Vol.2012-HPC-135 No.38
2012/8/3

マルチ GPU 環境におけるストリーム処理を高速化するタスクスケジューラ

中野 瑛仁[1]　伊野 文彦[1]　萩原 兼一[1]

概要：本稿では，マルチ GPU（Graphics Processing Unit）環境におけるストリーム処理の高速化を目的として，カーネル実行およびデータ転送のオーバラップ効率を向上するタスクスケジューラを提案する．提案スケジューラは，ストリームを構成する要素のデータサイズがその処理（タスク）の実行時間を決定すると仮定し，データサイズに関してタスクを実行時に整列する．これにより同様の実行時間を要するタスク同士をオーバラップし，オーバラップの効率を高める．また，オーバラップが不可能な最初のタスクの実行時間を短縮するために，スケジューラは一定数のタスクを蓄えたのちにタスクの実行を開始する．さらに，コールバック関数に基づくインタフェースは，GPU だけでなく CPU を含めたソフトウェアパイプラインを構築するために有用である．実験の結果，タスクを整列しない場合と比較して最大で 1.1 倍の速度向上を得た．

キーワード：ストリーム処理，GPGPU，CUDA，タスクスケジューリング

A Task Scheduler for Accelerating Streaming Processing on Multi-GPU Environments

Akihito Nakano[1]　Fumihiko Ino[1]　Kenichi Hagihara[1]

Abstract: In this paper, we present a task scheduler capable of realizing efficient overlap of kernel execution and data transfer, aiming at accelerating stream processing on multi-graphics processing unit (GPU) environments. Our scheduler assumes that the data size of an element in streams determines the execution time of its computation (i.e., task). According to this assumption, we sort tasks in terms of their data sizes during execution. This increases the overlap efficiency because tasks with similar execution times are overlapped each other. Furthermore, in order to reduce execution time of the first task, which cannot be overlapped with others, our scheduler delays task execution until it receives a certain number of tasks. Our callback function based interface is useful to construct software pipelines laying over not only GPUs but also CPUs. In experiments, we find that the speedup over an unsorted version reaches a factor of 1.1.

Keywords: Stream processing, GPGPU, CUDA, task scheduling

1. はじめに

GPU[1] はグラフィクス処理を高速化するための演算器である．GPU は SIMT（Single Instruction, Multiple Thread）型アーキテクチャを採用し，2048 個ものコアを用いる並列処理により高い浮動小数点演算性能を実現している [1].

この高い性能を汎用計算に活用するための開発環境として，NVIDIA 社は CUDA[2] を提供している．一般に CUDA プログラムは，CPU 上の計算，GPU 上の計算および CPU・GPU 間のデータ転送を処理する．CUDA は，各処理をパイプラインのステージとみなし，一連の処理をオーバラップ実行するための機構を提供している．この機構は，グラフィクス分野などに頻出するストリーム処理 [3]

[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

に対して有用である．

ここで，ストリームとは互いに類似する要素からなるデータ構造である．本稿では，各要素に施す一連の処理をタスクと呼び，要素のデータサイズがそのタスクの粒度（実行時間）を決めるものとする．例えば，タスクは CPU 上の計算，GPU 上の計算および CPU・GPU 間のデータ転送を内包する．一般に，ストリーム処理は各タスクがデータ依存に関して独立であることを前提にしている．したがって，パイプライン処理による高速化が望める．

CUDA では，タスクを CUDA ストリームに関連づけることによりストリーム処理を実装できる．ここで，CUDA ストリームとは逐次実行される一連の命令であり，各命令はカーネル起動やデータ転送などに相当する．異なる CUDA ストリームは並列処理されるため，複数の CUDA ストリームを用いることによりオーバラップを実現できる．しかし，実際には暗黙の同期 [2] が原因で，一方の処理が他方をブロックしうる．このブロックはパイプラインをストールさせ，オーバラップ効率を低下させてしまう．したがって，開発者はブロックを生じないような命令の発行順を探す必要があり，プログラム開発時の負担となる．

この負担を軽減するために，中川ら [4] はアウトオブオーダ型のスケジューラを提案している．この既存スケジューラは CUDA ストリームに対する命令の発行順を動的に決めることにより，ブロックを回避する．したがって，開発者は一連の要素を先頭から順に処理する単純なプログラムを記述すればよく，命令の発行順を考慮する必要がない．ただし，既存スケジューラはタスクをインオーダ実行する．したがって，実行時間の長いタスクを短いものとオーバラップする可能性があり，オーバラップ効率に関して改善の余地がある．

そこで本稿では，マルチ GPU 環境におけるストリーム処理の高速化を目的として，オーバラップ効率を向上するタスクスケジューラを提案する．提案するスケジューラは，タスクの実行時間がその要素のデータサイズに依存すると仮定し，データサイズに関してタスクを実行時に整列する．これにより同様の実行時間を要するタスク同士をオーバラップし，オーバラップ効率を高める．また，オーバラップが不可能な最初のタスクの実行時間を短縮するために，スケジューラは一定数のタスクを蓄えたのちにタスクの実行を開始する．さらに，GPU だけでなく CPU を含めたパイプラインを構築するために，コールバック関数に基づくインタフェースを提供する．提案するスケジューラは既存スケジューラ [4] に対する拡張として実装している．

以降では，まず 2 節で関連研究を紹介し，次に 3 節で予備知識としてストリーム処理についてまとめる．4 節で提案するタスクスケジューラについて述べる．5 節で評価実験の結果を示し，6 節で本稿をまとめる．

2. 関連研究

GPU 上のストリーム処理を支援するための記述体系は，これまでに多く提案されている．Sharp ら [5] は医用画像処理を対象とする記述体系を提案している．この記述体系は開発者の負担を軽減するという点において本研究と類似しているが，特定の応用に特化している．

中川ら [4] の既存スケジューラは，暗黙の同期が発生しないように，命令の発行順を動的に並び替える．しかし，タスクは先入れ先出し方式により CUDA ストリームに割り当てられている．提案スケジューラはタスクの並び替えにより既存スケジューラを拡張する．また，提案スケジューラはコールバック関数により，CUBLAS などの CUDA ライブラリを含めたパイプラインを構築できる．

Huynh ら [6] はマルチ GPU 環境においてストリーム処理を実現するためのフレームワークを提案している．このフレームワークは並列化のための処理の分割に主眼があり，オーバラップ効率を高めるものではない．Hagiescu ら [7] は実行効率の高いカーネルを自動生成するために，メモリ参照もしくは計算のいずれかに専念するスレッドの役割分担を提案している．本研究は GPU 上の計算を高速化するのではなく，データ転送を含めたパイプライン処理を高速化する．Hou ら [8] は GPU におけるストリーム処理のためのプログラミング言語を提案している．この言語は C 言語の拡張であり，逐次コードから単一 GPU 向けの CUDA コードを作成する．

Mikhail ら [9] は，GPU におけるストリーム処理のためのスケジューラを提案している．このスケジューラは，単一ではなく複数のアプリケーションのタスクを対象にする．また，本研究が最後のタスクが完了するまでの実行時間を短縮するのに対し，Mikhail らは各アプリケーションにおけるタスクごとの実行時間時間を短縮する．

本研究が対象とするスケジューリング問題はフローショップ問題の 1 種とみなせる．ハイブリッドフローショップ問題に対してタスクの実行時間を最小化する手法 [10] が提案されている．本研究の目的は最後のタスクが完了するまでの実行時間を短縮することである．

3. ストリーム処理

3.1 処理のモデル

本研究は，ストリームの各要素に対する処理が分岐しない単純なストリーム処理を扱う．入力ストリーム E_{in} が n 個の要素を持つとし，E_{in} における i 番目の要素を e_i と表す $(1 \leq i \leq n)$．このとき，$E_{in} = \{e_i \mid 1 \leq i \leq n\}$ となる．各要素 e_i $(1 \leq i \leq n)$ に対して独立に m 個の連続した処理 $f_1, f_2, ..., f_m$ を施す場合，その出力 $E_{out} = \{e'_i \mid 1 \leq i \leq n\}$ は式 (1) で与えられる．

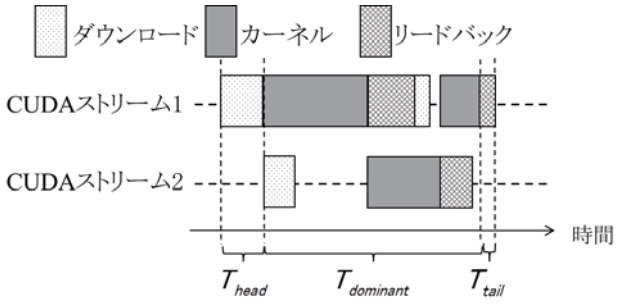


図 1 データ転送およびカーネル実行がオーバラップされている様子

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

ここで，$\circ$ は関数の合成演算を表す．

式 (1) において，e_i から e'_i を出力する処理をタスク t_i とする．一般に，$i \neq j$ のとき t_i および t_j の間にデータ依存はなく，任意の順でタスクを実行できる．また，本研究においては，e_i および e_j のデータサイズ $|e_i|$ および $|e_j|$ に関して $|e_i| < |e_j|$ が成り立つとき，t_i よりも t_j の実行時間の方が長いと仮定する．すなわち，t_i の実行時間を $|t_i|$ と表記すると，式 (2) が成り立つ．

$$\forall i, j \in \{1, 2, \ldots, n\}\ (|e_i| < |e_j|) \Rightarrow (|t_i| < |t_j|) \tag{2}$$

3.2 実行時間の下界

単一 GPU における実行時間の評価に用いるための下界 T_{opt} を示す．簡単のために，時刻 0 においてタスク集合 $T = \{t_1, t_2, \ldots, t_n\}$ が発行されているとする．また，t_i $(1 \leq i \leq n)$ に対して f_1 としてダウンロード，f_2 としてカーネル実行，f_3 としてリードバックを施すタスクを対象とする．既述のように，CUDA においてデータ転送およびカーネル実行はオーバラップ可能である．したがって，実行のガントチャートは**図 1** のようになる．

T_{head} および T_{tail} をそれぞれ最初のダウンロード命令 I_{head} および最後のリードバック命令 I_{tail} の実行時間とする．t_i におけるダウンロード，カーネル実行およびリードバックの実行時間をそれぞれ $T_D(t_i)$，$T_K(t_i)$ および $T_R(t_i)$ と表記すると，T_{head} および T_{tail} の下界はそれぞれ式 (3) および式 (4) で与えられる．

$$T_{head} = \min_{1 \leq i \leq n} T_D(t_i) \tag{3}$$

$$T_{tail} = \min_{1 \leq i \leq n} T_R(t_i) \tag{4}$$

ただし，$n > 1$ のとき I_{head} および I_{tail} に対して同じタスクが選ばれることはない．

すべてのタスクが発行する命令のうち，I_{head} および I_{tail} のどちらにも含まれない命令の集合を I_{dom} とし，その実行時間全体を T_{dom} とする．T_{dom} が下界に近づくのは，性能ボトルネックとなる命令が絶え間なく実行されるときである．I_{dom} におけるデータ転送時間の合計は $\sum_{i=1}^{n}(T_D(t_i) + T_R(t_i)) - T_{head} - T_{tail}$，カーネル実行時間の合計は $\sum_{i=1}^{n} T_K(t_i)$ である．カーネル命令およびデータ転送命令のうち，実行時間の合計が大きい方が性能ボトルネックであるため，T_{dom} の下界は式 (5) で与えられる．

$$T_{dom} = \max\left(\sum_{i=1}^{n}(T_D(t_i) + T_R(t_i)) - T_{head} - T_{tail}, \sum_{i=1}^{n} T_K(t_i)\right) \tag{5}$$

以上から，実行時間の下界 T_{opt} は

$$\begin{aligned} T_{opt} &= T_{head} + T_{dom} + T_{tail} \\ &= \max\left(\sum_{i=1}^{n}(T_D(t_i) + T_R(t_i)), \min_{1 \leq i \leq n} T_D(t_i) + \sum_{i=1}^{n} T_K(t_i) + \min_{1 \leq i \leq n} T_R(t_i)\right) \end{aligned} \tag{6}$$

と表される．

4. 提案するタスクスケジューラ

提案するスケジューラは，既存のスケジューラに対して，以下の拡張を施している．

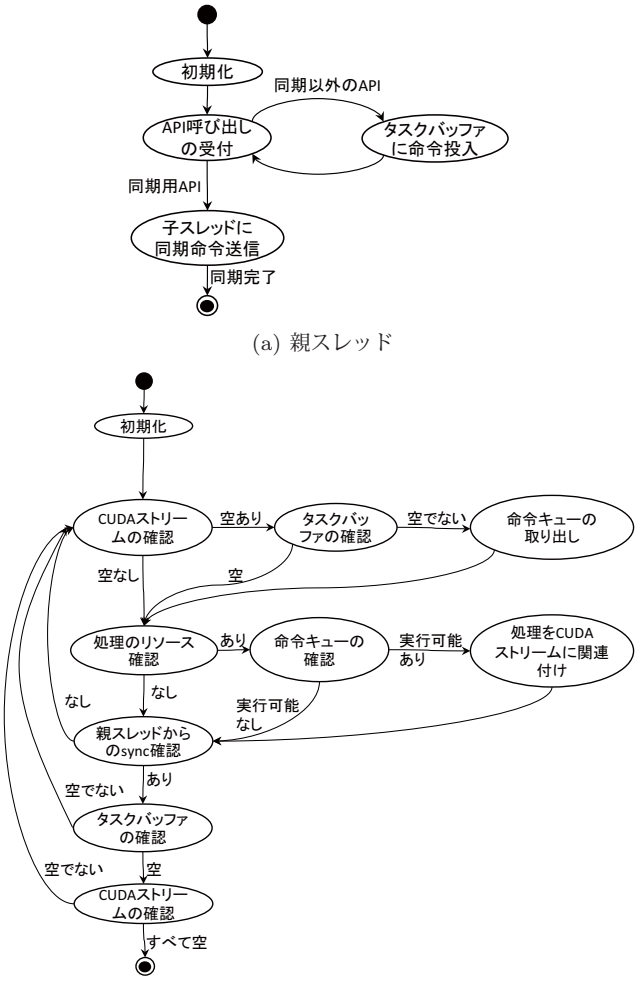
- タスクのスケジューリング機構．タスクの実行順を入力要素のデータサイズに関して昇順，降順または山型に並べ替える．
- タスクの実行開始を遅延させる機構．スケジューラにタスクを蓄え，並び替えるタスクの範囲を広げる．
- コールバック関数．GPU だけでなく CPU を含めたソフトウェアパイプラインを実現する．

以降では，これら 3 点の拡張について説明する．

4.1 スケジューラの設計

既存スケジューラと同様に，提案するスケジューラはアプリケーションレベルでタスクの並び替えを実現する．開発者はソースコード内にある CUDA の関数呼び出しをスケジューラが提供するものに置換すれば，動的な並び替えを実現できる．一方，スケジューラは親スレッドとして動作し，API 呼び出しにより発行された命令をタスクバッファに蓄える．そして，これらの命令の実行は，GPU ごとに用意する子スレッドが担当する．また，タスクバッファはタスクのスケジューリング機能を実現するために，命令をタスクごとに管理する．親スレッドがタスクバッファにタスクを追加するたびに，もしくは子スレッドがタスクを取り出すたびに，後述する基準に基づき整列を施す．

命令をタスクごとに管理するために，タスクバッファは各タスクに対して命令キューを持ち，発行済みの命令をそれぞれの命令キューに追加する．子スレッドがすべてのタスクを取り出し命令キューが空になった場合，その命令キューを整列の対象から除外する．ただし，タスクバッファは命令キューのポインタを保持し，命令を追加する．



(a) 親スレッド

(b) 子スレッド

図 2 スケジューラを構成するスレッドの状態遷移図

これにより，命令キューにすべての命令が揃う前に，命令の実行を開始できる．

図 2 にスケジューラの親スレッドおよび子スレッドの動作を示す．親スレッドは，ホストコードがスケジューラの関数を呼び出すまで待機する．タスクの命令を渡す関数が呼び出された場合は，その命令をタスクバッファに追加する．同期関数が呼ばれた場合，子スレッドの処理が完了するまで待機してから，制御をホストコード側に戻す．

一方，子スレッドは，タスクの取り出しおよび命令の実行を担当する．まず，自身が監視する GPU の CUDA ストリームが空である場合，タスクバッファからタスクの命令キューを取り出す．次に，オーバラップ可能な命令があるかどうかを確かめるために，データ転送およびカーネル実行のリソースが余っているかどうかを確かめる．そして，命令キューの先頭にオーバラップ可能な命令があれば，それを実行する．また，親スレッドから同期命令が送られた場合，タスクバッファおよび CUDA ストリームが空になり次第，処理を終了する．

4.2 スケジューリング方針

タスクバッファに拡張を施すことにより，タスクの整列を実現する．整列の方針として，既述のように，スケジューラはタスクを要素のデータサイズに関して整列する．整列の方針は，昇順，降順，山型の 3 つである．昇順および降順の整列は，T_{dom} を短縮することにより全体の実行時間を短縮するスケジューリング方針である．一方，山型の整列は，T_{dom} だけでなく T_{head} および T_{tail} も短縮する．

T_{dom} を短縮するためには，オーバラップにより隠蔽する実行時間を増加させる必要がある．つまり，同様の実行時間を有するタスク同士をオーバラップさせる必要がある．式 (2) の仮定より，タスクをデータサイズに関して昇順または降順ソートすることにより，T_{dom} を短縮できる．

ただし，タスクの実行順を入力データサイズに関して降順に並び替える場合，最初に実行するタスクとして，実行時間が最も長いタスクが選択される．したがって，T_{head} が増加する．逆に，昇順に並び替えた場合，T_{tail} が増加する．これらはオーバラップ効率を低下させる．

これらを解決することを目的として，山型の整列は T_{head} および T_{tail} を短縮する．T_{head} を短縮するためには，最初に実行するタスクとして，実行時間の短いものを選択すればよい．同様に，T_{tail} を短縮するためには，最後に実行するタスクとして，実行時間の短いものを選択すればよい．提案手法においては，実行時間が最も短いタスクを最後に実行し，2 番目に短いタスクを最初に実行する．

これらの選択のもとで，T_{dom} を短縮する．上記で述べたように，T_{dom} を短縮するためには，実行順の前後においてタスク間の実行時間の差を小さくすればよい．最初と最後にデータサイズの小さいタスクを配置するため，タスクの並び順は最初に昇順，のちに降順となるように並び替えるのがよい．

データサイズに関してが k 番目に小さいタスクを $AT(k) \in T$ とする．最初に実行するタスクは $AT(2)$ であるため，昇順に実行するタスクは $AT(2), AT(4), AT(6), ...$ というように，偶数番目にデータサイズの小さいタスクを選択する．一方，降順に実行するタスクの順番は，奇数番目のタスクをデータサイズに関して降順に $..., AT(5), AT(3), AT(1)$ と実行する．

提案するスケジューラはオンラインアルゴリズムとして動作する．したがって，タスクの並び替えは，スケジューラに対して新たなタスクが投入されるたびに実行する必要がある．山型の並び替えにおいては，タスクの並び順がデータサイズだけでなくタスク数にも依存するため，オンラインアルゴリズムにおいては正しく動作しない．ただし，E_{in} の要素を生成する間隔が $f_1, ..., f_m$ の実行時間よりも十分に短い場合，2 番目以降に実行するタスクに関しては，オンラインアルゴリズムにおいても山型に近い並び順を実現できる．

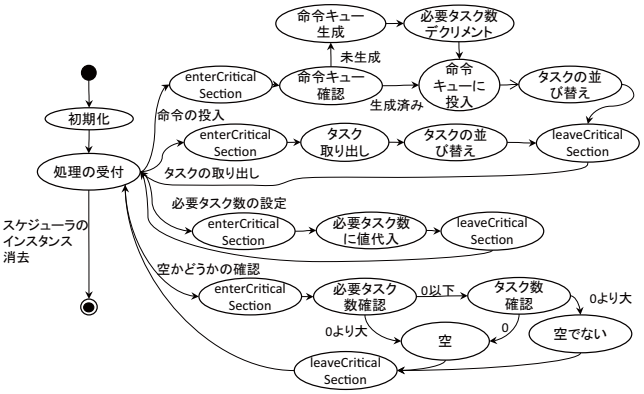


図 3 タスクバッファの動作図

表 1 実験に用いた計算機の仕様

項目	内容
CPU	Intel Xenon X5670
GPU	Geforce GTX 580 × 4
OS	CentOS 5.4
CUDA のバージョン	4.2
CUDA のドライババージョン	295.41

4.3 タスク実行の遅延機構

図 2(b) に示すように，子スレッドはタスクバッファが空でないときに限り，タスクを取り出す．したがって，タスクバッファの中身を空に見せかけることにより，バッファにタスクを蓄える．

タスクバッファの動作を**図 3** に示す．1 節で述べたように，スケジューラが一定数のタスクを受け取るまでタスク実行を遅延させる．また，タスク実行を開始するために必要なタスク数を指定する関数として，待ち関数を追加する．

タスクバッファの状態が「処理の受付」のとき，待ち関数が呼び出されると，動作「必要タスク数の設定」に遷移し，タスクの実行開始に必要なタスク数を記憶する．タスクバッファにタスクを投入するときは，動作「タスクの投入」に遷移し，必要タスク数をデクリメントする．子スレッドがタスクバッファの中身を確認したときは，必要タスク数の値を確認する．値が 0 より大きい場合，タスクを蓄えるために空を返す．一方，値が 0 以下の場合，タスクバッファが実際に空かどうかを確認し，その結果を返す．

5. 評価実験

まず，スケジューラを用いたストリーム処理のオーバヘッドを評価するために，式 (6) の下界と比較する．次に，スケジューラを実際のアプリケーションに適用した場合の性能を評価するために，アミノ酸配列のアライメントを処理するアプリケーションにスケジューラを適用する．なお，実験環境は**表 1** のとおりである．

5.1 オーバヘッドの評価

スケジューラの性能を測定するために，次のような CUDA アプリケーションに対してスケジューラを適用した．まず，整数型配列を GPU 上のグローバルメモリにダウンロードする．次に，ダウンロードした配列に対してシフト演算を一定回数施す．最後に，シフト演算の結果をリードバックする．このアプリケーションは，タスクの各ステージにおける実行時間を容易に調整できるため，スケジューラの詳細な性能解析に適している．

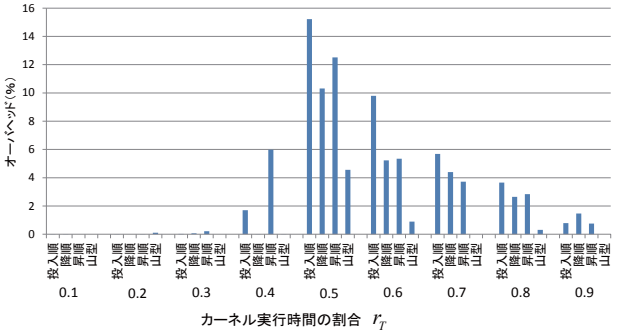


図 4 オーバヘッドの調査

図 4 にカーネル実行時間の割合 r_T を変えたときのオーバヘッド σ を示す．ここで，ダウンロード時間を T_D，カーネル実行時間を T_K，リードバック時間を T_R とするとき，$r_T = T_K/(T_D + T_K + T_R)$ と表せる．r_T の値が 0.1 に近いほどデータ転送が支配的となり，0.9 に近づくほどカーネル実行が支配的となる．また，オーバヘッド σ は，実際の実行時間 T に対して $\sigma = ((T - T_{opt})/T_{opt}) \times 100$ により与えられる．σ の値が小さいほど，スケジューリングの性能が良いことを表す．値が 0 のとき，タスクの並び順は最適である．

$r \leq 0.4$ のとき，ダウンロード時間が支配的となる．$0.1 \leq r_T \leq 0.3$ のとき，すべてのスケジューリング方針においてオーバヘッドは 1%以下に抑えられる．スケジューラによる整列をしない場合においてもオーバヘッドが小さい理由は，データ転送時間がカーネル実行時間に対して十分大きく，整列を施さずとも I_{dom} においてすべてのカーネル実行時間を隠蔽できたためである．$r_T = 0.4$ のとき，昇順のオーバヘッドが，他のスケジューリング方針と比べて，大きい．昇順においては，最初に小さいタスクを実行するため，部分的にカーネル実行時間を隠蔽できなかったことが原因である．

$r \geq 0.5$ のとき，カーネル実行時間が支配的となる．まず，$r_T = 0.5$ のときに山型を除くスケジューリング方針において，オーバヘッドが 10%を超える．これは，カーネル実行時間およびデータ通信時間が同等であり，I_{dom} においてデータ通信時間を隠蔽しきれなかったためである．このとき，スケジューリングによる効果が最も現れている．なお，$r_T = 0.5$ のときの山型のオーバヘッドは，他のスケジューリング方針の半分程度である．これは，$T_{head} + T_{tail}$ の値が他のスケジューリング方針に比べ小さいためであ

る．$0.5 < r_T \leq 0.9$ のとき，が 0.9 に近づくにつれ，オーバヘッドが小さくなる．これは，カーネル実行時間がデータ転送時間に比べて十分に大きくなり，データ転送時間の隠蔽に成功しやすくなったためである．ただし，式 (6) より，性能ボトルネックがカーネル実行であるときは T_{head} および T_{tail} が実行時間を増加させることがある．したがって，r_T が 0.9 のときも，山型以外はオーバヘッドが 1% 程度存在する．

5.2 アミノ酸配列のアライメントにおける評価

塩基配列の局所アライメントを処理するアプリケーション [11] は，入力として塩基配列を表すクエリ配列およびデータベース配列をディスクから読み込む．次に，読み込んだデータベース配列を Smith-Waterman アルゴリズム [12] を用いてクエリ配列と比較することにより局所アライメントを算出する．最後に，結果をディスクに書きだす．

実験に用いるクエリ配列として，配列長 63 および 511 の塩基配列を用いた．また，データベース配列は配列長 2〜35,213 の塩基配列を 536,029 個ほど有する．プログラムは，以下の処理を繰り返す．まず，データベースから配列を 8,192 個読み出し，GPU にダウンロードする．次に，アライメントを計算するためにカーネルを実行し，その後結果をリードバックする．最後に，リードバックした結果をディスクに書き込む．なお，待ち関数を用いる場合，タスクを 30 個蓄えてから処理を開始する．

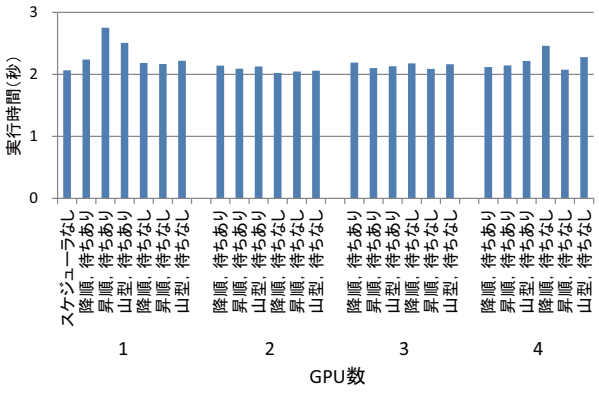
図 5 に，クエリ配列長が 63 および 511 のときの実行時間を示す．ここで，実行時間は試行 20 回の平均値である．クエリ配列長が短いとき，性能ボトルネックはディスクアクセスである．したがって，GPU 数が増加しても，実行時間は短縮しない．なお，実行時間にばらつきがあるのは，ディスクアクセスの時間にばらつきがあるためである．実行時間の最小値は，すべて 2 秒であった．

一方，クエリ配列長が 511 のとき，カーネル実行の占める割合が増加したため，GPU 数が増加するにつれ，実行時間が減少している．しかし，スケジューリング方針の違いによる実行時間の差は僅かである．これは，タスクの粒度に差が小さいことが原因である．

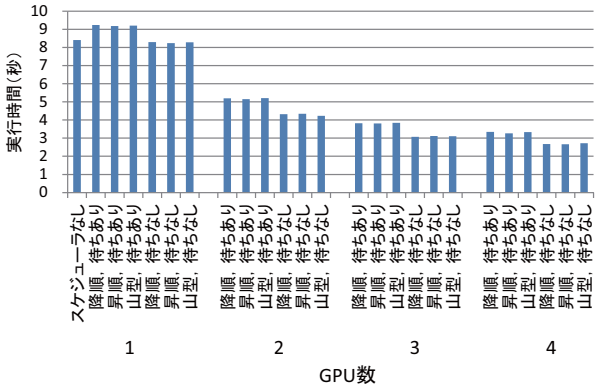
待ち関数の有無により実行時間を比較すると，なしの方が高速となっている．このアプリケーションはタスクの前処理としてディスクの読み込みをするため，待ち関数によるオーバヘッドが発生する．また，このアプリケーションはタスクスケジューリングの恩恵が少ないため，オーバヘッドを隠蔽できない．したがって，待ち関数ありのときに実行時間が増加してしまう．

6. まとめ

本稿では，マルチ GPU 環境におけるストリーム処理の高速化を目的として，タスクの実行順を動的に並び替えるタスクスケジューラを提案した．提案するスケジューラは，ストリームを構成する要素のデータサイズがそのタスクの実行時間を決定すると仮定し，データサイズに関してタスクを実行時に整列する．これによりタスク粒度に関して不均一なタスクに対し，高いオーバラップ効率を実現する．また，オーバラップが不可能な最初のタスクの実行時間を短縮するために，一定数のタスクを蓄えたのちにタスクの実行を開始する．さらに，GPU だけでなく CPU を含めたソフトウェアパイプラインを構築するために，コールバック関数に基づくインタフェースを提供する．



(a) クエリ配列長 63



(b) クエリ配列長 511

図 5 アライメントの実験結果

評価実験では，不均一なタスクに対して全体の実行時間を短縮でき, タスクを整列しない場合と比較して最大で 1.1 倍の速度向上を得た．また，タスクの前処理がない場合，待ち関数のオーバヘッドよりもスケジューリングによる性能改善の方が大きいことがわかった．

今後の課題としては，タスク粒度の差が小さい場合やタスクの前処理があるアプリケーションに対する性能改善が挙げられる．

参考文献

[1] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110, may 2012. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.

pdf.

[2] NVIDIA Corporation. CUDA Programming Guide Version 4.2, may 2012. http://developer.nvidia.com/cuda/.

[3] Brucek Khailany, Wiliam J. Dally, Ujval J. Kapasi, Peter Mattson, Jinyung Namkoong, Jhon D. Owens, Brian Towles, Andrew Chang, and Scott Rixner. Imagine: Media processing with streams. *IEEE Micro*, Vol. 21, No. 2, pp. 35–46, March 2001.

[4] 中川進太, 伊野文彦, 萩原兼一. 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. 情報処理学会研究報告, Vol. 2010, No. 19, pp. 1–8, 2010-07-27.

[5] Gregory Sharp, Nagarajan. Kandasamy, Harman. Singh, and Michael. Folkert. GPU-based streaming architectures for fast cone-beam CT image reconstruction and demons deformable registration. *Physics in Medicine and Biology*, Vol. 52, pp. 5771–5783, September 2007.

[6] Huynh Phung Huynh, Andrei Hagiescu, Weng-Fai Wong, and Rick Siow Mong Goh. Scalable framework for mapping streaming applications onto multi-gpu systems. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pp. 1–10, New York, NY, USA, 2012. ACM.

[7] Andrei Hagiescu, Huynh Phung Huynh, Weng Fai Wong, and Rick Slow Mong Goh. Automated architecture-aware mapping of streaming applications onto GPUs. In *Proc. 25th IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS'11)*, pp. 467–478, May 2011.

[8] Qiming Hou, Kun Zhou, and Baining Guo. BSGP: Bulk-synchronous GPU programming. *ACM Trans. Graphics*, Vol. 27, No. 3, August 2008. Article 19.

[9] Mikhail. Bautin, Ashok. Dwarakinath, and Tzi-cker Chiueh. Graphic engine resource management. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, Vol. 6818 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, January 2008.

[10] Valerie Botta-Genoulaz. Hybrid flow shop scheduling with precedence constraints and time lags to minimize maximum lateness. *International Journal of Production Economics*, Vol. 64, No. 1-3, pp. 101–111, 2000.

[11] Yuma Munekawa, Fumihiko Ino, and Kenichi Hagihara. Accelerating Smith-Waterman algorithm for biological database search on CUDA-compatible GPUs. *IEICE Trans. Information and Systems*, Vol. E93-D, No. 6, pp. 1479–1488, June 2010.

[12] Huynh Phung Huynh, Andrei Hagiescu, Weng-Fai Wong, and Rick Slow Mong Goh. Scalable framework for mapping streaming applications onto multi-GPU systems. In *Proc. 17th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP'12)*, pp. 1–10, February 2012.