Vol.2011-HPC-129 No.22
2011/3/16

GPGPUにおけるデータ転送とカーネル実行のヒューリスティックスケジューリング

本間咲来[†1]　須田礼仁[†1,†2]

GPGPU 用の統合開発環境である CUDA では，ストリームと呼ばれる機構を用いることで，依存関係がないデータ転送とカーネルをオーバーラップして実行することができる．これにより，処理性能のボトルネックとなりがちな，主記憶－ GPU メモリ間のデータ転送を隠蔽することが可能である．しかし，性能を最大限に引き出すには，データの分割やストリームの割り当て，API を呼び出す順番などを最適化しなければならない．そこで本研究では，それらの要因がどのように所要時間に影響するか調査を行った．さらに，計測を行った結果からモデルを構築することにより，最適に近いスケジュールを構成するヒューリスティックを提案する．評価実験では，行列積にスケジューリングを適用し，ストリームを適用せずに実行した場合と比較し，検証を行った．

HEURISTIC SCHEDULING FOR DATA TRANSFER AND KERNEL EXECUTION IN GPGPU

Saki Homma[†1] and Reiji Suda[†1,†2]

In CUDA, which is integrated development environment for GPGPU, data transfer and kernel execution without dependence can be executed in parallel by using the mechanism called stream. However, it is necessary to optimize such factors as division of data, allocation of streams, and order of API calls, in order to achieve the maximum performance of hardware. In this study, we investigate how these factors influenced the execution time. We propose a near-optimal heuristic to configure the schedule by constructing the model by the result of measurement. In the experiment, we applied the schedule to matrix multiplication program, and compared with the case executed without using stream.

1. はじめに

近年 GPU はその高い演算性能やコストパフォーマンスの良さから，演算資源として注目を集めている． GPU をグラフィックス以外の処理に用いる技術は GPGPU と呼ばれ，専用の開発環境も提供されるようになり，活発にアプリケーションの研究が行われている．本研究では NVIDIA から提供されている CUDA という IDE を用いて開発を行った．

GPU を用いて計算を行う場合，まず主記憶から GPU 側のメモリにデータを転送し，GPU コード（カーネル）を実行し，計算した結果を再び主記憶に戻さなければならない．これら主記憶－ GPU メモリ間のデータ転送は，処理性能のボトルネックとなりがちだが，ストリームと呼ばれる機構を用いることで， CUDA では依存関係がないデータ転送とカーネルをオーバーラップして実行することが可能となっている．

そこで本論文では，データ転送とカーネル実行を複数のタスクに分割し，それらをオーバーラップして実行させるヒューリスティックを提案する．本手法を用いることで，転送ルートとプロセッサの資源を有効に使い，プログラムの所要時間を短縮することができる．データ転送時間とカーネル実行時間のどちらか短い方を，もう片方の時間に隠蔽することが目標である．本手法ではブロック間でカーネルにかかる時間は一定ではないと仮定している．しかし，転送に必要な時間は一定とは限らないと仮定した．

研究ではまず，多くの GPU に対応するために，複数の環境が異なるプラットフォームで，各 API の呼び出し順序が GPU の種類や世代，ドライバのバージョンがどのように所要時間に影響を及ぼすか調査を行った．その後，上記の結果を元に提案するスケジューラーを CUDA で実装した．

本論文の構成を以下に述べる．まず 2 章では，CUDA のストリームを使った処理の流れについて説明する．次に 3 章では，GPU やドライバのバージョンが異なる複数の環境で，API の呼び出し順序を変えて測定した結果を紹介する．4 章では，スケジュールを実現するアルゴリズムについて述べる．5 章では，本研究の手法を行列積に適用し，所要時間の測定を行った結果を示す．また，データ転送とカーネル実行を逐次に処理した場合と比較し，

†1 東京大学大学院情報理工学系研究科コンピュータ科学専攻
Department of Computer Science, Graduate School of Information Science and Technology, The University of Tokyo
†2 JST CREST
CREST, JST

結果を考察する．最後に 6 章で将来研究とともに，結論を述べる．

1.1 関 連 研 究

本研究で扱うトピックと関連のある論文を以下に紹介する．

須田ら[1] は CPU − GPU 間のデータ転送の遅延を隠蔽する最適スケジュールを提案している．提案されたアルゴリズムでは，タスクは任意のサイズで区切られ，オーバーラップして実行される．全てのタスクのオーバーヘッドは等しく，データ転送をカーネル実行に掛かる時間はデータサイズにのみ依存すると仮定している．我々の手法では，オーバーヘッドや所要時間はデータサイズに依存するとは限らないとしており，仮定が異なる．

また中川ら[2),3)] は効率的なストリーム処理のために，カーネルとデータ転送を out-of-order 実行させるミドルウェアを提案している．CUDA API の代わりに提案したミドルウェアを用いることで，タスクをオーバーラップして実行できる．彼らのミドルウェアはタスクサイズを開発者が決めなければならないが，我々の方法ではタスクのサイズはスケジューラーが決定する．

2. CUDA におけるストリーム処理

CUDA ではストリームと呼ばれる，転送とカーネル実行を非同期で行うための機構が用意されている．同じストリームに登録されたタスクは呼び出した順番通りに実行されるが，異なるストリームに登録されたタスクの順序は入れ替わる可能性があるという特徴がある．CUDA にはデータ転送用とカーネル用の 2 つのキューがあり，現在実行しているタスクとストリームに依存関係がないものは，先に現在のタスクと同時に実行される．

以後メインメモリから GPU メモリへのデータの転送をセンド，GPU メモリからメインメモリへの転送をレシーブと呼ぶ．また，ストリーム s に登録されたセンドタスクを「センド s」と表記する．カーネルやレシーブも同様である．

2 つのストリームを使い，タスクを重ねた例を**図 1**(a) に示す．(a) のように処理されるには，例えばセンド 1，カーネル 1，センド 2，カーネル 2，センド 1，カーネル 1 という順で API を呼ぶ必要がある．また，API の呼び出し順序が，センド 1，センド 2，センド 1，カーネル 1，カーネル 2，カーネル 1 という場合に図 1 (b) のようになる．これは，ストリーム 1 のカーネルがセンド 1 の後に実行されなければならないためである．同じ種類のタスク間や，同じストリームのタスク間では実行順序は変わらないため，タスクが重なることなく処理されている．

このように，データ転送とカーネルを並列に行うには，タスクに異なるストリームを割り当て，適切な順序で API を呼び出さなくてはならない．

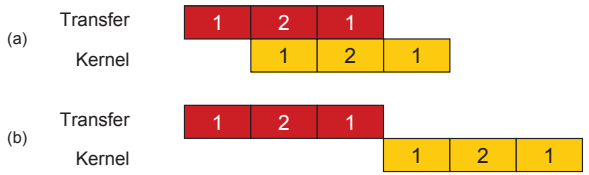


図 1 2 つのストリームを使いタスクをオーバーラップさせる
Fig. 1 Overlapping tasks using two streams

3. 環境による API の処理の違い

このセクションでは，API の呼び出し順番がどのように処理に影響を与えるか実験を行った結果を紹介する．

2 節で述べた様に CUDA では異なるストリームに登録されたデータ転送とカーネルは並列に処理される．しかし，それ以外にもオーバーラップさせて処理時間を短くするための条件があることが分かった．次にタスクが予測通りに処理されなかった 2 つの例を紹介する．

3.1 異なるストリームに登録された複数のカーネル呼び出し

ここでは，API の呼び出し順によって，タスクが並列に処理されず，逐次に実行されるケースを紹介する．

実験では，以下の 4 パターンの順序で API を呼んだ．色はグラフ内の線の色を表す．このパターン 3 と 4 は**図 2**(a) のように処理されるはずである．カーネル 2 のサイズを変えて測定した結果を**図 3** に示す．

パターン 1 センド 1 −カーネル 2　黄
パターン 2 カーネル 2 −センド 1　黒
パターン 3 カーネル 1 −センド 2 −カーネル 1　ピンク
パターン 4 カーネル 1 −カーネル 2 −センド 1　水色

図 3(a)，**図 3(b)** の Tesla C1060 と GeForce GTX 295 はパターン 3 と 4 で差はない．どちらも，センドとカーネルはオーバーラップして処理されていることが分かる．しかし，**図 3(c)** の Tesla C2050 では，パターン 3 はオーバーラップして実行されているにも関わらず，パターン 4 は図 2(b) に示すように逐次的に処理されている．さらに，**図 3(d)** の ION では，パターン 3 の場合も逐次に処理されている．

Vol.2011-HPC-129 No.22
2011/3/16

図 2 API の呼び出し順序により処理順序が変わる
Fig. 2 Processing flow changes depending on order of calling API

Tesla 2050 の他に，あるストリームのカーネルと転送の間に，他のストリームのカーネルがあると，その転送はカーネルと並行には実行されない GPU は他にも GeForce GTX 460, 470, 480, GeForce GTX 580 があった．しかしこれらはどの GPU もパターン 1 と 2 はオーバーラップして処理されていた．これより，上記のタイプの GPU でも，カーネルとセンド API を呼ぶたびに同期 API を挿入し，処理の終了を待機すれば，オーバーラップは実現できることが分かった．また，図 3(d) のように，一切のタスクを並列に実行できない GPU は，ION の他に GeForce 8400 GS と GeForce 9300M GS があった．これらの 3 つの GPU は CUDA バージョンが 3.1 で一致しているが，GeForce 8400 GS とプラットフォームが同一である Tesla C1060 はタスクを並行して実行できた．また，compute capability も共通しているが，おなじく compute capability が 1.1 である GeForce 8800M GTX はオーバーラップして処理可能であった．よって，CUDA バージョンや compute capability によってオーバーラップできるかどうかを判断することはできない．

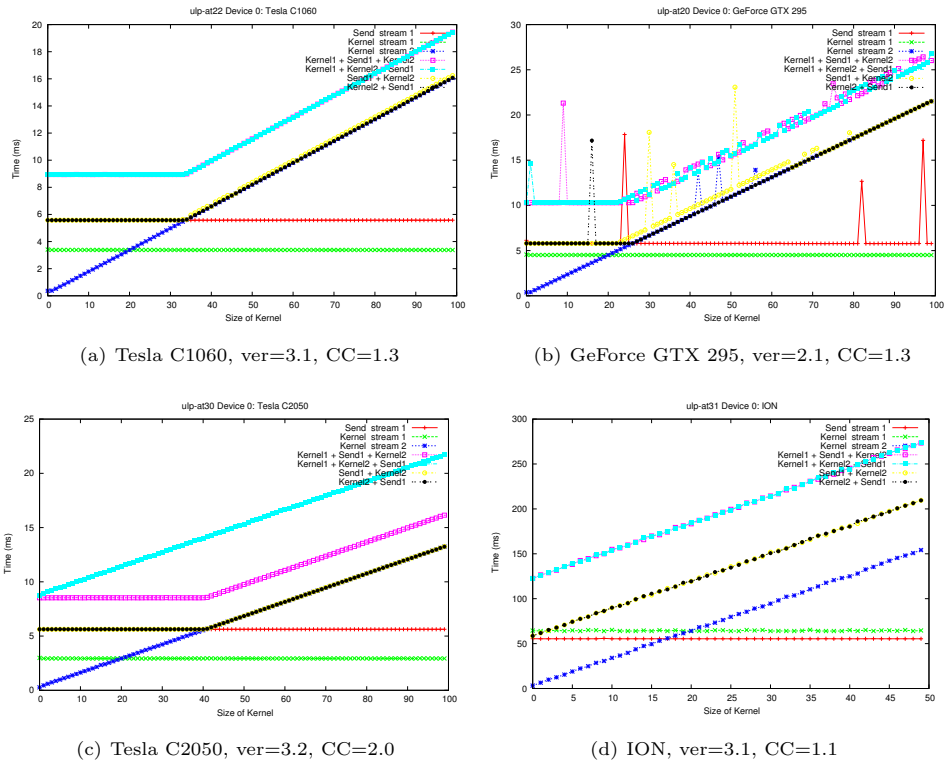
この結果から，あるストリームでカーネルと転送を行うとき，その間に別のストリームのカーネルを入れるべきではないが，同期 API を挿入することで，問題を回避できる GPU があること，また，ストリームを使ってもオーバーラップを実現できない GPU があることが分かった．

3.2 転送中のカーネル呼び出し

ここでは，API の組み合わせによって，転送スループットがおちるケースについて紹介する．

実験では，以下の 6 パターンの順序で API を呼んだ．

パターン 1 センド 1 －カーネル 2 －カーネル 0　ピンク
パターン 2 センド 1 －カーネル 2 －カーネル 1　水色
パターン 3 センド 1 －カーネル 2 －カーネル 2　黄色
パターン 4 カーネル 1 －センド 2 －カーネル 0　黒
パターン 5 カーネル 1 －センド 2 －カーネル 1　オレンジ



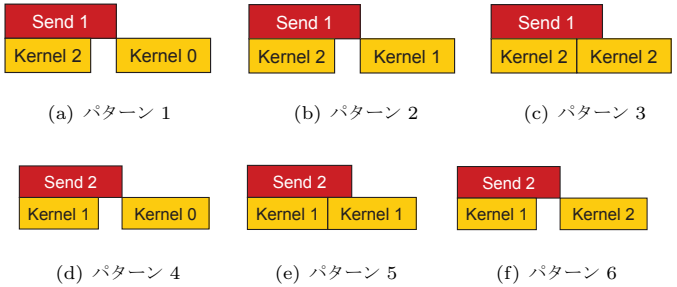
(a) Tesla C1060, ver=3.1, CC=1.3
(b) GeForce GTX 295, ver=2.1, CC=1.3
(c) Tesla C2050, ver=3.2, CC=2.0
(d) ION, ver=3.1, CC=1.1

図 3 異なるストリームに登録された 2 つのカーネル
Fig. 3 2 Kernels with different stream

パターン 6 カーネル 1 －センド 2 －カーネル 2　灰色

色はグラフ内の線の色を表す．これらのパターンは**図??**のように処理されることが予想される．2 番目のカーネルのサイズを変えて測定した結果を**図 5** に示す．

図 5(c) によると，Tesla 2050 は 6 つのパターン全てを同じように処理していることが分かる．しかし，**図 5(a)** と**図 5(b)** の Tesla C1060 と GeForce GTX 295 では少し異なっている．Tesla C1060 は，センド 1 とカーネル 2 のサイズが等しいとき最も小さい処理時間となり，カーネル 2 がセンド 1 より短いときはより時間が掛かっている．これはセンドのスループットが落ちていることを意味する．GeForce GTX 295 では，Tesla C1060 のマ



(a) パターン 1　(b) パターン 2　(c) パターン 3

(d) パターン 4　(e) パターン 5　(f) パターン 6

図 4 転送中のカーネル呼び出し
Fig. 4 Kernel invocation while Transfer

シンほど顕著ではないが，その傾向が見られる．

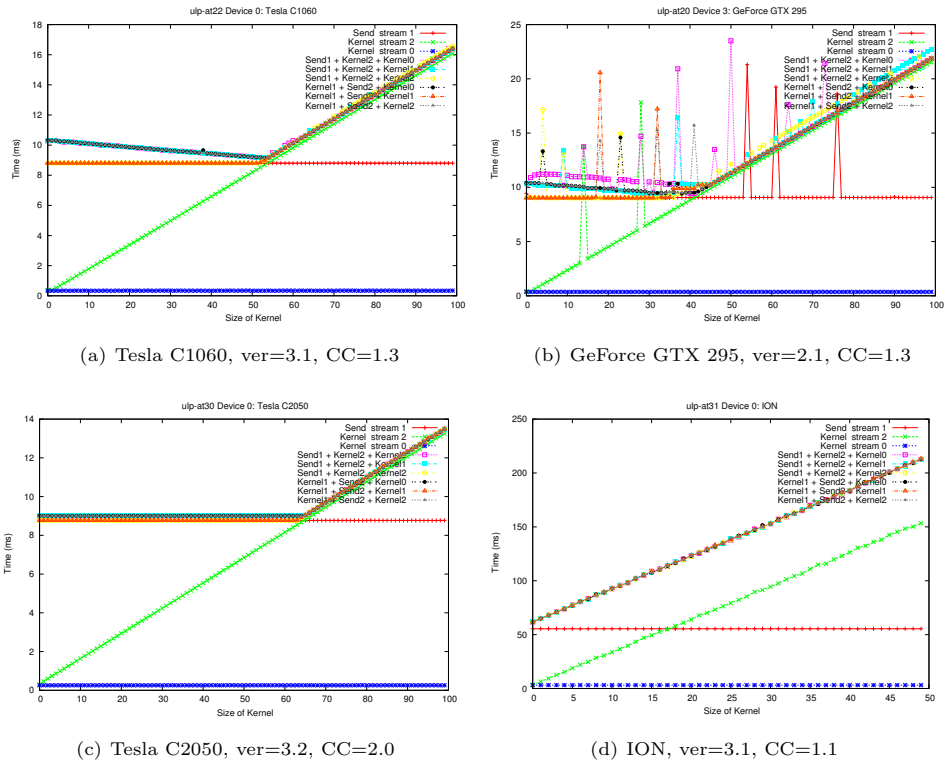
Tesla C1060 はパターン 3 と 5 を除く 4 つのパターンは同様に処理されている．GeForce GTX 295 はパターン 1 が比較的多く時間が掛かっているが，他は Tesla 2050 と同様である．

カーネルとセンドだけの組み合わせの場合，並列に処理されることは図 3(a) と図 3(b) より分かっている．パターン 1 と 4 では 2 番目のカーネルはセンドが終了後に実行される．これはストリーム 0 に登録されたタスクは，呼び出し済みの全てのタスクの終了後に実行されるからである．また，パターン 2 と 6 も，センドと 2 番目のカーネルのストリームが同じであるため，センドの後にカーネルは実行される．パターン 3 と 5 に共通するのは，センドの終了を待つカーネルが存在しないということである．Tesla C1060 と GeForce GTX 295 では，パターン 1，2，4 や 6 のように並列に処理されるカーネルが存在し，さらに転送の終了を待つカーネルが存在すると，転送のみ実行する部分のスループットが落ちるようである．そのため，カーネルとセンドだけの組み合わせや，センドの終了を待つカーネルがないパターン 3 と 5 はオーバーラップして実行できる．

GeForce GTX 295 と Tesla C1060 の compute capability は同じ 1.3 である．そのため，この特徴は compute capability 1.3 の特質だと考えられる．

このように，転送とカーネルを重ねるとき，カーネルが短く転送だけが実行される部分があると，その部分の転送効率が落ちる GPU があることがわかった．この問題を回避するには，以下の方法が挙げられる．転送と同じストリームか，ストリーム 0 のカーネルが転送の後に呼ばれるとき，

- カーネルとセンドを重ねて処理させない．



(a) Tesla C1060, ver=3.1, CC=1.3　(b) GeForce GTX 295, ver=2.1, CC=1.3

(c) Tesla C2050, ver=3.2, CC=2.0　(d) ION, ver=3.1, CC=1.1

図 5 転送中のカーネル呼び出し
Fig. 5 Kernel invocation while Transfer

- 転送と重なるカーネルは，転送よりも時間が掛かるタスクにする．
- 転送の終了を待機するカーネルを存在させないため，同期 API を挿入する．

4. アルゴリズム

本研究では，データ転送とカーネル実行の小さい方を隠蔽するスケジュールを実現することを目的とする．この目的のためにデータ転送とカーネルを動的に分割し，オーバーラップして実行する．本節ではスケジューラーの実装について述べる．

図 7 にスケジューラーを用いてタスクをオーバーラップして実行する例を示す．これを

```
1: プラットフォームの転送速度の測定
2: スケジューラーの実行
3: if 十分なブロック数がある then
4:     ブロック当たりのカーネル実行時間の測定
5:     タスクに分割し，センドとカーネルの実行
6:     カーネルの実行
7:     タスクに分割し，レシーブとカーネルの実行
8: else
9:     カーネルの実行
10:    レシーブの実行
11: end if
```

図 6 アルゴリズム
Fig. 6 Algorithm

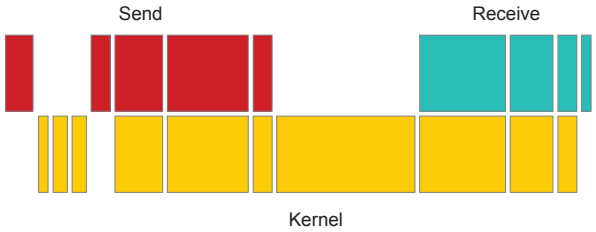


図 7 実現するスケジュール
Fig. 7 Timeline to achieve

実現するアルゴリズムを**図 6** に示す.

スケジューラーを使用して処理を始める前に，ターゲットプラットフォームごとに転送時間やオーバーヘッドを調べておかなくてはならない．これはプラットフォームごとに一度だけ行えばよい.

スケジューラーを実行すると，まずは小さいサイズのデータを転送する．次にそのデータを使い，カーネルを実行し，その時間を測定する．これをブロックのサイズを変えて数回実行し，ブロック当たりのカーネル時間を計算する．ここでは転送データ量を減らすため，測定のたびに同じブロックを計算している．これにより，カーネルにかかる時間が予測できるようになるので，センドとカーネルが重なるようにタスクに分割し，並列に実行していく．このとき 3 節で述べたように，カーネルとセンドを呼ぶたびに同期 API を挿入する．センドが全て終わると，レシーブとの終了を合わせるために，カーネルのみを実行する．そして，レシーブとカーネルのタスクサイズを合わせ，並列に処理していく．この場合も上記と同様に，同期 API を入れる.

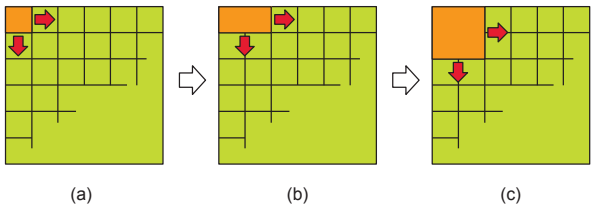


図 8 データを転送するブロックの決定
Fig. 8 Decision of block to which data is transmitted

4.1 転送するブロックの選択

次に，本アルゴリズムではどのようにデータを転送するブロックを決めるか説明する.

本研究では，各ブロックでカーネルにかかる時間は一定だが，転送データ量は一定ではないと仮定している．そのため転送ブロックの選び方によって転送にかかる時間が異なる．そこで，ブロックを決定する際の基準として，転送量あたりの実行可能ブロック数が多くなる様にブロックを選ぶこととした.

例えば，ブロックが**図 8** の (a) のような場合は，左上から始めて，実行可能ブロック数/データ転送量が大きくなるように，右か下かにデータを転送するブロックの選択を広げる．選択後の様子を (b) とすると，次は現在選択されているブロックから始めて，右か下かに広げる．これを繰り返すことで，必要なサイズの転送データを決定する．この転送が終わると，カーネルはそのデータを使い，ブロックの計算を行う．この処理のためには，どのデータが既に転送済み，もしくは選択済みか記憶しておく必要がある．選択するブロックを広げる際は，その情報を元にデータ転送量を計算する.

5. 評 価 実 験

本節では，前節で説明したアルゴリズムを実際の問題に適用した結果を紹介する．実験として，行列積のプログラムにスケジューリングを適用した.

行列積は，**図 9** にあるように，入力として，2 つの行列データを必要とし，計算結果として 1 つの行列を返す．行列 C を 16×16 のサイズで分割し，各ブロックはその部分行列を計算する．そのため，各ブロックは $16 \times M$ と $M \times 16$ のデータを必要とする．行列積では各部分行列の計算に必要なデータに共通部分があるため，ブロックあたりの転送量は一定ではない.

Vol.2011-HPC-129 No.22
2011/3/16

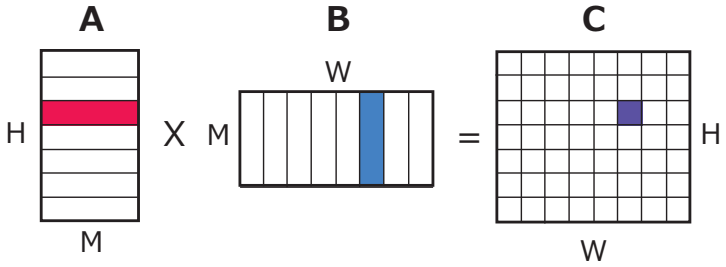


図 9 行列積プログラム
Fig. 9 Matrix multiplication

表 1 マシン情報
Table 1 Machine specifications

Machine	CPU	Ver	GPU	CC	Send speed	Recv speed
1	Intel Core i7 -980X	3.2	Tesla C2050	2.0	5.77 GB/s	5.62 GB/s
2	Intel Core i7 -870s	3.2	GeForce GTX 580	2.0	5.89 GB/s	5.93 GB/s

ここでは，**表 1** にある 2 つのマシンで，H，M と W の大きさを変えて測定した結果を**表 2** と**表 3** に示す．

多くのサイズで，本スケジューラーを使うと，逐次に実行した場合と比べ，実行時間短縮できている．しかし，転送のみ，カーネルのみの時間より時間は長く，オーバーヘッドが掛かっていることがわかる．これは初めの測定の部分はタスクを重ねて実行することができず，また，同じブロックを複数回計算していることで，余計に時間が掛かっているためである．また，$H = 256, M = 16384, W = 256$ や $H = 512, M = 16384, W = 512$ のときは，逐次に実行したときよりも時間が掛かってしまっている．これらのサイズの場合，総ブロック数が入力データ量に比べて少なく，1 ブロックあたりに必要な計算量が大きい．そのため，同じブロックを複数回計算する測定方法ではオーバーヘッドが大きくなるからである．

6. ま と め

本稿ではデータ転送とカーネルを自動で複数のタスクに分割し，オーバーラップして実行するスケジューラーを提案した．スケジューラーは事前測定と，実行時の測定を行うことでモデルを構築し，転送とカーネルをオーバーラップさせるようタスクのスケジュールを構築する．評価実験では行列積プログラムに対してスケジューリングを適用した．その結果，データ転送とカーネル実行をオーバーラップさせることにより実行時間を短縮することができた．しかし，ブロックの数が少ない場合に，オーバーヘッドが大きく，理論時間を大幅オーバーしてしまうことがわかった．今後の課題は，ブロック数が少ない場合であっても，オーバーヘッドが少ないカーネル測定方法を実装すること，また，複数 GPU に対応させることが考えられる．

表 2 マシン 1 による行列積プログラムの実行時間
Table 2 Execution time of matrix multiplication program with Machine 1

	H=512 M=512 W=16384	H=16384 M=512 W=512	H=8192 M=8192 W=256	H=256 M=256 W=8192	H=256 M=16384 W=256	H=512 M=16384 W=512
センド	5.803	5.810	46.102	46.116	5.628	11.211
カーネル	49.069	49.104	214.988	214.495	16.271	62.142
レシーブ	5.204	5.201	1.328	1.328	0.075	0.196
逐次	60.005	60.051	262.399	262.173	21.905	73.487
スケジュール	53.250	51.011	231.716	232.876	45.263	89.303

	H=2048 M=2048 W=2048	H=4192 M=4192 W=4192	H=16384 M=4096 W=32	H=16384 M=8192 W=32	H=8192 M=32 W=8192	H=16384 M=32 W=16384
センド	5.630	23.431	44.810	89.526	0.394	0.746
カーネル	97.829	833.395	24.804	54.065	27.284	108.911
レシーブ	2.618	10.854	0.359	0.357	41.372	165.330
逐次	106.012	867.669	69.924	144.161	68.978	274.909
スケジュール	102.368	849.511	51.613	107.436	45.124	171.600

謝辞 本研究の一部は JST CREST「ULP-HPC: 次世代テクノロジのモデル化・最適化による超低消費電力ハイパフォーマンスコンピューティング」，科学研究費新学術領域研究「コンピューティクスによる物質デザイン」計画研究「超高速・超低消費電力物質科学シミュレーション方式の研究開発」の支援を受けています．

参 考 文 献

1) R.Suda, T.Aoki, S.Hirasawa, A.Nukada, H.Honda, and S.Matsuoka. Aspects of gpu for general purpose high performance computing. *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, pages 216–223, 2009.
2) S.Nakagawa, F.Ino, and K.Hagihara. A middleware for efficient stream processing in cuda. *Computer Science - Research and Development*, 25(1):44–49, 2010.

Vol.2011-HPC-129 No.22
2011/3/16

表 3 マシン 2 による行列積プログラムの実行時間
Table 3 Execution time of matrix multiplication program with Machine 2

	H=512 M=512 W=16384	H=16384 M=512 W=512	H=8192 M=8192 W=256	H=256 M=256 W=8192	H=256 M=16384 W=256	H=512 M=16384 W=512
センド	5.632	5.628	44.831	44.845	5.458	10.892
カーネル	31.736	31.853	139.008	138.614	10.144	39.545
レシーブ	5.397	5.399	1.386	1.359	0.056	0.182
逐次	31.736	42.904	185.411	185.194	15.669	50.675
スケジュール	35.224	33.727	149.536	150.023	35.890	58.128

	H=2048 M=2048 W=2048	H=4192 M=4192 W=4192	H=16384 M=4096 W=32	H=16384 M=8192 W=32	H=8192 M=32 W=8192	H=16384 M=32 W=16384
センド	5.450	21.747	43.586	87.155	0.359	0.699
カーネル	63.518	511.291	16.035	35.004	17.768	70.982
レシーブ	2.711	10.841	0.352	0.350	43.226	172.917
逐次	71.694	544.018	60.210	123.059	61.473	244.822
スケジュール	67.174	520.175	49.356	99.154	46.279	180.389

3) S.Nakagawa, F.Ino, and K.Hagihara. 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**, 2010-HPC-126(19):1–8, 2010.