Vol.2016-HPC-155 No.8
2016/8/8

GPGPUフレームワークMESI-CUDAにおけるデータ再利用性を高めるスケジューラ

田中 宏明[1,a)]　山本 怜[1,b)]　大野 和彦[1,c)]

概要：GPU 上で汎用計算を実行する GPGPU の分野において，複数の GPU を利用するマルチ GPU 環境を用いてより高い計算性能を実現する試みがなされている．現在主流の開発環境である CUDA はマルチ GPU に対応しているが，個々の GPU を明示的に操作する必要があり，プログラムの記述が煩雑になる．また，各 GPU に分散させたメモリデータの結合・再利用や個々のマルチ GPU 環境に適したスケジューリングをユーザ自身が実装する必要がある．我々は CUDA より簡単なプログラム記述で GPU を扱える GPGPU フレームワーク MESI-CUDA を開発している．MESI-CUDA はすべての CPU・GPU コアが単一の仮想共有メモリにアクセスするプログラミングモデルを採用しており，ホストメモリ・デバイスメモリの確保・解放やデータ転送などのコードを処理系で自動生成することで，このモデルで記述されたプログラムを CUDA コードに変換する．また，マルチ GPU をサポートしており，ユーザが記述したカーネル実行をタスクへ分割し，単一ホスト上の利用可能なすべての GPU へ振り分ける．現在の実装は，冗長なデータ転送の発生を防ぐためメモリ管理を行っている．具体的には，仮想共有される配列の内容をアクセス範囲によりセグメントとして分割し，デバイスメモリ上にキャッシュする．しかし，タスクの実行順や実行デバイスの選択によってキャッシュの利用効率が低下することがある．そこで本稿では，同じセグメントを利用するタスクを可能な限り同じデバイス上で連続して実行することによりキャッシュの利用効率を高めるスケジューリング手法を提案する．

キーワード：並列コンピューティング，GPGPU，CUDA

1. はじめに

GPU 上で汎用計算を実行する GPGPU は高性能計算に広く使われている．しかしながら，GPU デバイス 1 台の計算能力や物理メモリサイズは大規模計算において不十分である．そこで，複数の GPU を利用するマルチ GPU 環境を用いてより高い計算性能を実現する試みがなされている．現在主流の開発環境である CUDA[1] はマルチ GPU に対応しているが，個々の GPU を明示的に操作する必要があり，プログラムの記述が煩雑になる．また，各 GPU に分散させたメモリデータの結合・再利用や個々のマルチ GPU 環境に適したスケジューリングをユーザ自身が実装する必要がある．さらに，高い性能を得るためには，タスクの粒度や負荷分散のチューニングが必要である．このような手動最適化は難しく，移植性が低い．他のフレームワークには OpenACC[2] がある．OpenACC は OpenMP[3] のようなディレクティブ・ベースのプログラミング手法を採用しており，低レベルな API を隠蔽している．しかしながら，最適化のためには，低レベルなディレクティブが必要である．また，現行のバージョンではマルチ GPU を自動で使用しない [4].

我々は CUDA より簡単なプログラム記述で GPU を扱える GPGPU フレームワーク MESI-CUDA[5], [6], [7], [8], [9] を開発している．MESI-CUDA は GPU の低レベルな特徴を抽象化したモデルを採用しており，仮想共有変数や論理スレッドマッピング機構によって複雑なメモリ階層や物理的な特徴をユーザから隠蔽する．MESI-CUDA コンパイラは MESI-CUDA プログラムを CUDA コードへと，低レベルなコードを自動生成することで変換する．この際，コンパイラは性能向上のために静的解析と最適化を行う．また，動的スケジューリング機構 [9] により，マルチ GPU をサポートしている．本機構はユーザプログラムのカーネル起動をタスクへと分割し，利用可能なすべての GPU へ振り分ける．また，タスクの実行時，冗長なデータ転送の発

[1] 三重大学大学院 工学研究科
Mie University
[a)] hiroaki@cs.info.mie-u.ac.jp
[b)] yamamoto@cs.info.mie-u.ac.jp
[c)] ohno@cs.info.mie-u.ac.jp

生を防ぐためメモリ管理を行っている．具体的には，仮想共有される配列の内容をアクセス範囲によりセグメントとして分割し，デバイスメモリ上にキャッシュする．しかし，タスクの実行順や実行デバイスの選択によってこのキャッシュの利用効率が低下することがある．

そこで，キャッシュの利用効率を高めるスケジューリング手法を提案する．本手法では，アイドル状態のデバイスに割り当てるタスクを未スケジューリングのタスクから選択する際，その時点でのキャッシュの内容を考慮し，必要なデータ転送が最も少ないタスクを選択する．また，キャッシュミスによりキャッシュされているセグメントから置き換えるものを選択する際に，アクセスする未スケジューリングタスクが最も少ないセグメントを選択する．これらにより，キャッシュ効率が高まり実行時間を削減できる．LRU でセグメントを置換する従来手法では，各セグメントが順にアクセスされるような場合，毎回アクセス前に置換されてしまう状況が発生していたが，提案手法により回避できるようになった．これにより，従来手法に比べ最高 80 %程度実行時間を削減できた．

2. 背景

2.1 GPU と CUDA

GPU は一定数の CUDA コアを持つストリーミングマルチプロセッサー（SM）の集合である．**図 1**(a) に GPU カードを搭載した PC の典型的なアーキテクチャを示す．CPU がメインメモリ（ホストメモリ）を共有するように，すべての CUDA コアは大きなオフチップメモリであるデバイスメモリを共有する．さらに，それぞれの SM は小さなオンチップメモリであるシェアードメモリを持ち，シェアードメモリは SM 内のすべての CUDA コアにより共有される．

CUDA(Compute Unified Device Architecture)[1], [10], [11] は GPGPU プログラミングフレームワークであり，C/C++または Fortran を拡張した文法とライブラリ関数を用いて GPU プログラミングを行える．CUDA では`__device__`もしくは`__global__`修飾子のついたカーネル関数と呼ばれる関数のみ GPU 上で実行され，その他の関数（ホスト関数）はホスト上にて実行される．

CUDA はグリッドとブロックを用いてデータと物理資源へのスレッドマッピングを制御する．ブロックはスレッドの集合であり，同じ SM 上で実行される．また，グリッドは同じサイズのブロックの集合である．カーネルの起動時には，このブロックとグリッドのサイズをそれぞれ指定する必要がある．

CPU と GPU 間でデータを共有するには，両メモリに対する領域確保とデータ転送が必要である．ユーザはこれら行うため，低レベルな API を記述しなければならない．また，ホストからデバイスへのデータ転送とデバイスからホストへのデータ転送はそれぞれダウンロード，リードバックと呼ぶ．

ホストに複数のデバイスが搭載されている場合，スレッドはそれらのデバイス上で並列に実行できる．しかしながら，ユーザは `cudaSetDevice()` によって使用デバイスを明示的に切り替え，データ転送やカーネルの実行は個々のデバイスに対して行う必要がある．

2.2 MESI-CUDA

CUDA の API は GPU の複雑なアーキテクチャを反映している．この低レベルな API はハードウェアのスペックを考慮したチューニングを可能としているが，そのコーディングは難しく，実行環境に依存する．そこで，我々はより簡単なフレームワーク MESI-CUDA[5], [6], [7], [8], [9] を開発している．

MESI-CUDA はすべての CPU/CUDA コアが一つのグローバルメモリを共有する，仮想共有メモリモデルを採用している（図 1(b)）．実際には，`__global__`修飾子が付加された変数のみ共有され，我々はこれを仮想共有変数，または，VS 変数と呼ぶ．各 VS 変数に対応した，ホストメモリ・デバイスメモリのデータ確保・解放やカーネル関数の前後でのデータ転送は自動的に行われる．

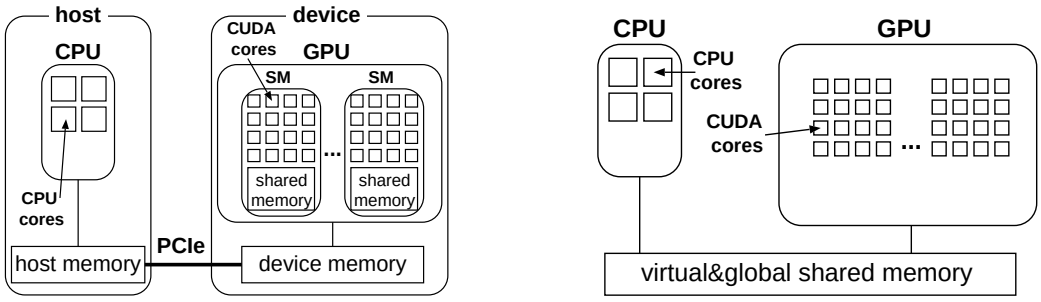
また，ユーザの手動チューニングなしで高い性能を得るためにコンパイラは静的解析に基づき最適化を行う．そのために我々はスレッド実行とデータ転送のオーバーラップ [5]，シェアードメモリを利用した明示キャッシュ [7]，デバイスメモリアクセスを改善するスレッドマッピング [8] といった自動最適化機構を開発している．

2.2.1 動的タスクスケジューリング機構

GPU は仮想メモリやファイルシステムをサポートしておらず，デバイスメモリのサイズは最新のモデルでも 16GB と一般的な PC のメインメモリと比べると少ない．よって，デバイスメモリサイズを越えるようなデータ配列を扱う大規模計算は一度のカーネル起動で実行できない．しかし，多くのケースでスレッド内の配列のアクセス範囲は限られている．ゆえに，計算を複数のカーネル起動に分けることで，それぞれの配列の転送をアクセス範囲のサイズに減らすことができる．

また，GPU デバイス１台の計算能力は大規模計算に対し不十分な場合がある．CUDA は多くのスレッドを生成することができるが，スレッドやブロックの並列実行は SM 数などの物理資源に制限される．そこで，利用可能な GPU が複数ある場合，カーネル起動をそれらの GPU へ振り分けることで実行時間を削減できる．

これらのケースに対応するため，我々は MESI-CUDA の動的タスクスケジューリング機構を開発している．本機構では，カーネル起動および対応するデータ転送はジョブへと置き換えられ，ランタイムスケジューラへと投入され



(a) GPU アーキテクチャ　(b) MESI-CUDA プログラミングモデル

図 1　GPU アーキテクチャとプログラミングモデル

る．その後，スケジューラはジョブをタスクへと分割し，利用可能な GPU デバイスへ動的にタスクを割り当てる．

2.2.2　データ再利用

同じデバイス上のタスク間でデータを共有するため，タスクのアクセスする配列要素を含む VS 配列変数の部分的なコピーを VS セグメントとして管理している．タスク開始時，入力 VS 変数に対応する VS セグメントに対して，動的に領域確保・転送が行われる．この際，領域確保・転送を行ったデバイスと VS セグメントを記録しておき，それ以降のタスクで同じ VS セグメントが使われる場合に再利用している．また，タスク終了時，タスクで書き込みを行った VS セグメントが他デバイス上にも存在する場合，それを無効化することで一貫性を保っている．このように，VS セグメントはデバイスメモリ上でキャッシュとして扱われる．

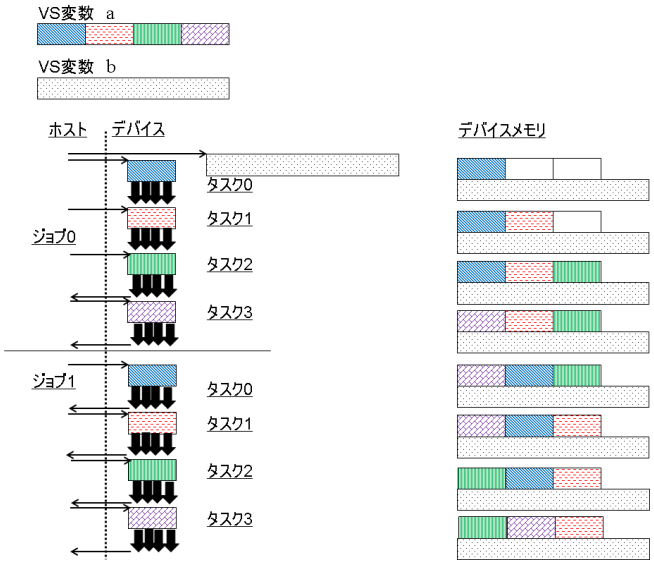


図 2　従来手法の動作例

2.3　現状の問題点

前述のように，動的タスクスケジューリング機構は VS セグメントをキャッシュすることで不必要なデータ転送を削減している．しかしながら，デバイスメモリサイズを越えるデータを扱うプログラムではキャッシュされた VS セグメントの置換が発生し，キャッシュの利用効率が下がる場合がある．

例えば，hotspot のように配列全体のステンシル計算を行うステップシミュレーションの場合を考える．1 ステップの計算を 4 つのタスクに分割した実行の様子を**図 2** に示す．ここで，デバイスメモリには VS 変数 a について，3 セグメント分しか領域が確保できなかったとする．現状の実装では，GPU が利用可能になるとタスクキューの先頭からタスクを一つ取り出し，実行を開始する．また，キャッシュの置換は同サイズの VS セグメントに対して LRU で行われる．従って，タスク 3 開始時，タスク 0 に対応した VS セグメントが追い出され，タスク 3 に対応する VS セグメントがキャッシュされる．ゆえに，タスク 3 終了時点でデバイスにはタスク 1,2,3 にそれぞれ対応する VS セグメントがキャッシュされている．

2 ステップ目も同様に．まずタスクキューの先頭のタスク 0 が実行される．その際，タスク 0 で使用する VS セグメントは前ステップで置換されており，再度データ転送が必要となる．また，この時置換する VS セグメントとして LRU により VS セグメント 1 が選択されるため，次のタスク 1 の実行で再びデータ転送が発生する．これを繰り返すため，キャッシュした VS セグメントが利用される事はなく，データ再利用は起こらない．これに対し，最適なタスク実行順では毎ステップ 1 回の転送で済む（**図 3**）．

3.　提案手法

前述の問題点を解決するため，MESI-CUDA に対するキャッシュを考慮したスケジューリング手法を提案する．

3.1　基本方針

今回はメモリの管理方法について，次のような実装を前提としている．まず，各 VS セグメントのサイズ分のメモリブロックを予め一定数確保しておく．そして，各 VS セグメントは自身と同じサイズのメモリブロックにキャッシュされる．これは，複数サイズの領域確保・解放を繰り返すことにより，フラグメンテーションの問題が発生する

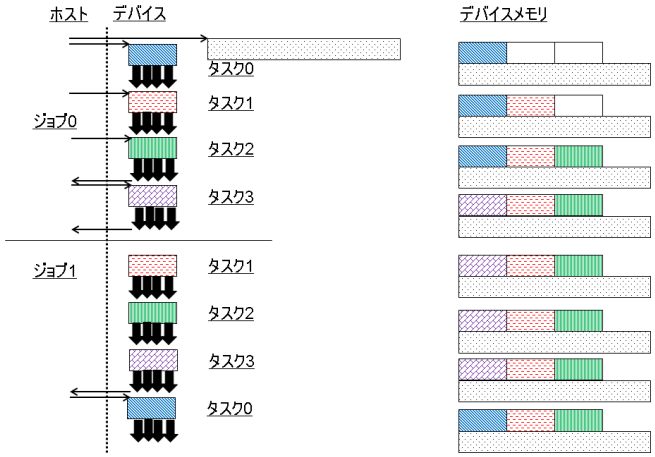


図 3 最適な動作例

のを回避するためである．各サイズのメモリブロックをいくつずつ確保するかによりキャッシュ効率が変動することが考えられるが，最適な設定方法については今後の課題とする．

スケジューラはアイドル状態のデバイスを発見すると，次に実行するタスクを未スケジューリングのタスクから 1 個選択する．また，同デバイス上にそのタスクで必要な VS セグメントがなく，同サイズのメモリブロックに空きがない場合，キャッシュされた同サイズの VS セグメントから 1 個選択し置換する．本手法ではこれら二つの選択アルゴリズムを，未スケジューリングタスクがアクセスする VS セグメントとキャッシュされている VS セグメントを考慮したものに改良する．

3.2 実行タスクの選択

アイドル状態のデバイスで実行するタスクを選択するアルゴリズムを**図 4** に示す．select_next_task は未スケジューリングタスクのリストである task_list から device_ID で指定されたアイドル状態のデバイスで次に実行するタスクを選択する．すべての未スケジューリングタスクを走査し (4-11 行目)，各タスクがアクセスするすべての VS セグメントについてデバイスにキャッシュされていないものの総転送量を求める (6-10 行目)．8 行目の *get_segment_size* は引数で与えられた VS セグメントのメンバからサイズを返す．そして，求めた総転送量が最小のタスクを求める (11-13 行目)．

この方法により，ある時点でのキャッシュ状況に基づいた最適なタスクの選択が行えると考える．しかしながら，アイドル状態のデバイスのキャッシュのみを考慮しているため，デバイスが複数あった場合，他のデバイスのキャッシュ状況により総合的に最適なタスクかどうかは判断できない．従って，VS セグメントを共有するタスク同士が連続して実行されるよう，ジョブをタスクに分割する際，共有セグメントを基準にタスクをクラスタリングする．

```
 1: procedure select_next_task(task_list, device_ID)
 2:     min_transfer_task = NULL
 3:     min_eval = MAX_VALUE
 4:     for each task in task_list do
 5:         eval = 0
 6:         for each segment in task do
 7:             if segment is Invalid in segment_table then
 8:                 eval = eval + get_segment_size(segment)
 9:             end if
10:         end for
11:         if eval < min_eval then
12:             min_eval = eval
13:             min_transfer_task = task
14:         end if
15:     end for
16:     return min_transfer_task
17: end procedure
```

図 4 割り当てタスク選択アルゴリズム

3.3 置換 VS セグメントの選択

キャッシュの置換が必要となったとき，置換する VS セグメントを選択するアルゴリズムを**図 5** に示す．select_replace_segment は device_ID で指定されたデバイスにキャッシュされている VS セグメントの中で，メモリ確保したい VS セグメントである target_segment と同サイズのものから置換対象を選択する．まず，置換する VS セグメントの候補を target_segment_list へ取得し (2,3 行目)，これらについて走査する (6-11 行目)．各置換候補 VS セグメントについて，アクセスする未スケジューリングのタスク数を求める (7-9 行目)．そして，求めたタスク数が最小になる VS セグメントを求める (10-13 行目)．

この方法により，キャッシュの置換が発生した時点で，未スケジューリングタスクをそのデバイスで全て実行するとしたときアクセスするタスク数の一番少ない VS セグメントが置換される．前述の実行タスクの選択方法により，キャッシュに留めた VS セグメントをアクセスするタスクは近い将来実行されるため，キャッシュヒット率が向上することが期待できる．しかしながら，スケジューラに投入済みのジョブのタスクしか走査していないため，後に投入されるジョブのタスクで再利用できる VS セグメントがあったとしても，考慮できない．

4. 関連研究

計算データの再利用を行う機構として，1 次キャッシュや TLB(Translation Lookaside Buffer) のメモリアクセス高速化のための古典的なメモリ管理機構が存在する [12]．また，複数のホストを同一ネットワーク上に構築する分散メモリ環境においては，ホスト間の通信コストを軽減するために，効率よいデータ転送やデータキャッシングの手法が数多く提案されている．マルチ GPU 環境を想定する我々の手法は，解析情報を元に各デバイスメモリ上に分散

```
1: procedure select_remove_segment(target_segment, device_ID)
2:     segment_size = get_segment_size(target_segment)
3:     target_segment_list = get_segment_list(segment_size, device_ID)
4:     remove_segment = NULL
5:     min_eval = MAX_VALUE
6:     for each segment in target_segment_list do
7:         for each task in unscheduled tasklist do
8:             eval + = eval_function(segment, task)
9:         end for
10:        if eval < min_eval then
11:            min_eval = eval
12:            remove_segment = segment
13:        end if
14:    end for
15:    return remove_segment
16: end procedure

1: procedure eval_function(segment, task)
2:     eval = 0
3:     for each segment_ in task do
4:         if is_same_segment(segment,segment_) then
5:             eval = eval + 1
6:         end if
7:     end for
8: end procedure
```

図 5 置換 VS セグメント選択アルゴリズム

されたデータが再利用可能か否かをホスト上で管理し，ホスト・デバイス間データ転送の最小化を実現する．このメモリ管理機構は，従来のメモリ管理機構を適用することで実現できる可能性がある．しかし，あるデバイスでメモリ不足の際に，どの再利用可能なデータを破棄するかを決定するアルゴリズムは，従来のメモリ管理機構とは異なり，アプリケーション依存の問題として議論する必要がある．また，そのアルゴリズムは各デバイスの稼動状況と配置メモリデータに依存するため，スケジューリング問題と合わせて議論する必要がある．

計算単位を計算資源に割り当てるタスクスケジューリングはこれまでに多数の研究が行われている [13]．一般的なスケジューリング問題では，ジョブの間に依存関係があることが多く，全体としては DAG 型のタスクスケジューリング問題と見なせる．しかし，ユーザはカーネル起動をコード内で逐次的に書くので，ジョブ間の並列性は期待できない．一方で，直接スケジューリングの対象になるタスクについては，同一のジョブ内のタスクとの間に依存がないという特徴がある．よって，我々の研究対象は，計算負荷についてはジョブごとに小規模の独立タスクスケジューリングとみなすことで，従来手法を応用できる．しかし，転送量の最小化については，セグメントをジョブ間で共有する場合があるため，ジョブ間にまたがる最適化が必要である．

また，GPU プログラムを対象に効率的なメモリ管理を行う研究として，様々なものが提案されている [14], [15], [16].

表 1 評価プログラム

プログラム名	概要
hotspot	2 次元過渡熱シミュレーション（size:8192^2, 1000steps）
srad	2 次元超音波画像のノイズ除去（size:4096^2, 1000steps）

表 2 評価環境

	環境 1	環境 2
GPU	Geforce GTX TITAN	Geforce 980
device memory	6 GB	4 GB
CPU	Core i7-4820K 3.70GHz	Xeon E5-1620 3.60Ghz
host memory	16 GB	16 GB

5. 評価

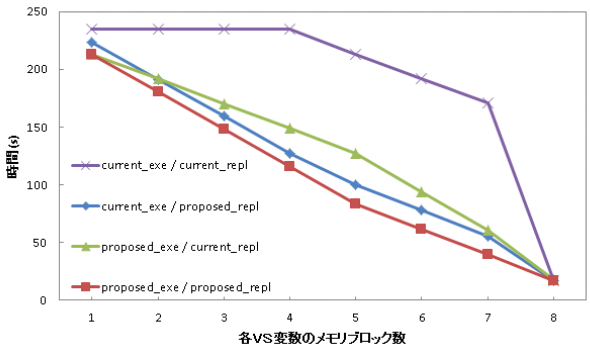
提案手法による性能向上を調べるため評価を行った．評価に用いたプログラムと実行環境をそれぞれ**表 1**,**2** に示す．

各プログラムの実行時間をそれぞれ**図 6**,**7** に示す．各プログラムに対し，実行タスクの選択に従来手法（current_exe）と提案手法（proposed_exe），また，置換 VS セグメントの選択に従来手法（current_repl）と提案手法（proposed_repl）の 4 つの組み合わせで評価を行った．横軸は各 VS 変数のメモリブロックの確保数（N_b）である．本来は各サイズごとのメモリブロックを最適な数確保するが，これはプログラムとデバイスメモリ量によって変化し，キャッシュできる VS セグメント数も変化する．今回はキャッシュできる VS セグメント数を変えて評価を行うことで，より様々な環境での従来手法と提案手法の関係を調べた．縦軸は実行時間である．また，すべてのジョブをそれぞれ 8 個のタスクに分割している．ただし，今回は 3.2 節で述べたタスクのクラスタリングは適用していない．

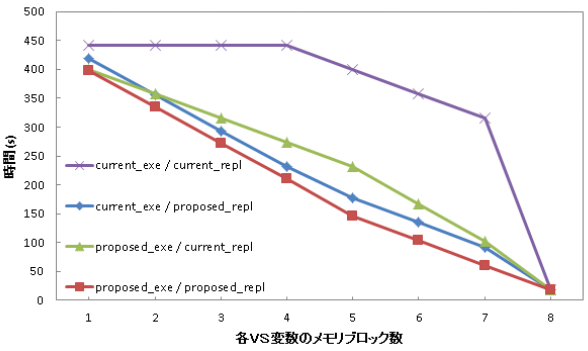
両プログラムとも，N_b が 8 の時，VS セグメントの入れ替えが発生しないため，キャッシュの挙動は手法によらず同一となる．このとき手法による実行時間差がほとんどみられないことから，提案手法によるオーバヘッドは実用上無視できるほど小さいことがわかる．また，N_b が 1 から 7 の時，proposed_exe/proposed_repl が最もよい結果となっている．よって，両プログラムにおいては提案手法の効果があったと言える．current_exe/current_repl では，N_b が 8 の時に比べて，1 から 7 の時の性能は極端に低い．これは 2.3 節の例で説明したような，次に実行されるタスクがアクセスする VS セグメントの追い出しが順々に起きたためだと考えられる．

6. おわりに

本研究では MESI-CUDA におけるデータ再利用性を高めるスケジューリング手法を提案した．評価の結果，提案手法は 2 つのプログラムでキャッシュ効率を向上させるこ

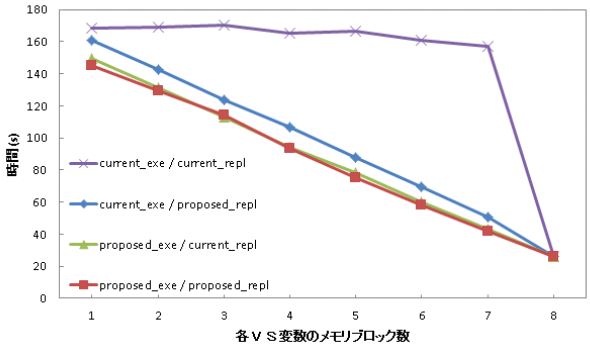


(a) 環境 1

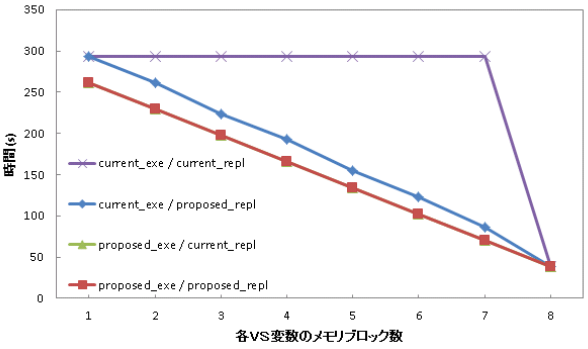


(b) 環境 2

図 6 hotspot



(a) 環境 1



(b) 環境 2

図 7 srad

とにより，従来手法に比べ実行時間を削減できた．今後の課題として，複数のジョブやタスクで共有する VS セグメントがある場合のスケジューリング，VS セグメントの各サイズごとのメモリブロックの最適な確保方法などがあげられる．

参考文献

[1] NVIDIA Corporation: CUDA Zone, 入手先 ⟨https://developer.nvidia.com/cuda-zone⟩（参照 2016-07-10）.

[2] OpenACC Home, 入手先 ⟨http://www.openacc-standard.org/⟩（参照 2016-07-10）

[3] OpenMP, 入手先 ⟨http://www.openmp.org/⟩（参照 2016-07-10）

[4] T. Komoda, S. Miwa, H. Nakamura, and N. Maruyama. *Integrating multi-GPU execution in an OpenACC compiler.* In Proc. 42nd Intl. Conf. on Parallel Processing, pages 260-269, 2013.

[5] K. Ohno, D. Michiura, M. Matsumoto, T. Sasaki, and T. Kondo. *A GPGPU programming framework based on a shared-memory model.* Parallel and Distributed Computing and Networks, 3:1-14, 2013.

[6] K. Ohno, M. Matsumoto, T. Kamiya, and T. Maruyama. *Supporting dynamic data structures in a shared-memory based GPGPU programming framework.* In Proc. 24th IASTED Intl. Conf. on Parallel and Distributed Computing and Systems, pages 122-131, 2012.

[7] T. Kamiya, T. Maruyama, K. Ohno, and M. Matsumoto. *Compiler-level explicit cache for a GPGPU programming framework.* In Proc. The 2014 Intl. Conf. on Parallel and Distributed Processing Techniques and Applications, pages 632-638, 2014.

[8] K. Ohno, T. Kamiya, T. Maruyama, and M. Matsumoto. *Automatic optimization of thread mapping for a GPGPU programming framework.* In 2014 2nd Intl. Symp. on Computing and Networking, pages 198-204, 2014.

[9] K. Ohno, R. Yamamoto, and H. Tanaka. *Dynamic Task Scheduling Scheme for a GPGPU Programming Framework* In 2015 3rd International Symposium on Computing and Networking(CANDAR), pages 181-187, 2015.

[10] NVIDIA Corporation: *CUDA C Programming Guide*, (2012).

[11] NVIDIA Corporation: *CUDA C Best Practices Guide*, (2012).

[12] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. *A modified approach to data cache management.* In Proc. 28th Ann. Int. Symp. on Microarchitecture(MICRO-28), pages 93-103, 1995.

[13] 須田礼仁, “ヘテロ並列計算環境のためのタスクスケジューリング手法のサーベイ, ” 情報処理学会論文誌:コンピューティングシステム, vol. 47, no. SIG 18(ACS 16), pages 92-114, 2006.

[14] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. *StarPU: a unified platform for task scheduling on heterogeneous multicore architectures.* Concurr. Comput. : Pract. Exper., Vol.23, pages 187-198, 2011.

[15] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R.

Beard, and D. I. August. Automatic *CPU-GPU communication management and optimization.* SIGPLAN Notice 47(6), pages 142-151, 2011.

[16] K. Wang, X. Ding, R. Lee, S. Kato, and X.Zhang. *GDM: Device Memory Management for GPGPU Computing.* In Proceedings of the 2014 ACM International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '14, pages 533-545, 2014.