Vol.2010-HPC-126 No.19
2010/8/4

複数のCUDA互換GPUによるストリーム処理のためのミドルウェア

中川進太[†1]　伊野文彦[†1]　萩原兼一[†1]

GPU（Graphics Processing Unit）向けの開発環境であるCUDA（Compute Unified Device Architecture）では，ストリーム処理によりGPUにおける計算をCPU・GPU間のデータ転送とオーバラップできる．本稿では，複数のGPUを用いたストリーム処理に要する開発負担を軽減するためのミドルウェアについて述べる．提案するミドルウェアは，線形の速度向上を実現するために各GPUに対して均等に計算処理を割り当てる．また，効率的なオーバラップを実現するために，API関数の実行順序を動的に並び替える．開発者はCUDAのAPI呼び出し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，光学シミュレータおよびDNAの配列アライメントプログラムに対してミドルウェアを適用した．このとき，前者において4台のGPUを用いることで4.41倍の高速化を実現した．

A Middleware for Stream Processing Using CUDA Compatible GPUs

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

Compute unified device architecture (CUDA) is a development environment for graphics processing units (GPUs). By stream processing, it allows us to overlap GPU computation with data transfer between the GPU and the CPU. This paper presents a middleware for reducing development efforts required to perform stream processing using multiple GPUs. To achieve linear speedup, our middleware uniformly assigns tasks to each GPU. It then dynamically reorders the order of API executions to achieve efficient overlap. Using the middleware, developers can achieve overlap by replacing CUDA API calls with our API calls. In experiments, we apply the middleware to an optical simulator and a DNA sequence alignment program. As a result, it accelerates the simulator by 4.41 times using 4 GPUs.

1. はじめに

グラフィクス処理用の演算器であるGPU（Graphics Processing Unit）は，ストリーム処理に最適化することでCPUの10倍以上の浮動小数点演算性能を有している[1)]．そこで，GPUを汎用計算に用いるGPUコンピューティング[2)]が注目を集めている．このための開発環境として，NVIDIA社はC言語を拡張したCUDA[3)]（Compute Unified Device Architecture）を提供している．CUDAではストリームプログラムモデル[4)]に基づくアプリケーションを開発できる．このモデルでは，入出力データは互いにデータ依存がない複数のストリーム要素に分割できる．また，アプリケーションは入力ストリームに対して複数のカーネルを適用することで計算を実行し，出力ストリームを生成する．この処理は全ストリーム要素に対して繰り返し適用する．また，異なるストリーム要素に対する処理は並列に実行できる．

CUDAプログラムでは，GPUで動作するプログラム（カーネル）をCPUが起動する．GPUは主記憶を直接参照できないため，入出力ストリームはGPU側のデバイスメモリに格納する．したがって，あらかじめ入力ストリームをデバイスメモリに転送し，計算後に出力ストリームを主記憶に転送する必要がある．これらのデータ転送はそれぞれダウンロードおよびリードバックと呼び，計算上の性能ボトルネックとなりえる．このボトルネックを解消するため，CUDAはストリーム処理の機構を提供している．CUDAにおけるストリーム処理では，逐次実行する関数の系列であるCUDAストリームを用いる．このとき，複数のCUDAストリームから非同期APIを用いて関数を実行することで，異なるCUDAストリーム間でデータ転送をカーネルとオーバラップできる．

一方，オーバラップを実現するためには，開発者は適切な順序でAPIを呼び出す必要がある．中川ら[5)]は動的なスケジューリングを実現するミドルウェアを提案しているが，複数GPU環境におけるストリーム処理には対応していなかった．複数GPU環境では，CUDAの仕様によりGPUごとに異なるスレッドからAPIを呼び出す必要がある．また，線形の性能向上を実現するためにはGPU間の負荷分散が必要となる．さらに，ハードウェアによっては複数のGPUが1本のデータ転送バスを共有する場合がある．このとき，バスを共有するGPU間では同時にデータを転送できない．したがって，特定のGPUがデータを転

†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

送し続けると，他の GPU は転送待ちによりカーネルを実行できず，処理の不均衡が生じる可能性がある．

本稿では，複数 GPU 環境において効率的なストリーム処理を実現するためのミドルウェアについて述べる．提案するミドルウェアでは，線形の速度向上，および開発負担の軽減を両立するために，文献 5) におけるスケジューリングに加えて以下の 3 点を実現する．まず，スレッドの生成および管理を自動化する．次に，各 GPU に対して均等に計算処理を割り当てる．最後に，バスを共有する GPU 間では，各 GPU に対するデータ転送を均等にスケジューリングする．なお，ミドルウェアはストリームプログラムモデルに基づくアプリケーションにのみ適用可能である．また，現時点では，ストリーム要素間でカーネル実行やデータ転送に要する時間の差が小さいことを前提としている．

以下では，2 節で 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 節でミドルウェアの設計およびスケジューリング手法について述べる．4 節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た結果について報告する．最後に，5 節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

本節では，本研究で対象とするストリームプログラムモデルについて述べる．また，CUDA プログラムにおけるストリーム処理，およびその制約について述べ，これらをモデル化する．

入力ストリームに対して m (≥ 1) 個のカーネルを適用するものとし，それぞれのカーネルを $f_1, f_2, \cdots, f_m$ とする．また，I および O をそれぞれ入力ストリームおよび出力ストリームとし，n 個のストリーム要素に分割可能とする．ここで，e_i $(1 \leq i \leq n)$ を入力ストリーム要素とすると，$I = \{e_1, e_2, \cdots, e_n\}$ と表せる．同様に，g_i $(1 \leq i \leq n)$ を出力ストリーム要素とすると，$O = \{g_1, g_2, \cdots, g_n\}$ と表せ，1 つの入力ストリーム要素に対する計算は式 (1) で与えられる．

$$g_i = f_m \circ f_{m-1} \circ \cdots \circ f_1(e_i) \tag{1}$$

すなわち，タスクを同一のストリーム要素に対する計算処理と定義するとき，1 つの CUDA プログラムでは n 個のタスクを実行する．

ここで，$\rightarrow$ がデータ依存関係を表すものとすると，$f_{s,i} \rightarrow f_{t,j}$ $(1 \leq s, t \leq m, 1 \leq i, j \leq n)$ は，e_i に対する s 番目のカーネルを，e_j に対する t 番目のカーネルより先に実行する必要があることを示す．あるストリーム要素に関するデータ依存関係は式 (2) で与えられる．

$$\forall s < t,\ f_{s,i} \rightarrow f_{t,i}, \tag{2}$$

すなわち，あるストリーム要素に対するカーネルは逐次実行する必要がある．一方，$\not\rightarrow$ を $\rightarrow$ の否定とすると，異なるストリーム要素間のデータ依存関係は式 (3) で与えられる．

$$\forall i \neq j,\ f_{s,i} \not\rightarrow f_{t,j} \tag{3}$$

すなわち，異なるストリーム要素は独立に処理できる．

次に，文献 5) より，1 台の GPU を用いた CUDA プログラムにおけるストリーム処理は以下のようにモデル化できる．T_D，T_K および T_R を，それぞれタスク 1 つ当たりのダウンロード，カーネルおよびリードバックの実行時間とすると，カーネル実行時間の合計は nT_K となる．また，カーネル実行により隠蔽可能なデータ転送の実行時間 T_S は式 (4) で与えられる．

$$T_S = (n-1)(T_D + T_R) \tag{4}$$

このとき，最適なストリーム処理を実現した場合の実行時間の理論値 T_{single} は，式 (5) で与えられる．

$$T_{single} = \begin{cases} T_D + nT_K + T_R, & \text{if } nT_K > T_S, \\ n(T_D + T_R), & \text{otherwise.} \end{cases} \tag{5}$$

ただし，式 (5) は，**図 1(a)** のようにカーネルを連続実行する場合に成り立つ．これは，**図 2(a)** のように，l (≥ 1) 個の CUDA ストリームを用いて，非同期 API により事前にすべてのダウンロードを呼び出した後，カーネルおよびリードバックを実行することで実現できる[3]．全関数の呼び出し後には，cudaThreadSynchronize() 関数による同期が必要となる．

一方，**図 2(b)** のように，タスクごとにダウンロード，カーネルおよびリードバックを実行する場合には，オーラバップを実現できない．このとき**図 1(b)** より，あるタスクのリードバックが後続のタスクのダウンロードをブロックしていることが分かる．これは，CUDA プログラムにおけるストリーム処理では以下の制約が存在するためである．

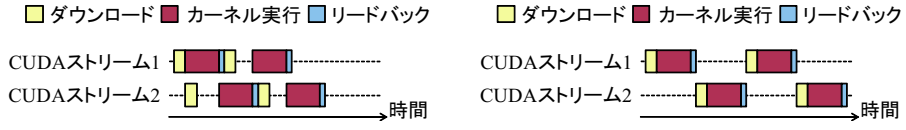
C1. 同一プログラム内のデータ転送は逐次実行される

C2. 同一プログラム内のカーネルは逐次実行される

さらに，以上のモデルを複数 GPU 環境に対して拡張する．c を GPU の台数，P を GPU の集合，p_i $(0 \leq i \leq c-1)$ を GPU とすると，$P = \{p_0, p_1, \cdots, p_{c-1}\}$ と表せる．また，n_S を 1 台当たりのタスク数とすると，$n_S = \lceil n/c \rceil$ と表せ，カーネル実行時間の合計は $n_S T_K$ となる．ここで，複数 GPU 環境には以下の制約が存在する．

C3. バスを共有する GPU 間では，同時にデータを転送できない

h $(1 \leq h \leq c)$ をバスの本数とすると，バスを共有する GPU ごとに，P を h 個の部分集合



(a) オーバラップを実現できている場合　(b) オーバラップを実現できていない場合

図 1 ストリーム処理による関数の実行系列

$P_0, P_1, \cdots, P_{h-1}$ へと直和分割できる．ただし，P_j $(0 \leq j \leq h-1)$ は j 番目のバスを共有する GPU の集合である．$|P_j|$ が最大となる j について $a = |P_j|$ とし，n_M を P_j に属する GPU が処理するタスク数とすると，$n_M = an_S$ と表せる．さらに，P_j に属する GPU では，制約 C3 より**図 3** のように初回のカーネル実行に最大で aT_D の遅延が生じる．したがって，カーネル実行により隠蔽可能なデータ転送の実行時間 T_M は式 (6) で与えられる．

$$T_M = (n_M - a)T_D + (n_M - 1)T_R \tag{6}$$

以上のことから，c 台の GPU を用いる場合の実行時間の理論値 T_{opt} は式 (7) で与えられる．

$$T_{opt} = \begin{cases} aT_D + n_S T_K + T_R, & \text{if } n_S T_K > T_M, \\ n_M(T_D + T_R), & \text{otherwise.} \end{cases} \tag{7}$$

ただし，式 (7) が成立するためには，実行時間に対してカーネル実行時間もしくはデータ転送時間のいずれかが支配的となっており，もう一方を隠蔽できる必要がある．

3. ミドルウェアによるストリーム処理の支援

本研究は図 2(b) のプログラム構造を維持しつつ，オーバラップを実現することを目的とする．この目的のために，発行された関数を動的に並び替え，適切な順序で実行する．本節では，ミドルウェアにおけるスケジューラの実装について述べる．

3.1 ミドルウェアの設計

NVIDIA 社はグラフィクスドライバ内部の仕様を公開していないため，アプリケーションレベルで並び替えを実現する．開発者はソースコード内の CUDA の API 呼び出しをミドルウェアのものに置き換える．一方，ミドルウェアは関数の実行前に実行順序を動的に並び替える．2 節で述べたように，カーネル実行に待ちが生じる場合には T_{opt} に近い実行時間を実現できない．したがって，カーネルの実行待ちを避けるために，関数を蓄えるための関数バッファを用いる．スケジューラは，CUDA におけるランタイム API の仕様[3] に基づき，各 GPU に対して生成したスレッド上で実行する．

```
Input: Input stream I = {e_1, e_2, ..., e_n}, size n, and number l of CUDA streams.
Output: Output stream O = {g_1, g_2, ..., g_n}.
 1: cudaStream_t str[l];
 2: for i = 1 to n do
 3:     Download stream element e_i using str[i mod l];
 4: end
 5: for i = 1 to n do
 6:     Launch kernel to compute g_i from e_i using str[i mod l];
 7: end
 8: for i = 1 to n do
 9:     Readback output stream g_i using str[i mod l];
10: end
11: cudaThreadSynchronize();
```

(a) 事前発行版[3]

```
Input: Input stream I = {e_1, e_2, ..., e_n}, size n, and number l of CUDA streams.
Output: Output stream O = {g_1, g_2, ..., g_n}.
1: cudaStream_t str[l];
2: for i = 1 to n do
3:     Download stream element e_i using str[i mod l];
4:     Launch kernel to compute g_i from e_i using str[i mod l];
5:     Readback output stream g_i using str[i mod l];
6: end
7: cudaThreadSynchronize();
```

(b) 個別発行版

図 2 ストリーム処理を適用したプログラムの擬似コード

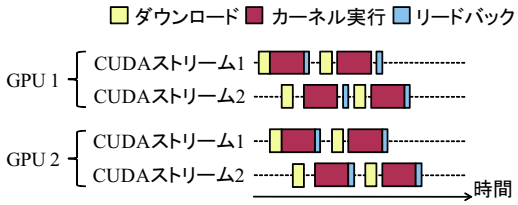


図 3 複数 GPU 環境における関数の実行系列

図 3 のようにカーネルを実行し続けるために，各 CUDA ストリームに対して均等にタスクを割り当てることにより，CUDA ストリーム間で負荷を分散する．さらに，GPU の台数に比例した高速化のために，各 GPU に対して均等にタスクを割り当てることにより，GPU 間においても負荷分散を実現する．また，バスを共有する GPU 間では，転送待ちに起因する処理の不均衡を回避するために，FCFS（First Come, First Served）方式によりデータ

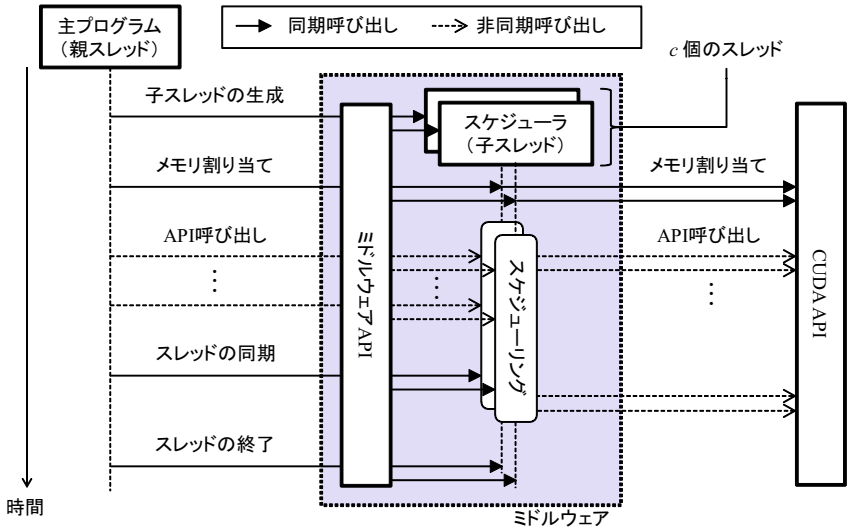


図 4 ミドルウェアの実装

転送を均等にスケジューリングする．データ転送のスケジューリングには，バスごとに存在する優先度キューを用いる．以上のことを実現するためのアルゴリズムについては 3.2 節において述べる．

図 4 にミドルウェアの実装を示す．実線は同期実行する関数を，破線は非同期実行する関数をそれぞれ表している．図のように，ミドルウェアの API 呼び出しにより CUDA の API 呼び出しを置き換える．ミドルウェアは，プログラムの先頭において c 台の GPU に対して，それぞれを管理するための子スレッドを生成する．その後，主プログラムによって呼び出された API 関数において，タスクの割り当ておよび関数のバッファリングを実行する．一方，子スレッドは各 CUDA ストリームに対してデバイスメモリを割り当てた上で，関数バッファから適切な関数を選択し，非同期に実行する．子スレッドは親スレッドと非同期に実行するため，全 API の実行完了後にスレッドを同期する．全スレッドの同期後，子スレッドは終了する．なお，関数バッファおよび優先度キューは複数のスレッドからアクセスするため，クリティカルセクションを用いて排他制御を実現する．

3.2 スケジューラ

実行する関数は以下の優先度に基づいて選択する．まず，GPU 上で常にカーネルを実行し続けるためにカーネル実行の優先度を最高とする．次に，後続のカーネルを実行可能な状態にするためにダウンロードに対して 2 番目の優先度を与える．最後に，リードバックに対して最低の優先度を与える．

スケジューリングには，CUDA ストリームの集合 S，関数バッファの集合 Q，優先度キューの集合 R を用いる．ここで，l を各 GPU 上の CUDA ストリームの数とする．また，d $(0 \leq d \leq c-1)$ を GPU を識別するためのデバイス識別子とする．$s_{d,i}$ $(0 \leq d \leq c-1, 0 \leq i \leq l-1)$ を CUDA ストリームのオブジェクトとすると，$S = \{s_{0,0}, s_{0,1}, \cdots, s_{0,l-1}, s_{1,0}, s_{1,1}, \cdots, s_{1,l-1}, \cdots, s_{c-1,0}, s_{c-1,1}, \cdots, s_{c-1,l-1}\}$ と表せる．また，$q_{d,i}$ $(0 \leq d \leq c-1, 0 \leq i \leq l-1)$ を $s_{d,i}$ に関する関数バッファとすると，$Q = \{q_{0,0}, q_{0,1}, \cdots, q_{0,l-1}, q_{1,0}, q_{1,1}, \cdots, q_{1,l-1}, \cdots, q_{c-1,0}, q_{c-1,1}, \cdots, q_{c-1,l-1}\}$ と表せる．さらに，b $(0 \leq b \leq h-1)$ を各 GPU が接続されているバスを識別するためのバス識別子とし，r_b をバス b に関する優先度キューとすると，$R = \{r_0, r_1, \cdots, r_{h-1}\}$ と表せる．

スケジューリングアルゴリズムの概要を以下に示す．

A1. タスクの割り当ておよび関数のバッファリング
主プログラムがミドルウェアの関数を呼び出すごとに，その関数が処理するタスクを CUDA ストリーム $s_{d,i}$ に割り当てる．また，$s_{d,i}$ に関する関数バッファ $q_{d,i}$ に，呼び出した関数を挿入する．

A2. 関数の選択および実行
実行中のカーネルおよびデータ転送関数の状態を監視する．カーネルについては，実行完了後ただちに関数バッファから新たなカーネルを選択して実行する．一方，データ転送については，GPU が接続されているバスにおいてデータを転送しておらず，かつバスが割り当てられていれば，関数バッファから新たなデータ転送関数を選択して実行する．新たに実行する関数 f は，以下の 2 つのうちいずれかの条件を満たす場合に選択する．一方は，いずれの CUDA ストリームでも関数を実行していない場合である．もう一方は，f を実行する CUDA ストリームとは異なる CUDA ストリームにおいて実行中の関数を，f とオーバラップできる場合である．

A1 は親スレッドにおいて，A2 は子スレッドにおいて互いに非同期に実行する．

図 5 に，タスクの割り当ておよび関数のバッファリングを実行するためのアルゴリズムを示す．このアルゴリズムは，ミドルウェアの API 関数において実行する．関数 f およびその引数 $args$ に加えて，関数 f をタスクと関連付けるためのタスク識別子 t を与える．図 2(b) の場合には，ループ変数 i をタスク識別子として用いることができる．このタスク

Input: CUDA function f, its arguments $args$, task identifier t, shared buffer set Q, number c of GPUs, and number l of CUDA streams.
Output: Updated buffer set Q.
1: $i := t \bmod p$; $j := \lfloor t/c \rfloor \bmod l$;
2: Enter critical section;
3: Enqueue $f(args)$ into $q_{i,j} \in Q$;
4: Leave critical section;

図 5 タスクの割り当ておよび関数のバッファリングの擬似コード

識別子を用いて，各 GPU に対して巡回的にタスクを割り当てる．同様に，同一の GPU 上の各 CUDA ストリームに対しても巡回的にタスクを割り当てる．なお，タスクはバスの共有状況に関わらず各 GPU に対して均等に割り当てる．したがって，実行時間に対してデータ転送時間が支配的になる状況下では，バスを共有する GPU においてデータ転送がボトルネックとなりえる．

図 6 に，制御スレッドにおける関数の選択および実行のアルゴリズムを示す．このアルゴリズムは，集合 S，Q および R に加えて，CUDA ストリームの数 l，デバイス識別子 d およびバス識別子 b をとる．u, v $(0 \leq u, v \leq l-1)$ は，GPU およびバスを占有している CUDA ストリームの番号をそれぞれ表す．これらの変数を用いて関数を実行する CUDA ストリームを巡回的に切り替えることにより，CUDA ストリーム間の負荷分散を実現する．

実行する関数の選択においては，**図 7** に示す select() 関数を呼び出す．この関数は，選択する API 関数の種類 $type$，および CUDA ストリームの番号 j をとる．ここで，関数の種類は “kernel”，“download” もしくは “readback” である．$s_{d,j}$ および $q_{d,j}$ から開始して，巡回的に CUDA ストリームおよび関数バッファの状態を調べる．関数バッファの先頭に実行可能な関数が存在する場合，関数および CUDA ストリーム番号の組 $\langle f(args), i \rangle$ を返す．select() 関数の戻り値を元に，$s_{d,i}$ において $f(args)$ を実行する．

ダウンロードおよびリードバック関数の選択においては，事前に**図 8** に示す require() 関数を実行する．この関数は CUDA ストリーム番号 j をとり，$q_{d,j}$ から開始して関数バッファの状態を調べる．関数バッファの先頭にデータ転送関数が存在する場合には，バスの割り当てを要求する．その後，r_b の先頭が d であればバスが割り当てられているため，select() 関数を実行する．データ転送の完了後にバスを解放する．

Input: CUDA stream set S, buffer set Q, priority queue set R, number l of CUDA streams, device identifier d, and bus identifier b.
1: $u := 0$; $v := 0$; // identifiers of active CUDA streams
2: **while** (command left in Q) **or** (no sync. request) **do**
3: **if** GPU is idle **then**
4: $\langle f(args), i \rangle :=$ select$(S, Q, l, d,$ “kernel”$, u)$;
5: **if** $f \neq NULL$ **then**
6: $u := i$; // update active identifier
7: Execute $f(args)$ using $s_{d,i} \in S$;
8: **end**
9: **end**
10: **if** no request of bus allocation **then**
11: require$(Q, r_b \in R, l, d, v)$; // require bus allocation
12: **end**
13: Set w as the first element of $r_b \in R$;
14: **if** (b-th bus is idle) **and** $(w = d)$ **then**
15: $\langle f(args), i \rangle :=$ select$(S, Q, l, d,$ “download”$, v)$;
16: **if** $f = NULL$ **then**
17: $\langle f(args), i \rangle :=$ select$(S, Q, l, d,$ “readback”$, v)$;
18: **end**
19: **if** $f \neq NULL$ **then**
20: $v := i$; // update active identifier
21: Execute $f(args)$ using $s_{d,i} \in S$;
22: **end**
24: **end**
25: **if** data transfer has done **then**
23: Enter critical section;
26: Dequeue the first element of $r_b \in R$; // release the bus
27: Leave critical section;
28: **end**
29: **end**

図 6 関数の選択および実行の擬似コード

4. 評価実験

本節では，ミドルウェアを用いたストリーム処理のオーバヘッド，および実アプリケーションへの適用結果について述べる．実験では，CPU として 4 コアの Intel Xeon E5450（3.0 GHz）を持ち，16GB の主記憶を搭載する計算機を用いた．計算には，4 台の GPU を搭載する NVIDIA Tesla S1070-400 を用いる．これは 2 本の PCI Express 2.0 x8 バスを用いて計算機に接続しており，各バスを 2 台の GPU で共有する．OS は CentOS 5.3 x86_64

```
Function select(S, Q, l, d, type, j)
Input: CUDA stream set S, buffer set Q, number l of CUDA streams, device identifier d,
       command type type, and CUDA stream identifier j.
Output: Pair ⟨f(args), i⟩ of executable function f(args)
        and CUDA stream identifier i.
 1: for k = j to j + l − 1 do
 2:   i := k mod l;
 3:   if (s_{d,i} ∈ S is idle) and (q_{d,i} ∈ Q is not empty) then
 4:     Set f as the first element of q_{d,i}; // f: buffered command
 5:     if f is type command then
 6:       Enter critical section;
 7:       Dequeue f(args) from q_{d,i}; // args: arguments of f
 8:       Leave critical section;
 9:       return ⟨f(args), i⟩;
10:     end
11:   end
12: end
13: return ⟨NULL, 0⟩; // no left
```

図 7 実行する関数を選択する擬似コード

であり，グラフィクスドライバのバージョンは 195.18 である．また，CUDA はバージョン 2.3 を用いた．以下では，1 台の GPU を用いてすべてのタスクを逐次処理するプログラムを基本実装（Original）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（$c = 1, 2, 3, 4$）と呼ぶ．ただし，$c = 2$ のときにはバスを共有していない 2 台の GPU を用いる．また，時間比率 r_T をカーネル実行時間に対するデータ転送時間の比率とすると，$r_T = (T_D + T_R)/T_K$ と表せる．

4.1 オーバヘッドの評価

計算集中型およびメモリ集中型のダミープログラムを用いて，ミドルウェアによるストリーム処理において生じるオーバヘッドについて評価した．ここで，基本実装の実行時間 T_{base} と比較した場合の速度向上比を r_I とすると，$r_I = T_{base}/T$ と表せる．また，実行時間 T を T_{opt} と比較した場合のオーバヘッドを δ とすると，$\delta = (T - T_{opt})/T_{opt}$ と表せる．実験では入出力ストリームをそれぞれ 24 個のストリーム要素に分割した（$n = 24$）．各ストリーム要素は 3MB〜30MB の integer 型配列であり，カーネル実行時間を一定に維持しながら配列の要素数を変化させることにより，時間比率を調整できる．

計算集中型プログラムはデバイスメモリから値を読み出し，レジスタ上でシフト演算を繰り返す．その後，結果をデバイスメモリに書き出す．メモリアクセスは Coalesced 参照[3]

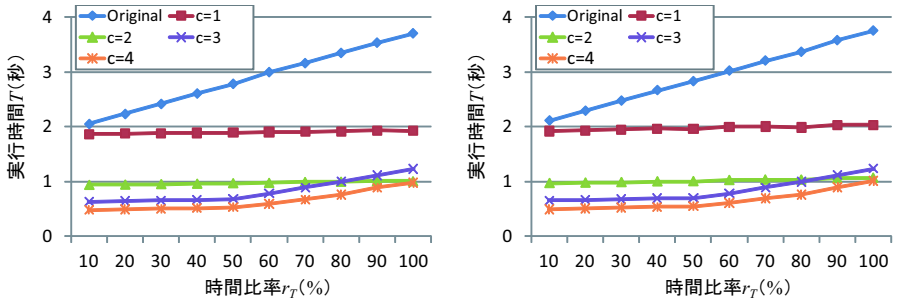
```
Function require(Q, r, l, d, j)
Input: Buffer set Q, priority queue r, number l of CUDA streams, device identifier d, and
       CUDA stream identifier j.
Output: Updated priority queue r.
 1: for k = j to j + l − 1 do
 2:   i := k mod l;
 3:   if q_{d,i} ∈ Q is not empty then
 4:     Set f as the first element of q_{d,i};
 5:     if (f is "download" command) or (f is "readback" command) then
 6:       Enter critical section;
 7:       Enqueue d into r;
 8:       Leave critical section;
 9:       return;
10:     end
11:   end
12: end
```

図 8 データ転送パスの割り当てを要求する擬似コード

の条件を満たす．一方，メモリ集中型プログラムはデバイスメモリへの読み書きを繰り返すことにより，メモリのバンド幅を計測する[6]．データ転送はデバイスメモリのバンド幅を消費するため，メモリ集中型プログラムの実行結果から，バンド幅が逼迫している状況下におけるストリーム処理の有用性を評価できる．

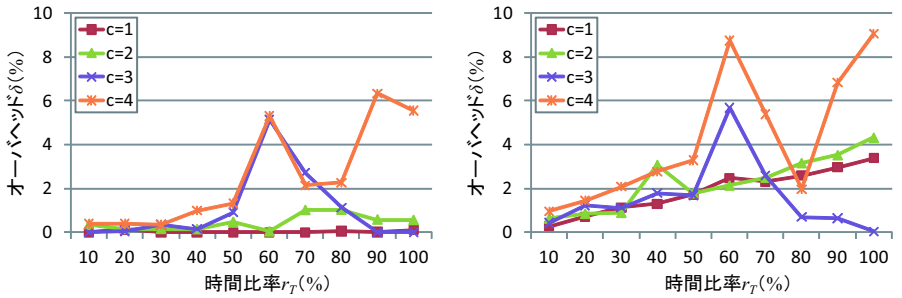
図 9(a) および**図 9(b)** にダミープログラムの実行時間 T を示す．$c = 1$ を基本実装と比較すると，データ転送の隠蔽により最大で 1.92 倍の高速化を実現できている．また，$c = 1$ との比較では，$c = 2$ においてはいずれの場合にも，$c = 3, 4$ においては $r_T \leq 0.5$ の場合に，GPU の台数に比例した高速化を実現できたことが分かる．ただし，$c = 3, 4$ においては，$r_T \geq 0.6$ の場合にはデータ転送時間を隠蔽できないため，時間比率に比例して実行時間が増加している．さらに，$c = 3$ では $r_T \geq 0.8$ の場合に $c = 2$ と実行時間が逆転している．これは，3.2 節で述べたように，データ転送がボトルネックとなっているためである．以上のことから，カーネル実行時間が十分に長い場合には，ストリーム処理によりデータ転送を隠蔽し，複数 GPU 環境において線形の速度向上を実現できるといえる．

次に，**図 10(a)** および**図 10(b)** にオーバヘッド δ を示す．メモリ集中型では，計算集中型と比較してオーバヘッドが増加する傾向が見られる．これは，カーネル内でのメモリアクセスがデータ転送と競合することにより，カーネル実行時間 T_K が増加していることが原因である．ストリーム処理中のカーネル実行時間を T'_K，T_K の増加率を δ_K とすると，



(a) 計算集中型プログラム　(b) メモリ集中型プログラム

図 9　ダミープログラムの実行時間 T(秒)



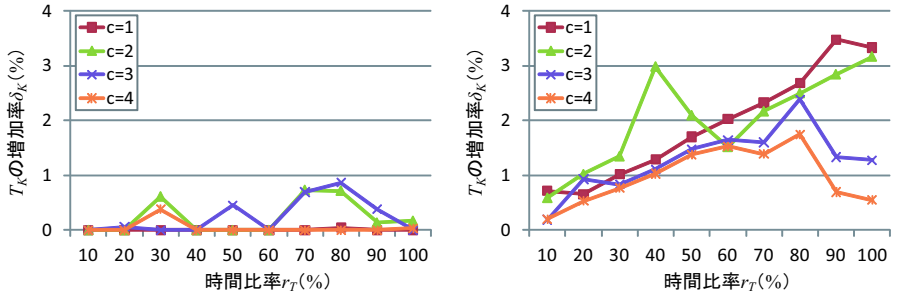
(a) 計算集中型プログラム　(b) メモリ集中型プログラム

図 10　ダミープログラムのオーバヘッド δ(%)

$\delta_K = (T'_K - T_K)/T_K$ と表せる．**図 11(a)** および**図 11(b)** にダミープログラムにおける T_K の増加率 δ_K を示す．δ_K は計算集中型では 1%未満だが，メモリ集中型では最大で 3.5%に達する．また，$c = 3, 4$ においては，式 (7) の境界付近に当たる $0.5 < r_T < 0.7$ の場合にオーバヘッドが増加している．これは，カーネル実行時間およびデータ転送時間の差が小さく，式 (7) の前提条件が成立していないためであると考えられる．以上のことから，カーネル実行時間が十分に長い場合には，数%のオーバヘッドでストリーム処理を実現できるといえる．ただし，バンド幅が逼迫する場合には，メモリアクセスの競合に起因するオーバヘッドの増加が生じる可能性がある．

4.2　実アプリケーションへの適用

実アプリケーションにおけるストリーム処理の有効性を確認するために，2 種類のプログラムに対してミドルウェアを適用した．一方はフーリエ変換に基づく光学シミュレータ（FFT）である．このプログラムは，6 種類の入力パラメータに対して 2 個のカーネルをそれぞれ 1 回ずつ適用し，2 個のテーブルを出力する．1 タスク当たりの入力パラメータはいずれも 4KB の float 型配列であり，出力テーブルはいずれも 16KB である．もう一方は Needleman-Wunsch アルゴリズムによる配列のグローバルアライメント（NW）であり，文献 7) の実装に基づく．このプログラムは，2 個の行列を入力として，まず 1 個目のカーネルを 128 回実行する．その完了後に 2 個目のカーネルを 127 回実行し，アライメント結果の行列を出力する．1 タスク当たりの入力行列，スコア行列および出力行列は，いずれも 16MB の integer 型要素からなる．なお，FFT では 600 個，NW では 24 個のタスクを用



(a) 計算集中型プログラム　(b) メモリ集中型プログラム

図 11　カーネル実行時間 T_K の増加率 δ_K(%)

いた．**表 1** に 1 タスク当たりの実行時間の内訳を示す．FFT では $r_T = 0.14$ であり，NW では $r_T = 1.61$ である．したがって，NW ではデータ転送が大きなボトルネックであるといえる．

表 2 に実行時間 T，速度向上比 r_I，およびオーバヘッド δ を示す．FFT では，$c = 1$ において 1.12 倍の速度向上比を達成した．また，$c = 1$ との比較では GPU の台数に比例した高速化を実現できたことが分かる．このときのオーバヘッドは最大で 3.1%であり，これはカーネル実行時間の増加に起因する．一方，NW では $c = 1$ において 1.32 倍，$c = 2$ にお

表 1 1 タスク当たりの実行時間の内訳 (ミリ秒)

	FFT		NW	
	実行時間	割合 (%)	実行時間	割合 (%)
ダウンロード	0.51	8.9	87.75	42.3
カーネル 1	3.20	56.1	40.30	19.4
カーネル 2	1.79	31.4	39.16	18.9
リードバック	0.21	3.6	40.49	19.5
合計	5.71	100.0	207.69	100.0

表 2 実アプリケーションの実行時間 T(秒)，速度向上比 r_I(%) およびオーバヘッド δ(%)

	FFT			NW		
	T	r_I	δ	T	r_I	δ
基本実装	3.41	-	-	4.99	-	-
$c = 1$	3.06	1.12	1.9	3.78	1.32	22.8
$c = 2$	1.54	2.22	2.4	1.96	2.55	27.4
$c = 3$	1.03	3.33	2.6	2.13	2.34	4.0
$c = 4$	0.77	4.41	3.1	1.74	2.87	13.0

いて 2.55 倍の速度向上比を達成している．ただし，$c = 3, 4$ ではデータ転送時間を隠蔽できないため，線形の速度向上は実現できていない．$c = 3$ においては 4.1 節と同様の理由で $c = 2$ と実行時間が逆転している．さらに，オーバヘッドは最大で 27.4%である．これは，カーネル実行時間をデータ転送時間が大きく上回る場合には，カーネルの連続実行を前提としたスケジューリング手法では，適切な関数の実行順序を実現できない場合があることが原因だと考えられる．以上のことから，ストリーム処理は実アプリケーションにおいても有効といえる．ただし，本稿のスケジューリング手法は，カーネルを連続実行できないような高い時間比率を持つ場合には適さない．

5. ま と め

本稿では，複数の CUDA 互換 GPU を用いるストリーム処理において，関数の動的な並び替えを実現するミドルウェアを提案した．ミドルウェアは適切な順序で関数を実行することにより，オーバラップを実現する際の開発負担を軽減する．複数 GPU 環境においては，スレッド管理の自動化，GPU 間の負荷分散，およびデータ転送バスを共有する GPU 間での均等なデータ転送を実現する．

評価実験では，2 種類のダミープログラムに対してストリーム処理を適用した．その結果，データ転送をカーネル実行とオーバラップすることにより実行時間を短縮できた．また，複数 GPU 環境においては，カーネル実行時間が十分に長い場合に，GPU の台数に比例した線形の速度向上を実現した．ただし，メモリバンド幅の消費が多い場合にはオーバヘッドが増加することが分かった．実アプリケーションへの適用では，4 台の GPU を用いることにより，光学シミュレータにおいて実行時間を 4.41 倍の高速化を実現した．一方，配列アライメントにおいては 2.87 倍の速度向上に留まった．

今後の課題には，タスク間で関数の実行時間にばらつきがある場合，および実行時間に対してデータ転送時間が支配的になる場合の適切な負荷分散が挙げられる．

謝辞 本研究の一部は，科学研究費補助金基盤研究（A）（2024002）および大阪大学グローバル COE プログラム「予測医学基盤」の補助による．

参 考 文 献

1) Lindholm, E., Nickolls, J., Oberman, S. and Montrym, J.: NVIDIA Tesla: A Unified Graphics and Computing Architecture, *IEEE Micro*, Vol.28, No.2, pp.39–55 (2008).
2) GPGPU: General-Purpose Computation Using Graphics Hardware (2007). `http://www.gpgpu.org/`.
3) NVIDIA Corporation.: CUDA Programming Guide Version 2.3 (2009). `http://developer.nvidia.com/cuda/`.
4) Khailany, B., Dally, W.J., Kapasi, U.J., Mattson, P., Namkoong, J., Owens, J.D., Towles, B., Chang, A. and Rixner, S.: Imagine: Media Processing with Streams, *IEEE Micro*, Vol.21, No.2, pp.35–46 (2001).
5) Nakagawa, S., Ino, F. and Hagihara, K.: A Middleware for Efficient Stream Processing in CUDA, *Computer Science - Research and Development*, Vol.25, No.1/2, pp.41–49 (2010).
6) 成瀬彰，住元真司，久門耕一：GPGPU 上での流体アプリケーションの高速化手法〜1GPU で姫野ベンチマーク 60GFLOPS 超〜，情報処理学会研究報告，2008-HPC-117, pp.49–54 (2008).
7) Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., ha Lee, S. and Skadron, K.: Rodinia: A Benchmark Suite for Heterogeneous Computing, *IEEE International Symposium on Workload Characterization (IISWC-2009)*, pp.44–54 (2009).