修士論文

題目

マルチGPU上の
効率的な実行手法のためのランタイム

指導教員

大野　和彦　講師

2017年

三重大学大学院 工学研究科 情報工学専攻
コンピュータ・ソフトウェア研究室

田中　宏明(414M513)

内容梗概

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，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り，大きな負担となっている．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，低レベル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．このとき，効率的なデータ転送を行うため，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，タスクのアクセスパターンによってはセグメント間に重複部分が存在する．こういったケースにおいて一貫性の管理が行えず，冗長なデータ転送が発生する問題がある．そこで本稿では，上記の問題に対応するランタイムを提案する．本ランタイムは一貫性を管理するため，タスクのアクセス範囲を元に仮想共有変数を重複のないセグメントへと分割する．これにより，一貫性を保証するためのデータ転送量を削減する．

三重大学大学院　工学研究科

Abstract

In the field of GPGPU which executes general purpose calculation on GPU, Using multiple GPUs environment to achieve high performance has been attempted. Although CUDA which is the mainstream development environment supports multiple GPUs, it is a heavy burden because the user needs to implement the coupling and reuse of the memory data dispersed in each GPU and the scheduling suitable for each multiple GPUs environment. We are developing a GPGPU framework MESI-CUDA which enables the user to handle GPU with a simple program description. MESI-CUDA adopts a virtual shared memory which all CPU / GPU cores can access. The program written in this model is converted to CUDA code by automatically generating code in the processing system. In addition, it supports multiple GPUs by a dynamic task scheduling scheme. This scheme divides the kernel execution described by the user into tasks and schedules them to available devices installed on a single host. In order to perform efficient data transfer, the content of the virtual shared array is divided into segments according to the access range and cached on the device memory. However, depending on the access pattern, there is an overlapping part between the segments. In this case, the consistency management is not able to be performed so that redundant data transfer occurs. In this paper, we propose runtime managing the above problem. In order to manage consistency, this runtime divides the virtual shared variable into non-overlapping segments based on the access range of the task, thereby reducing the amount of data transfer for ensuring consistency.

三重大学大学院　工学研究科

目 次

図 目 次

表 目 次

1　はじめに

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．また，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，しかし，タスクのアクセスパターンによってはセグメント間に重複部分が存在する．こういったケースにおいて一貫性の管理が行えず，冗長なデータ転送が発生する問題がある．そこで本稿では，上記の問題に対応するランタイムを提案する．本ランタイムは一貫性を管理するため，タスクのアクセス範囲を元に仮想共有変数を重複のないセグメントへと分割する．そして，分割された各セグメントの一貫性を管理することで，必要なデータ転送を削減する．また，カーネル実行において，各タスクがアクセスするセグメントは仮想共有変数毎に連続領域格納されている必要がある．そこで新たに連続領域に格納すべきセグメント集合をメタセグメントと定義し，これを管理することで各セグメントを適切なアドレスへ転送する．さらに，プログラムで使用する総データ量がデバイスメモリ量を超えるケースに対応するため，様々なサイズのメタセグメントを格納する領域としてメモリブロック，同サイズのメモリブロックの固まりをメモリチャンクと定義し，メモリ管理を行う．

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（SM）の集合である．図 2.1 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, 11, 12] 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．

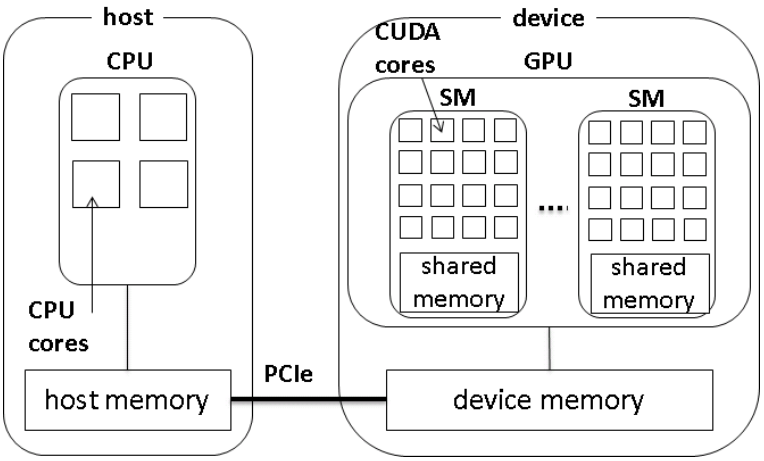

図 2.1: GPU アーキテクチャ

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, 10] 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 2.2）．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．

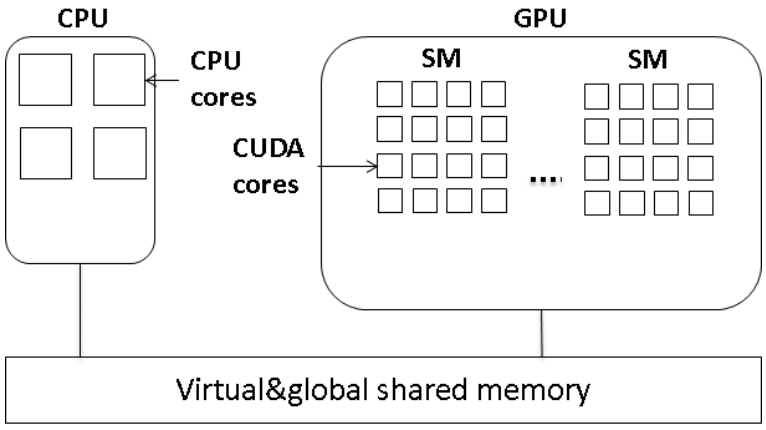

図 2.2: MESI-CUDA プログラミングモデル

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 [9]．本機構は静的解析を行い，カーネル起動および対応するデータ転送をジョブへと置き換えるコード生成機構と，実行時にジョブをタスクへと分割し，利用可能な GPU へ振り分けるランタイムによって構成される．

タスクの実行は，必要なデータをホストメモリからデバイスメモリへ

転送するダウンロード，カーネル起動，カーネル内で書き込んだデータを再びホストメモリへ戻すリードバック転送の順に行われる．タスク間に共有するデータが有る場合，該当データの転送を繰り返し行う事は性能低下の原因である．これを避けるため，動的タスクスケジューリング機構は，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．

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，タスクのアクセスパターンによってはVSセグメント間に重複部分が存在し得る．

重複が発生する例を図2.3に示す．これは1次元 (サイズ：1024) の配列 a,a_next について，a の各要素に対し近接要素を元にステンシル計算を行うステップシミュレーションである．ジョブ1はaを読み出して計算しa_nextに書き込む．ジョブ2は次のステップのためa_nextに書き込まれた結果を再びaに書き戻す．各ジョブは4つのタスク (t1-t4) へ分割され，矢印が各タスクのアクセス範囲を示している．この例では，変数aのVSセグメントがジョブ1において，各タスクのアクセス範囲の袖の2要素で重複する．

このようなアクセスパターンを持つ問題において，現状のデータ再利用機構は機能しない．ランタイムは重複部分の一貫性の管理が行えず，安全のため毎回重複部分を含むセグメントの転送を行う必要がある．ランタイムのこのふるまいは，図2.3のようなステップシミュレーションにお

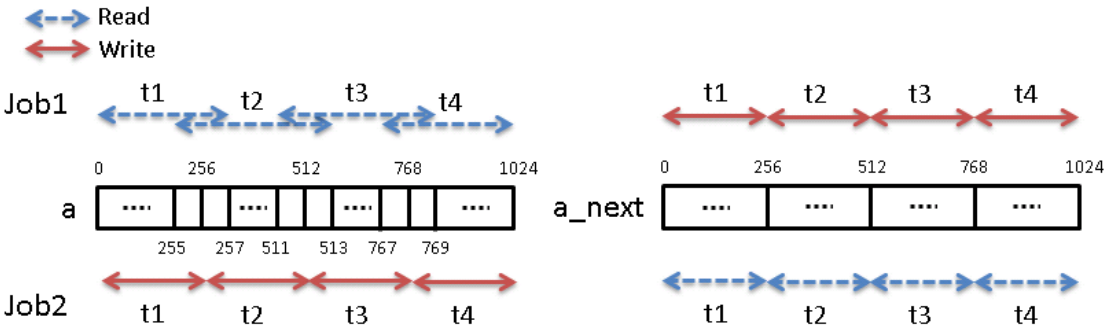

図 2.3: VS セグメントに重複が発生する例

いて，大きな性能低下を引き起こす.

3　提案手法

前述の問題を解決するため，タスク間のアクセス範囲の重複に対応したランタイムを提案する．

3.1　基本方針

前述の問題に対応するため，ランタイムは各 VS 変数をタスクが排他的にアクセスする部分とそうでない部分に分割して管理する必要がある．よって，ランタイムが VS 変数のホストメモリ上での物理的な位置と全タスクの各 VS 変数に対するアクセス範囲を把握する必要がある．そこで，プログラム開始時，コード生成機構により静的に埋め込まれた全 VS 変数の開始ポインタとサイズ，カーネル起動の為に必要な情報を格納したジョブをランタイムへ登録する．従来ランタイムへのジョブ登録はカーネル起動に置き換えられていたが，ここでは情報を登録するのみであり，カーネル起動はジョブの発火へと書き換えられる．また，新たにコード生成機構は各ジョブについて，VS 変数 ID，開始ブロック，ブロック数を元にアクセス範囲を返す関数を静的に埋め込む．ランタイムはジョブ登録時，ジョブをタスクへと分割する．このとき，各タスク・各 VS 変数のアクセス範囲をコード生成機構により新たに用意された関数より受け取り，VS 変数を分割する．すべてのジョブの登録終了後，分割された VS 変数の各範囲を VS セグメントと実行時に定義し，一貫性の管理を行う．

3.2　VS セグメントへの分割方法

各 VS 配列変数の VS セグメントへの分割方法を説明する．まずすべての VS 変数について，分割を行う際の処理用の木を用意する．その木のノードは，分割された VS 変数の開始要素のポインタと，最終要素のポインタを持つ．すべての VS 変数は登録された時点で，木のルートとしてその VS 変数の開始要素のポインタと最終要素のポインタを持つ．ジョブ登録時，ランタイムは任意のブロック数でタスクへと分割する．このとき，分割されたすべてのタスクと，それらのタスクのアクセスする VS 変数についてアクセス範囲を取得し，次の処理を適用する．

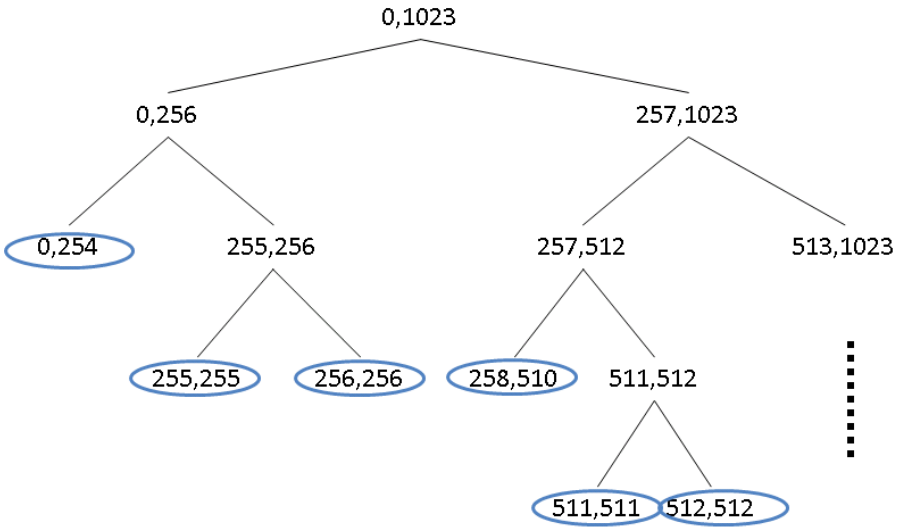

図 3.4: 分割後の木（図 2.3 の変数 a）

取得したアクセス範囲 (A 始端, A 終端) と既にある子無しノード N(N 始端, N 終端) を比較し

- N 始端 < A 始端 <= N 終端のとき
  - N の左の子ノードに N 始端から A 始端前の範囲のノードをセット
  - N の右の子ノードに A 始端から N 終端の範囲のノードをセット
- N 始端 <= A 終端 < N 終端のとき
  - N の左の子ノードに N 始端から A 終端の範囲のノードをセット
  - N の右の子ノードに N 終端後から A 終端の範囲のノードをセット

すべてのジョブ登録終了後，作られた木を前順深さ優先探索で子無しノードのみ取得することで，目的の分割が行える．図 2.3 の変数 a について分割を行った場合の例を図 3.4 に示す．なお，後半の要素については省略してあるが，前半と同様の分割が行われる．

3.3 メタ VS セグメント

プログラム中で使用される仮想共有変数の合計サイズがデバイスメモリ量を超える場合，すべての仮想共有変数の配列全体分の領域を確保することができない．このようなケースでは，タスクのアクセスする VS セグメントの領域をメモリの許す限り動的に確保していくなどする必要がある．しかしながら，カーネル内では仮想共有配列変数の要素は連続領域に存在するものとしてアクセスする．よって，タスクの実行には次の制約が存在する．タスク T_k のアクセスする各仮想共有変数 v_n の VS セグメント集合 $S(v_n)$ は物理メモリ上で連続に存在している必要がある．

この制約を満たすため，VS セグメント集合 $S(v_n)$ をメタ VS セグメント M_{nk} と定める．つまり，メタ VS セグメント M_{nk} はタスク T_k の仮想共有変数 v_n に対するアクセス範囲と同じ内容である．メモリ確保は VS セグメント単位でなく，メタ VS セグメント単位で行われる．ただし，あるメタ VS セグメントが他のメタ VS セグメントを包括する場合（ex:図 2.3 変数 a の Job1-t1 と Job2-t1）は，包括するメタ VS セグメントに統合する事でメモリ再利用を可能にする．

3.4 一貫性の管理

データの一貫性の管理は分割した VS セグメントに対して行われる．管理方法の基本的な方針は従来通り，全 VS セグメントが有効 (Valid) か無効 (Invalid) の状態とデバイスメモリ上の位置を示すポインタのテーブルを用意し，ダウンロード・リードバック・カーネル起動のタイミングで状態を更新する．ただし，メタ VS セグメントの導入によりテーブルを拡張する．従来のテーブルは，各デバイス，各仮想共有変数，各セグメントの 3 次元のテーブルである．前述の通り，メモリ確保はメタ VS セグメント単位で行われるため，メタ VS セグメント間で共有する VS セグメントが存在する場合，デバイスメモリ上に該当 VS セグメントが複数存在する．よって複数存在する VS セグメントを管理するため，メタ VS セグメント ID をキーにしたマップを追加する．さらに，ある VS セグメントに書き込みを行ったデバイス上に同 VS セグメントが存在する場合，無効化

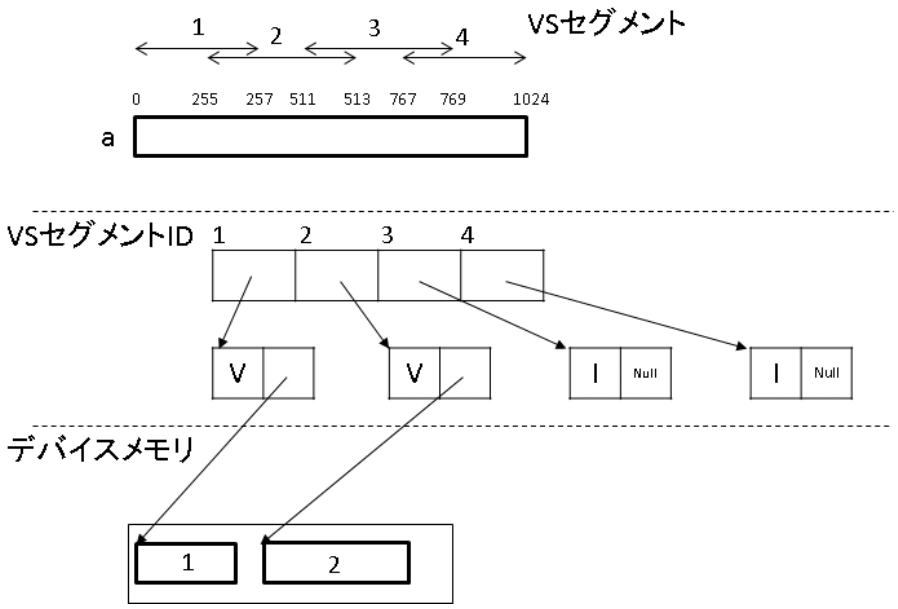

図 3.5: 図 2.3 変数 a のセグメントテーブル（従来）

する．図 3.5,3.6 に図 2.3 変数 a の従来セグメントテーブルと拡張後のセグメントテーブルをそれぞれ示す．

3.5 メモリチャンクとメモリブロック

使用データがデバイスメモリ量を超えるようなケースで，プログラムの前半のみ扱う仮想共有変数が存在する場合，使い終わったタイミングでその仮想共有変数のメタ VS セグメントを解放し，他の仮想共有変数のメタ VS セグメント用にその領域を使う事が好ましい．現状，こういったケースには対応していないが，将来対応する際に問題になると考えられることが，デバイスメモリの確保・解放の繰り返すによるメモリフラグメンテーションの発生である．これを防ぐには，デバイスメモリをすべて確保し，同じサイズのメタ VS セグメント用の領域を連続した空間に配置する事が有効である．そこで，様々なサイズのメタ VS セグメントを格納する領域としてメモリブロック，さらに，同じサイズのメモリブロックの固まりをメモリチャンクとして定義する．ランタイムはジョブ登録

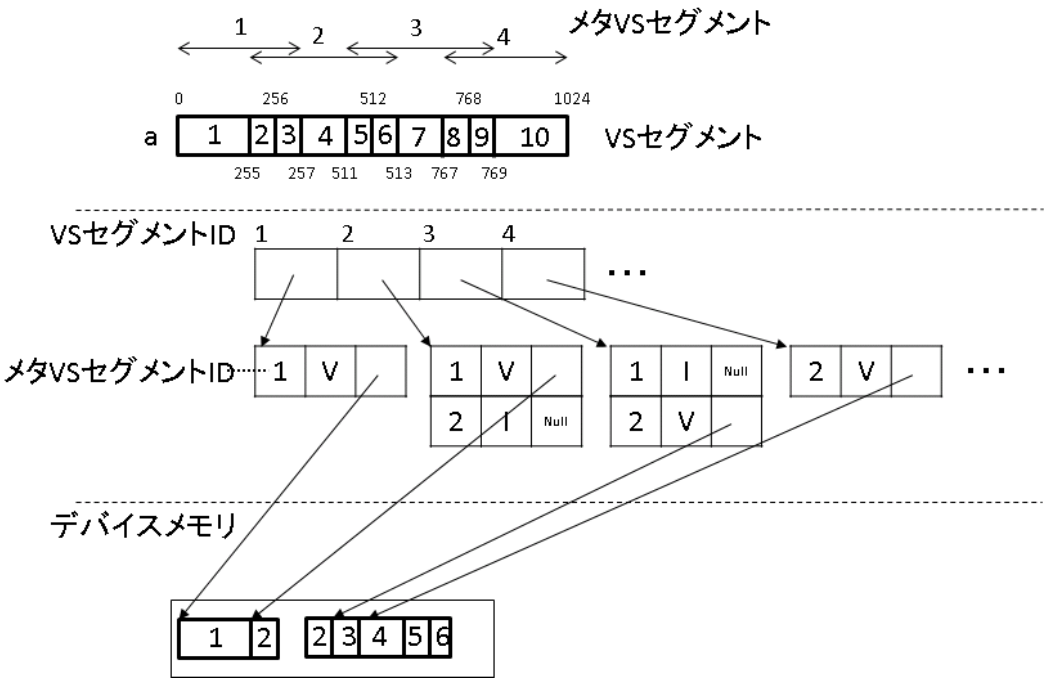

図 3.6: 図 2.3 変数 a のセグメントテーブル（拡張後）

終了後，全てのタスクについて最低限実行に必要なメモリブロックのサイズとその数，及び全てのメタ VS セグメントを格納するのに必要なメモリブロックのサイズとその数を計算する．デバイスメモリサイズからそれらの必要領域サイズを差し引いた残りのメモリを残りのメモリブロックに割り当てる．このとき，格納されるメタ VS セグメントのプログラム全体での属性が READWRITE であるメモリブロックを最優先し, 続いて WRITE, READ のものを割り当てていく．これは，メタ VS セグメントの置き換えが発生した場合のコストが高いと考えられるメタ VS セグメントをできるだけキャッシュに残して置くためである．各メモリブロックの数決定後，各メモリチャンクのサイズが決定し，デバイスメモリのアドレスを設定する．

4　評価

4.1　性能評価

2つのベンチマークプログラム（表4.1）を用いて提案手法の評価を行った．実行環境を表4.2に示す．Tesla K80は2つの同じデバイスを搭載した1枚のボードであり，ソフトウェアによって2つのデバイスとして認識される．よって環境1はヘテロジニアスなマルチGPU環境，環境2はホモジニアスなマルチGPU環境である．各環境でのマルチGPU環境をそれぞれ‘960+950’，‘k80(2)’と表す．また，各GPU1台での環境をそれぞれ，‘960’，‘950’，‘k80(1)’と表す．

表 4.1: 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（1000steps）
srad_v2	2次元超音波画像のノイズ除去（1000steps）

表 4.2: 評価環境

	環境1	環境2
GPU1	Geforce GTX 960	Tesla K80
Memory	2GB	24GB
GPU2	Geforce GTX 950	
Memory	2GB	
CPU	Xeon E5-2620 2.1GHz	Xeon E5-2630 2.4GHz×2
Memory	16GB	32GB

各プログラムの実行時間をそれぞれ表4.3, 表4.4に示す．提案ランタイムを実装した動的タスクスケジューリング機構を使用する結果を‘with’と表し，使用しない結果を‘without’と表す．使用しない場合でマルチGPUを扱うために，それぞれのカーネル起動を単純に2分割する．つまり，k80(2)において，‘without’の結果はマルチGPU環境で最適な結果

である．なお，‘with’ において，タスク数は 2,4,8,16 の中で最も良い結果のものである．プログラムで使用する総データ量がデバイスメモリを超えるケースは下線で示す．

プログラムに対し十分なデバイスメモリがある時，本機構を使ったマルチ GPU 環境と各シングル GPU 環境を比較すると，srad_v2・k80(2)・2096 を除いて実行時間が短い．また，メモリが足りない場合では，通常の実装ではプログラムを実行できない．しかしながら，本機構を使うことでデータの入れ替えなどを自動で行い実行できる．このことから，本機構はデバイスを増設した際，ユーザがコードに手を加えることなく性能を向上できる．さらに，デバイスメモリが足りないケースにおいても，本来実行できないプログラムを実行できる．

メモリが足りるケースにおいて，各環境の ‘without’ と ‘with’ を比較する．2048 において 1.5 から 2 倍近く実行時間がかかる．これは実行時間が短いため，本機構のオーバーヘッドが影響したためである．しかしながら，4096 以上では本機構のオーバーヘッドはわずかであり，無視できる．

また，メモリが足りないケースではタスク数 16 のとき最も良い結果が得られた．これは，タスク数が多くなるほどメモリブロックの粒度が増え，デバイスメモリを余りなく使えるようになるからである．よってタスク数の決定には，これまでに分かっていたデバイス間の負荷分散とスレッドブロック数によるデバイスリソースの利用効率の 2 つのトレードオフに加え，メモリブロックの粒度も考慮する必要があることが分かる．

さらに，注目すべき点として srad_v2・k80(2) において，‘with’ の結果よりメモリが足りるケースであるが，‘without’ の性能は大幅に悪化している．よって，マルチ GPU 環境におけるメモリブロック数の決定が最適でないと考えられる．

表 4.3: 実行時間 (秒) : hotspot

実行環境 / 問題サイズ		2048	4096	8192	16384
960	without	1.978	7.876	31.458	-
	with	2.663	8.903	32.917	829.603
950	without	2.191	8.694	34.408	-
	with	2.931	9.754	35.810	800.264
960+950	without	1.345	4.625	17.696	-
	with	1.757	5.004	18.5820	408.006
k80(1)	without	1.149	4.277	16.667	65.048
	with	1.727	5.019	17.130	66.368
k80(2)	without	0.845	2.478	8.844	34.878
	with	1.343	2.915	9.331	35.377

表 4.4: 実行時間 (秒) : srad_v2

実行環境 / 問題サイズ		2048	4096	8192	16384
960	without	9.091	35.675	-	-
	with	10.633	37.023	407.202	2737.722
950	without	11.332	44.448	-	-
	with	12.928	45.861	422.445	2814.324
960+950	without	6.319	23.898	-	-
	with	6.840	24.220	126.959	1855.378
k80(1)	without	3.251	11.914	46.707	-
	with	4.382	13.088	48.003	466.725
k80(2)	without	2.182	6.985	26.463	104.251
	with	3.165	7.485	26.626	491.288

4.2　メモリブロック数と総転送量

メモリブロック数の決定方法の評価を行うため，環境 1・hotspot・問題サイズ 16384・タスク数 16・10step でメモリブロック数を変化させ，その時の総データ転送量を評価した．メモリブロックの確保方法を図 4.7 に，結果を図 4.8 に示す．hotspot で利用するメモリブロックのサイズは次の 3 通りである．袖領域 2 つ分 $(N^2/16 + N * 2)$ を含んだメタ VS セグメントを格納するメモリブロック，袖領域 1 つ分 $(N^2/16 + N)$ を含んだメタ VS セグメントを格納するメモリブロック，そして袖領域を含まない $(N^2/16)$ メタ VS セグメントを格納するメモリブロックである．それぞれのメタ VS セグメントとその読み書きモードの数を表 4.5 に示す．

メモリブロック数はマルチ GPU 環境において性能に影響を与えることがわかる．‘960+950’ では ‘7,1,7’ が最も良い結果であり，提案手法は十分に性能を引き出せていない．原因として，タスクスケジューリングの結果，一度も使われないメモリブロックが発生する事が考えられる．今回のケースでは ‘960’ と ‘950’ それぞれに 8 タスクずつスケジューリングされており，$N^2/16 + N$ のメモリブロックは各デバイスで 1 つしか使われていなかった．‘7,1,7’ では該当メモリブロックを他のメモリブロックに使うことで全てのメモリブロックを使用している．前述の srad_v2・k80(2) のケースや ‘2,1,12’ の性能が大幅に悪化するのも同じ理由と考えられる．

評価プログラムにおいては，各メモリブロック数を必要数/デバイス数-α(α：デバイスの性能差に応じて決定) で一様に確保し，その後メモリの許す範囲で提案手法のように READWRITE などの属性に応じて追加するといった方法で解決できると考える．しかしながら，一部のメモリブロックの使用頻度が高いプログラムではプログラム全体を通した各メモリブロックの使用頻度などを考慮する必要がある．よってメモリブロック数の決定方法は引き続き検討が必要である．

表 4.5: メタ VS セグメント数

	READWRITE	WRITE	READ
$N^2/16 + N * 2$	14	0	0
$N^2/16 + N$	2	0	0
$N^2/16$	16	0	16

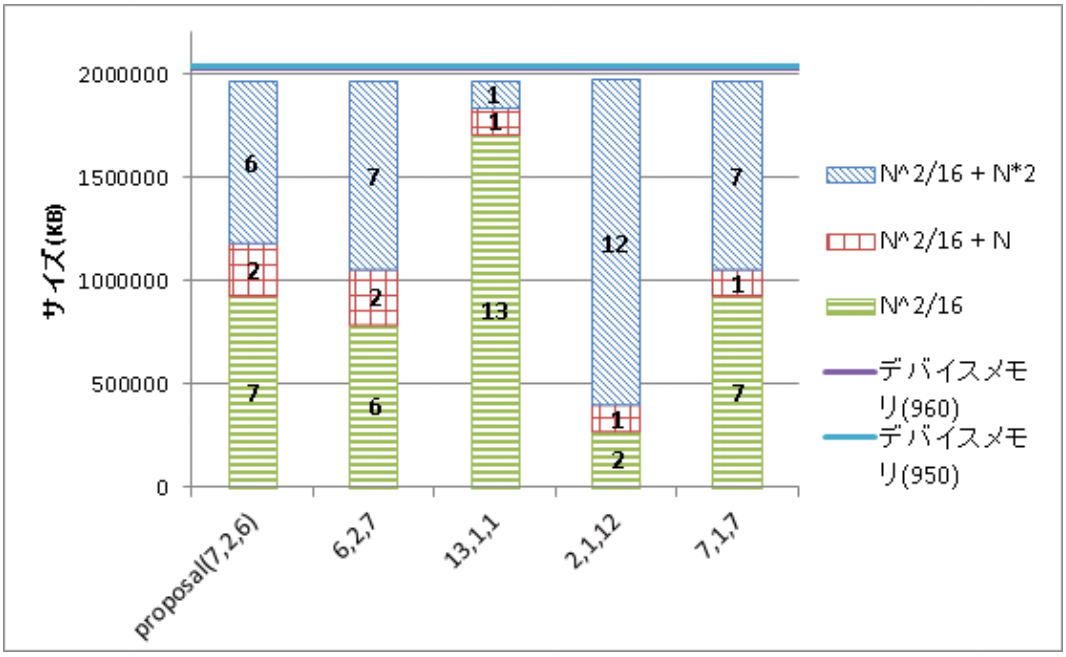

図 4.7: メモリブロックの確保方法

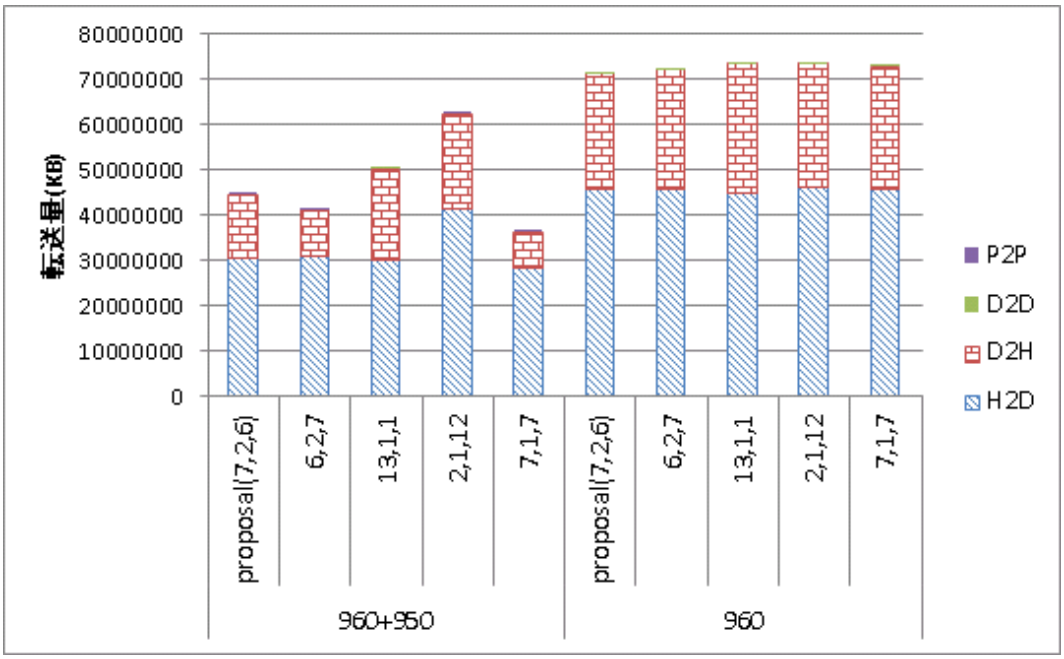

図 4.8: メモリブロック数による総データ転送量の変化

5　関連研究

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，StarPU[13], GMAC[14], GDM[15] などがある．StarPU は CPU/GPU のヘテロジニアスコンピューティング用のタスクプログラミングライブラリである．StarPU ではユーザがタスクグラフや配列の分割を用意された API を用いて記述することで，実行時のタスクスケジューリングやメモリ転送などを自動で行う．GMAC は ADSM と呼ばれる非対称分散共有メモリを採用したライブラリであり，CPU/GPU 両方からアクセスできる単一ポインタを利用してプログラムを記述できる．メモリ転送はページフォルトを監視することで動的に行われる．また，GDM も同様にページフォルトを利用して動的にメモリ管理を行う．これらの研究と MESI-CUDA を比較すると，ユーザの記述を必要としない点や静的解析を行うことにより事前に転送すべきデータを把握できるといった点で異なる．StarPU ではタスクの作成や依存関係を表すグラフの作成，またタスクがアクセスする部分配列をユーザが分割する必要がある．適切なタスク分割の粒度は実行環境によって変わる．これに伴いデータ配列の分割も変わるため，プログラム中で分割を記述する点で移植性に欠ける．GMAC, GDM ではデータ転送をページフォルトをトリガーに行う．また，一貫性の管理は一定のブロックサイズで行われる．MESI-CUDA は必要なデータ転送を事前に把握できるため，ページフォルトを監視するオーバーヘッドが発生しない．さらに，一貫性の管理もプログラムで実行するタスクのアクセス範囲により分割した VS セグメント単位で行うため，小さいオーバーヘッドで実現できる．

6　おわりに

本研究ではマルチ GPU 上の効率的な実行手法のためのランタイムを提案した．評価の結果，提案手法はタスク間のアクセス範囲に重複が発生する 2 つのプログラムで，重複部分を把握し一貫性を管理することで必要なデータ転送を削減できた．今後の課題として，適切なタスク数の決定や確保するメモリブロック数についての更なる考察，手法の確立などがあげられる．

謝辞

本研究を行うにあたり，御指導，御助言頂きました大野和彦講師，並びに多くの助言を頂きました山田俊行講師に深く感謝致します. また，様々な局面にてお世話になりました研究室の皆様にも心より感謝いたします.

参考文献

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

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

[3] OpenMP, `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] 田中 宏明，山本 怜 and 大野 和彦: *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．2016-HPC-155(8), 1-7, 2016.

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

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

[13] 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.

[14] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W. W. Hwu. *An asymmetric distributed shared memory model for heterogeneous parallel systems.* ASPLOS XV Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems, pages 347-358, 2010.

[15] 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.