令を発行できる．

インタフェース関数は以下の処理を行う．まず，制御スレッドを開始する．一方，制御スレッドを終了する際には，同期をとった後，制御スレッドに対して終了を指示する．次に，デバイスバッファの確保および破棄については，同期をとった後，制御スレッドに対してデバイスバッファの確保もしくは破棄を指示する．さらに，カーネル実行，ダウンロードおよびリードバックの各命令に関しては，命令が属するタスク単位で命令バッファに入れる．最後に，同期については，制御スレッドに対して同期を指示し，制御スレッドが同期の完了を伝えるまで待機する．

制御スレッドはプログラムが終了を指示するまでループを繰り返す．このループ中におい

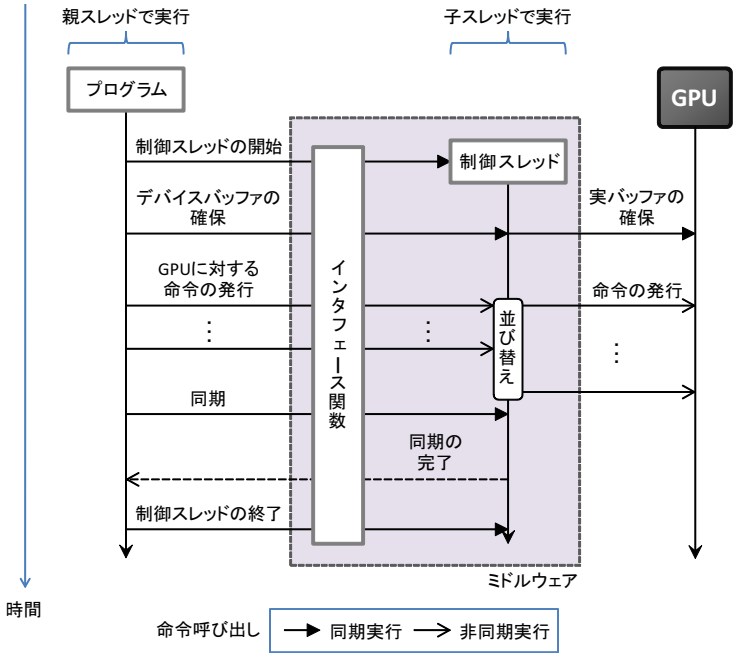

図 3 ミドルウェアの実装

てプログラムからの指示を受け取る．プログラムが終了を指示するとループを抜け，スレッドを終了する．制御スレッドは以下の処理を行う．まず，3.1 節で述べた方法によりデバイスバッファを確保する．一方，デバイスバッファの破棄については，仮想バッファに対応する実バッファを解放する．次に，命令バッファ中のタスクの有無を監視する．タスクが存在すれば，3.1 節で述べた方法により GPU に対して命令を発行する．最後に，同期については，GPU に対する命令がすべて完了するまで待機する．命令の実行完了後，プログラムに同期の完了を伝える．

3.3 ミドルウェアを用いる際に必要とする修正

ミドルウェアを用いてアプリケーションにストリーム処理を適用するためには，プログラム中の CUDA の関数を，ミドルウェアが提供する関数へと置き換える必要がある．また，初期化，終了および同期のためにいくつかの関数を追加する．この際に用いる関数を以下に挙げる．

- ストリームの数の設定：
  setupStream()
- 制御スレッドの開始および終了：
  startSection() および **finishSection**()
- デバイスバッファの確保および破棄：
  createDeviceBuffer() および **destroyDeviceBuffer**()
- カーネル実行，ダウンロードおよびリードバック：
  launchKernel()，**download**() および **readback**()
- 同期：
  synchronize()

プログラム内では，まずストリームの数を設定した後に制御スレッドを開始し，デバイスバッファの確保などの初期化を行う．続いて，すべてのタスクについて，カーネル実行，ダウンロードおよびリードバックの各命令を発行していく．それぞれの命令が属するタスクは，命令の引数として与えるタスク番号により区別する．ミドルウェアでは命令を非同期に実行するため，すべての命令を発行した後，同期を行い命令の実行完了を待つ必要がある．最後にデバイスバッファを破棄して制御スレッドを終了する．

図 4 はストリーム処理を行わず，すべての命令を逐次実行する場合のコードである．一方，**図 5** はミドルウェアを用いてストリーム処理を適用した後のコードである．obj はクラスのインスタンスである．また，make_arg() はカーネル関数に引数を指定する際に用いる関数である．初期化，終了および同期に関する処理の追加を除けば，関数を置き換える程度の変更にとどまる．

4. 評 価 実 験

ストリーム処理によるオーバラップの効果を確認するため，実機を用いた実験を行った．実験に用いた計算機は以下の通りである．CPU は Intel Xeon E5450（3.0GHz）であり，16GB の主記憶を搭載する．また，GPU は nVIDIA GeForce GTX 280 を用いた．これは 1GB のビデオメモリを搭載する．CUDA のバージョンは 2.2 であり，グラフィクスドライバのバージョンは 185.85 である．以下では，すべてのカーネルおよびデータ転送を逐次実行するプログラムを基本実装と呼ぶ．一方，ミドルウェアを用いてストリーム処理を適用したプログラムをストリーム実装と呼ぶ．

実験は 2 つのアプリケーションを対象に実施した．矩形関数のフーリエ変換は 2 つのカー

```
 1: float *d_in, *d_out;
 2: cudaMalloc((void **) &d_in, memsize);   // allocate device memory
 3: cudaMalloc((void **) &d_out, memsize);
 4:
 5: // process m tasks
 6: for (int i = 0;i < m;i++) {
 7:    cudaMemcpy(d_in, h_in + memsize * i, memsize, cudaMemcpyHostToDevice);
 8:    kernel<<< blocks, threads, 0 >>>(d_in, d_out, ...);
 9:    cudaThreadSynchronize();
10:    cudaMemcpy(h_out + memsize * i, d_out, memsize, cudaMemcpyDeviceToHost);
11: }
```

図 4 オリジナルのコード

```
 1: obj.setupStream(n);     // setup the number of streams
 2: obj.startSection();     // start Control thread
 3:    ...
 4: float *d_in, *d_out;
 5: obj.createDeviceBuffer(&d_in, memsize);                  // allocate device memory
 6: obj.createDeviceBuffer(&d_out, memsize);                 //   for each stream
 7:
 8: // process m tasks with n streams
 9: for (int i = 0;i < m;i++) {                              // i is the task number
10:    obj.download(i, d_in, h_in + memsize * i, memsize);   // download data to device
11:    obj.launchKernel(i, blocks, threads, 0, kernel,       // launch kernel
12:       make_arg(d_in), make_arg(d_out), ..., NULL);
13:    obj.readback(i, h_out + memsize * i, d_out, memsize); // readback result to host
14: }
15: obj.synchronize();     // wait for completion of tasks
16:    ...
17: obj.finishSection();   // finish Control thread
```

図 5 ミドルウェアを用いてストリーム処理を適用したコード

ネルを実行する．フーリエ変換で実行するタスク（以下，タスク 1）では，入力データをダウンロードした後，カーネル 1 およびカーネル 2 を実行する．最後に計算結果をリードバックする．一方，コーンビーム再構成[4] は，前半と後半で異なるタスクを実行する．前半のタスク（以下，タスク 2）では，入力データをダウンロードした後，カーネル 3 を実行する．最後に計算結果をリードバックする．後半のタスク（以下，タスク 3）では，入力データをダウンロードした後，カーネル 4 を実行する．カーネル 4 の計算結果はすべてのタスク 3 が完了した後にリードバックする．これを 8 個の計算領域に対して繰り返す．

ミドルウェアでは 1 回の命令発行により最大で数十マイクロ秒のオーバヘッドが生じる．ある命令を発行する時点で別の命令を実行している場合，実行中の命令により，新たな命令の発行によるオーバヘッドを隠蔽できる．フーリエ変換は 1 命令の実行に要する時間およびオーバヘッドの差が小さいため，オーバヘッドの隠蔽効果が低いと考えられる．一方，フーリエ変換と比較して，コーンビーム再構成は 1 命令の実行に要する時間が数十倍長いため，オーバヘッドを隠蔽しやすい．したがって，2 つの実験結果を比較することにより，オーバヘッドが実行時間に影響を与えないために必要となる，1 命令あたりの実行時間が分かる．

表 1 に各実験における命令の総実行回数を示す．1 タスクあたりの命令実行回数にタスクの実行回数をかけることにより，総実行回数を求めている．コーンビーム再構成では命令の実行回数が多いため，命令バッファへのアクセスが頻繁に発生する．したがって，インタフェース関数による命令バッファへのタスクの追加が，制御スレッドによる命令バッファへのアクセスと競合する可能性が高い．コーンビーム再構成の実行結果を理論値と比較することにより，このときの実行時間に対する影響を確認できる．

表 4 にフーリエ変換の実行結果を示す．ストリーム実装において基本実装と比較して全体の実行時間を約 14%短縮できている．これはストリーム処理によりデータ転送を隠蔽できたためである．一方，**表 2** より，ストリーム実装の実行時間は理論値より 2.8 ミリ秒長い．したがって，1 命令あたりの実行時間が数十マイクロ秒程度の場合，オーバヘッドを隠蔽できないことが原因となり，実行時間の短縮効果に影響を及ぼすことが分かる．

次に，コーンビーム再構成の実行結果について述べる．タスク 2 については，ストリーム実装において全体の実行時間を約 15%短縮できている．このときの実行時間が理論値に近いことより，カーネル実行とオーバラップできるデータ転送をすべて隠蔽できたことが分かる．このとき，**表 3** より，1 命令あたりの実行時間は数百マイクロ秒以上である．したがって，オーバヘッドが実行時間に影響を与えないためには，1 命令あたりの実行時間が数百マイクロ秒必要であることが分かる．さらに，1 万回以上の命令を実行する場合にも，命令バッファに対するアクセスの競合は実行時間に影響を与えないことが分かる．一方，タスク 3 については，基本実装よりも全体の実行時間が長くなっている．ここで，カーネル 4 ではテクスチャメモリ[2] から入力データを読み込むために，CUDA アレイ[2] を用いている．CUDA の仕様により，CUDA アレイを用いる場合は命令をオーバラップできない．したがって，タスク 3 の命令はすべて逐次実行される．それゆえ，命令発行のオーバヘッドにより実行時間が長くなっている．

表 1 命令の総実行回数（回）

	フーリエ変換	コーンビーム再構成	
	タスク 1	タスク 2	タスク 3
ダウンロード	6	1	5
カーネル実行	2	1	1
リードバック	2	1	0
タスクの実行回数	50	1200	1920
命令の総実行回数	500	3600	11520

表 2 フーリエ変換の実行時間の内訳（ミリ秒）

	タスク 1	
	1 タスクあたり	全タスク
ダウンロード	0.05	2.7
カーネル 1 実行	0.20	9.8
カーネル 2 実行	0.08	4.0
リードバック	0.05	2.7
合計	—	19.4

表 3 コーンビーム再構成の実行時間の内訳（秒）

	タスク 2		タスク 3	
	1 タスクあたり	全タスク	1 タスクあたり	全タスク
ダウンロード	0.0007	0.87	0.0035	6.69
カーネル実行	0.0079	9.43	0.0315	60.45
リードバック	0.0008	0.92	—	0.75
合計	—	11.22	—	68.42

以上のことから，ミドルウェアを用いてストリーム処理を適用することにより，実アプリケーションの実行時間を短縮できることが分かった．また，1 万回以上の命令を実行する場合にも，命令バッファへのアクセスの競合は実行時間に影響を与えないことが分かった．ただし，命令発行のオーバヘッドが実行時間に与える影響を小さくするためには，1 命令あたりの実行時間が数百マイクロ秒以上となるように，問題サイズを十分大きくする必要がある．

5. ま と め

本研究では，CUDA プログラムにおけるストリーム処理の実現に伴う開発者の負担を軽減するため，ストリーム処理を支援するミドルウェアを開発した．ミドルウェアでは命令を適切に並び替えて発行することによりオーバラップを実現する．また，効果的にオーバラップを実現するためにストリーム間で負荷分散を実施した．さらに，複数のデバイスバッファを仮想的に単一のデバイスバッファとして扱えるようにした．

表 4 実アプリケーションの実行時間

		基本実装	ストリーム実装	理論値
フーリエ変換（ミリ秒）	タスク 1	19.4	16.7	13.9
コーンビーム再構成（秒）	タスク 2	11.22	9.54	9.54
	タスク 3	68.42	70.60	68.42

評価実験では実アプリケーションを対象にストリーム処理を適用し，実行時間を計測した．その結果，矩形関数のフーリエ変換におけるデータ転送を計算により隠蔽できた．コーンビーム再構成においては一方のカーネル実行に伴うデータ転送を完全に隠蔽できた．フーリエ変換でデータ転送を隠蔽できたのは，すべてのタスクをストリーム処理できたことによる．一方，コーンビーム再構成で一方のカーネル実行に伴うデータ転送が隠蔽できなかったのは，CUDA の仕様によりオーバラップを実現できなかったためである．また，命令の実行回数が 1 万回以上の場合にも，命令バッファに対するアクセスの競合は実行時間に影響を与えないことが分かった．さらに，命令発行のオーバヘッドが実行時間に与える影響を小さくするためには，1 命令あたりの実行時間は数百マイクロ秒以上必要であることが分かった．

今後の課題としては，タスクの実行時間にばらつきがある場合にも負荷を適切に分散することを考えている．

謝辞 本研究の一部は，科学研究費補助金基盤研究（A）（2024002）の補助による．

参 考 文 献

1) GPGPU: General-Purpose Computation Using Graphics Hardware (2007). `http://www.gpgpu.org/`.
2) nVIDIA Corporation: CUDA Programming Guide Version 2.2 (2009).
3) Object Management Group: UML 2.2 Superstructure and Infrastructure, `http://www.uml.org/` (2009).
4) Okitsu, Y., Ino, F. and Hagihara, K.: Fast Cone Beam Reconstruction Using the CUDA-enabled GPU, *Proc. 15th Int'l Conf. High Performance Computing (HiPC'08)*, pp.108–119 (2008).