卒業論文

題目

マルチホストに対応した MESI-CUDA の ランタイムスケジューラ

指導教員

大野和彦　講師

2016 年

三重大学 工学部 情報工学科
コンピュータソフトウェア研究室

今泉秀章 (412810)

内容梗概

近年，気象予測や遺伝子解析といった様々な分野において大規模計算の需要が高まっており，CPU より性能向上がめざましい GPU を汎用計算に用いる GPGPU が注目されている．開発環境として CUDA[1] などの処理系が提供されているが，GPU アーキテクチャに合わせた低レベルなコーディングを必要とし，プログラミングが困難という問題がある．そこで，本研究室ではデータ転送コードを自動生成し，自動最適化をおこなう MESI-CUDA[2] を開発している．

一度に複数の GPU を用いるマルチ GPU により，単一 GPU での計算より実行時間を短縮することができるが，単一のコンピュータ (シングルホスト) では搭載できる GPU ボードが 1〜4 枚に制限される．複数のコンピュータ (マルチホスト) を用いることで，シングルホストでは実現できない枚数の GPU ボードによる並列処理が可能になり，より高い性能を実現できる．しかし，現行の MESI-CUDA 処理系は，マルチホストに対応していない．そこで，MESI-CUDA のランタイムスケジューラ [3] にマルチホスト対応のための機構を設計・実装した．

ホスト間におけるデータ転送のオーバーヘッド改善のために，擬似キャッシュ機能を実装した. これにより，ホスト間通信のオーバーヘッドを削減することができた.

Abstract

the performance of Graphics Processing Units (GPU) is improving rapidly. Thus, General Purpose computation on Graphics Processing Units (GPGPU) is expected as an important method for high-performance computing. Although programming frameworks, such as CUDA, are provided, they require explicit specification of memory allocations and data transfers. Therefore, we are developing a new programming framework MESI-CUDA, which hides such loe-level description from the user. GPGPU in a multi-host can achieve higher performance than GPGPU in a single-host because it is possible to parallel processing by the GPU board of the number that can not be achieved in single-host. But, Current MESI-CUDA processing system does not correspond to the multi-host. In this paper, runtime scheduler of MESI-CUDA correspond to the multi-host. As a result of adding the pseudo-cache function, it was possible to reduce the overhead of between hosts communication.

目 次

図 目 次

表 目 次

1　はじめに

1.1　研究目的

近年，気象予測や遺伝子解析といった様々な分野において大規模計算の需要が高まっており，CPU より性能向上がめざましい GPU を汎用計算に用いる GPGPU が注目されている．開発環境として CUDA[1] などの処理系が提供されているが，GPU アーキテクチャに合わせた低レベルなコーディングを必要とし，プログラミングが困難という問題がある．そこで，本研究室ではデータ転送コードを自動生成し，自動最適化をおこなう MESI-CUDA[2] を開発している．

また，一度に複数の GPU を用いるマルチ GPU により，単一 GPU での計算より実行時間を短縮することができるが，単一のコンピュータ (シングルホスト) では搭載できる GPU ボードが 1〜4 枚に制限される．複数のコンピュータ (マルチホスト) を用いることで，シングルホストでは実現できない枚数の GPU ボードによる並列処理が可能になり，より高い性能を実現できる．

現行の MESI-CUDA 処理系は，マルチホストに対応していない．そこで，MESI-CUDA ランタイムスケジューラ [3] にマルチホスト対応のための機構を設計・実装する．

1.2 論文の構成

2章では背景としてCUDA，マルチGPU，マルチホストにおけるマルチGPU，MESI-CUDAについて解説する．3章ではマルチホスト環境への対応として，クラス設計と擬似キャッシュ機能について述べる．4章ではマルチホスト環境とシングルホスト環境におけるシングルGPU，マルチGPUの3つの環境から評価プログラムの実行結果を比較し，マルチホスト対応の性能を評価する．また，擬似キャッシュ機能有りと無しの環境におけるプログラムの実行結果を比較し，擬似キャッシュ機能の評価を行う．最後に5章でまとめを行う．

2　背景

2.1　GPU アーキテクチャ

GPU の基本的なアーキテクチャは，多数のコアがグローバルメモリを共有している構造である．しかし，メモリは複雑に階層化されており，それぞれの用途ごとに使い分ける必要がある．また，コアは一定数でまとめられており，そのユニットごとにレジスタやシェアードメモリ，ローカルメモリを有する．fig.2.1 に GPU のアーキテクチャモデルを示す．

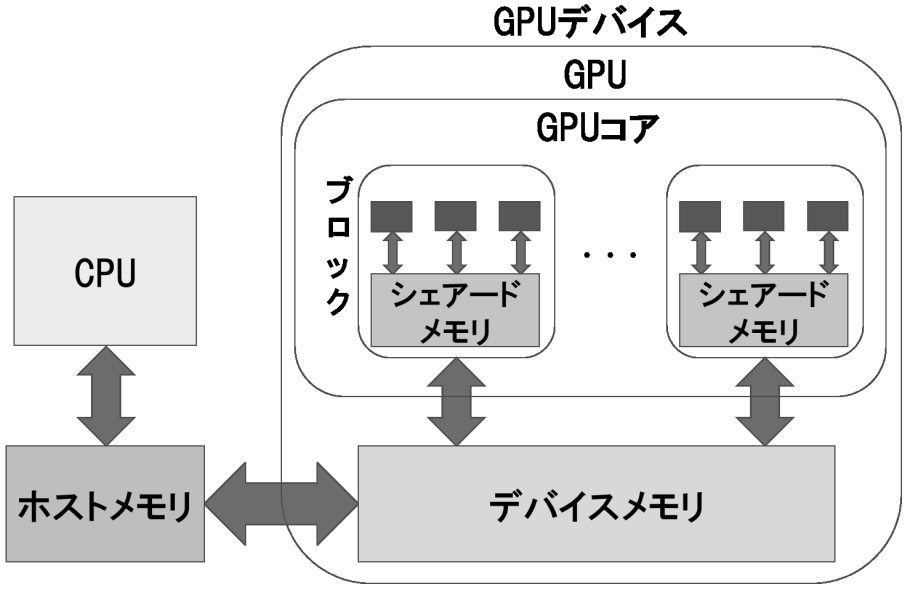

fig . 2.1: GPU のアーキテクチャモデル

2.2 CUDA

2.2.1 概要

CUDA は NVIDIA 社より提供されている GPGPU 用の SDK であり，ユーザは C/C++を拡張した文法とライブラリ関数を用いて GPGPU プログラムを容易に開発することができる．CPU，GPU をそれぞれホスト，デバイスと呼び，各々がアクセスするメモリをそれぞれホストメモリ，デバイスメモリと呼ぶ．デバイス上で実行する関数をカーネル関数と呼び，デバイスメモリの確保・解放，及びホスト・デバイス間のメモリデータのデータ転送はライブラリ関数呼び出しで行うことができる．

CUDA におけるデータ転送の種類はホストからデバイスへのデータ転送を行う download 転送と、デバイスからホストへのデータ転送を行う readback 転送の 2 種類がある．

2.3 マルチ GPU

2.3.1 概要

単一のデバイスが動作するシングル GPU に対し，複数のデバイスを並列動作させることができるマルチ GPU は，一度に複数の GPU 処理を実行できる．マルチ GPU プログラミングを行うには明示的にライブラリ関

数を用いて，使用する GPU を指定してから GPU 処理を行わせる必要がある．

2.3.2 マルチホストにおけるマルチ GPU

マルチホストでマルチ GPU を行う方法として，任意の計算機 1 台をマスターホストに割り当てる必要がある．マスターホストが，自身の持つ GPU とネットワークで繋がった単一もしくは複数の計算機 (リモートホスト) が持つ GPU を利用することでマルチホストでのマルチ GPU を実現することができる．fig2.2 に図示する．

しかし，CUDA はホスト間通信などのライブラリ関数を用意していない．このため，ソケットや MPI などホスト間通信用のライブラリと組み合わせる必要があり，プログラミングがより困難になる．

PCI Express 経由でホスト・デバイス間のデータ転送を行うシングルホスト (マルチホストにおいてはマスターホスト) に対して，リモートホスト上の GPU を利用するためには，マスター・リモートホスト間でのホスト間通信においてネットワーク通信を利用して転送しなくてはならない．PCI Express に比べて，ネットワーク通信 (Ethernet) はデータ転送速度が遅く，ネットワーク通信のオーバーヘッドを考慮したデータ転送が必要である．fig2.3 にホスト・デバイス間およびホスト間データ転送の流れ

を示す.

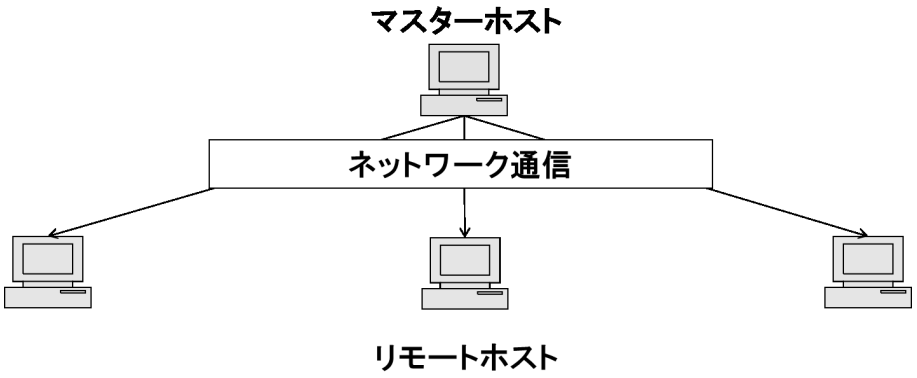

fig . 2.2: マルチホストにおけるマルチ GPU

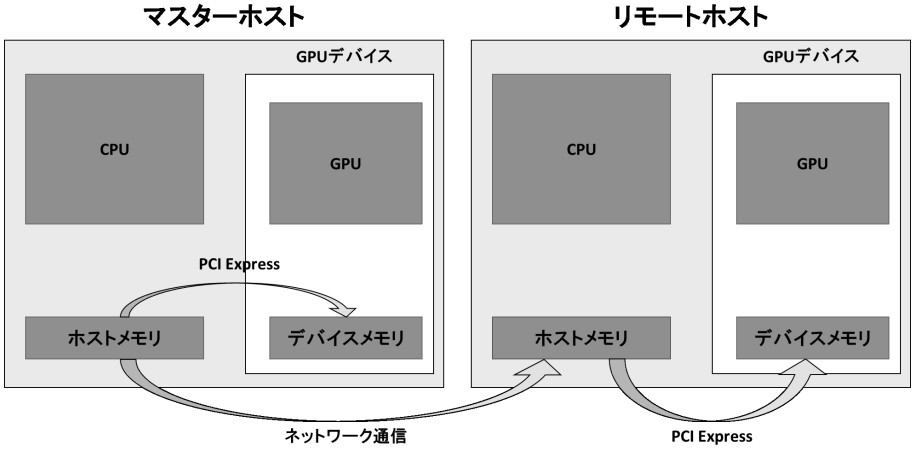

fig . 2.3: ホスト・デバイス間およびホスト間データ転送の流れ

2.4 MESI-CUDA

2.4.1 概要

MESI-CUDA は CUDA による GPGPU プログラミングを容易にするためのフレームワークである．データ転送やメモリ確保・解放などのコードを自動的に生成することで，ユーザの負担を軽減させる．ホストとデバイスへの処理の振り分けやカーネル関数の記述はユーザ自身が従来の CUDA に準じる形でコーティングを行う．

CUDA では fig.2.1 の GPU アーキテクチャをそのままプログラミングモデルとして用いる．これに対し，MESI-CUDA では fig.2.4 に示すプログラミングモデルを用いている．CUDA ではホストメモリ・デバイスメモリを意識してプログラミングする必要があったが，MESI-CUDA では１つの共有メモリに見せかけている．よってホスト関数・カーネル関数の違いによる変数の使い分けや，メモリ確保・解放，データ転送などの記述が不要になる．

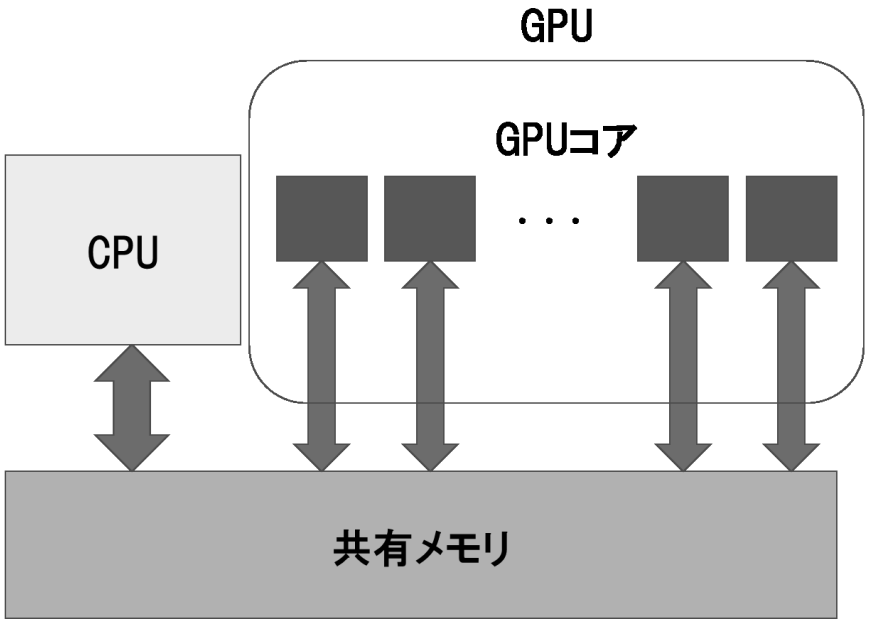

fig . 2.4: MESI-CUDA のプログラミングモデル

2.4.2 ランタイムスケジューラ

現行の MESI-CUDA にはランタイムライブラリによるスケジューリング機構が搭載されている．この機構は GPU 処理をジョブとタスクに分割し，タスクを各 GPU に振り分けて処理させる．ジョブはホスト・デバイスメモリ間のデータ転送とカーネル実行をまとめたものである．タスクはジョブを任意のブロック数で分割したものであり，実際に実行する処理の最小単位である．fig.2.5 にランタイムスケジューラの処理を図示する．

現行の MESI-CUDA ランタイムスケジューラはマルチホストに対応し

ていない．ランタイムスケジューラをマルチホストに対応させることにより，ユーザーは新たにホスト間通信のためのコーディングをすることなく，マルチホストに対応したMESI-CUDAを扱うことが可能になる．

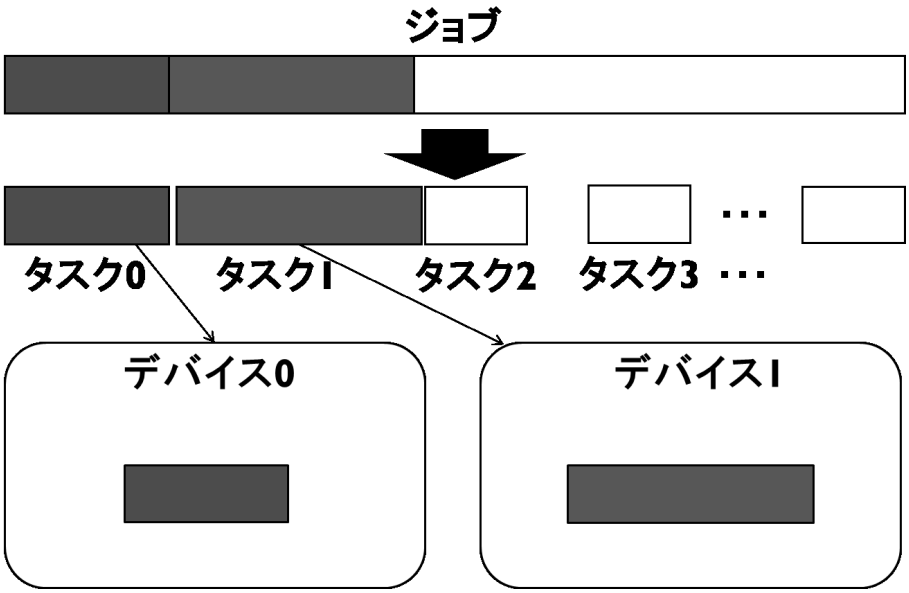

fig . 2.5: ランタイムスケジューラの処理

3　マルチホスト環境への対応

3.1　現行の問題点

現行の問題点としては前述したようにMESI-CUDAがマルチホストに対応していない事である．実際にタスクを各GPUに振り分けるのはランタイムスケジューラであることから，ランタイムスケジューラがタスクをリモートホストの各GPUに振り分けられるようにすれば良い．リモートホストへタスクを振り分けるにはホスト間通信を必要とするため，ホスト間通信用のライブラリを導入しなければならない．ホスト間通信用のライブラリを利用することで，スケジューラでのデータ転送においてマスター・リモートホストを区別する必要があり，マスター・リモートホストに応じたデータ転送の内容変更やホスト間通信の最適化の際にも内容の変更を必要とする．

3.2　概要

今回のマルチホストのためのランタイムスケジューラのクラス設計の説明をする．

`Master`はランタイムスケジューラにおいて中核となり，タスクの生成を行い，タスク処理を`GPUController`に要求する．`GPUController`は

Master から要求されたタスク処理を GPU 上で実行する．この二つのクラスによってマスターホスト上の GPU を扱うことができる．実際にタスク処理を行うクラスは GPUController なので，リモートホスト上の GPU を利用するには，タスクの処理要求をリモートホスト上の GPUController へと中継するクラスが必要である．

そこで，本研究ではその中継を行うクラスとして ProxyGPUController と Relay を設計・実装した．ProxyGPUController はマスターホスト上に存在し，GPUController と同じように Master からタスク処理を要求される．ただし，自身はタスク処理を実行せず，リモートホスト上の中継クラス Relay へ，タスク処理に必要なデータを送信する．Relay はデータの受信後，そのホスト上の GPUController へ処理を要求し，タスクを処理し終わった後に，計算結果をマスターホスト側へ送信する．ProxyGPUController と Relay はホスト間通信を隠し透過的に通信を行うため，Master はローカルホスト・リモートホストを区別する必要はない．また，ホスト間通信に関する最適化手法を，他のクラスの変更なしに実装できる．

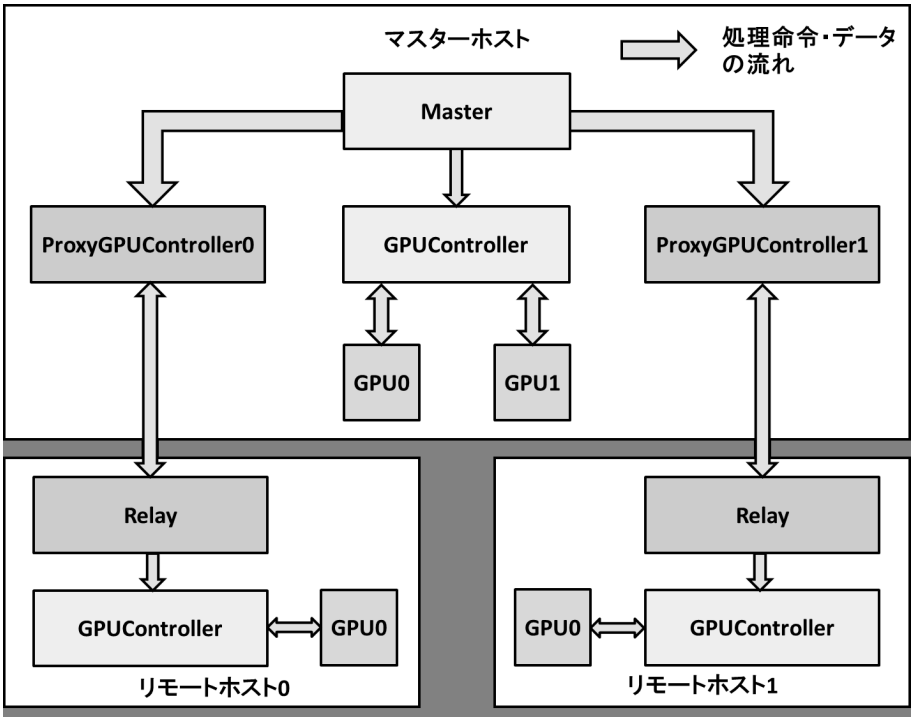

fig . 3.6: ランタイムスケジューラクラスモデル

3.3 ProxyGPUControllerとRelayの実装

ProxyGPUControllerの実装に関して，Masterから受け取る命令はGPUControllerと変わらないため，GPUControllerを継承したクラスとなっている．

前述したとおり，実際には自身はタスク処理を実行しないので，リモートホストへ処理を中継する必要がある．しかしProxyGPUControllerとRelay間では，実際の処理命令を共有するものがないので実装する必要がある．そこで，processIdを実装した．processIdはProxyGPUControllerとRelay間でMasterからの命令を識別する番号となっている．ProxyGPUController

はMasterからの処理命令を受け取り，その処理命令に対応するprocessId と命令処理に必要なデータをRelayに送信する．Relayは受け取ったprocessId に対応する命令をリモートホスト上のGPUControllerに送ることで，リモートホスト上のGPU処理を可能とした．

3.4 擬似キャッシュ機能

リモートホスト使用時のオーバーヘッド軽減のためには，ネットワーク通信によるデータ転送をなるべく行わないことが重要である．計算によっては，違うタスク同士で部分的に，または，まったく同じ範囲の配列の領域を計算に使用する場合がある．この場合の配列の領域とは，ジョブ・タスクの処理に使用される配列を，ランタイムスケジューラがジョブからタスクに分割する際，タスクで使用する範囲に分割し，各タスクに割り当てるものである．サイズは固定長である．3.7に簡単な例を示す．この場合に着目し，重複する領域を2回以上送信しないようにすれば無駄なネットワーク通信が減る．

そこで，擬似キャッシュ機能を追加した．領域が送信済みであることを示すフラグを持つキャッシュテーブルをマスターホスト上に設ける．リモートホストへタスクを送信する際に，タスクの持っている領域とその

領域に該当するテーブルのフラグを参照する．該当フラグが真の場合は配列を送信せず，偽の場合は送信し，フラグを真にする．

また，タスクによっては，先行するタスクが更新した配列を使って計算を行うことがある．その場合，上記のようなキャッシュでは，テーブルを参照した結果，更新後の配列が送信されない可能性がある．そこで領域が更新されたことを示す更新フラグをキャッシュテーブルに追加した．計算結果を受信した際に，テーブルの該当するフラグを真にする．キャッシュテーブル参照の際に該当領域の更新フラグが真かつ送信フラグがならば，送信済みのフラグが真の場合でも送信を行い，その後，フラグを偽にする．

擬似キャッシュテーブルを用いる一連の流れのフローチャートを fig. 3.8 に示す．

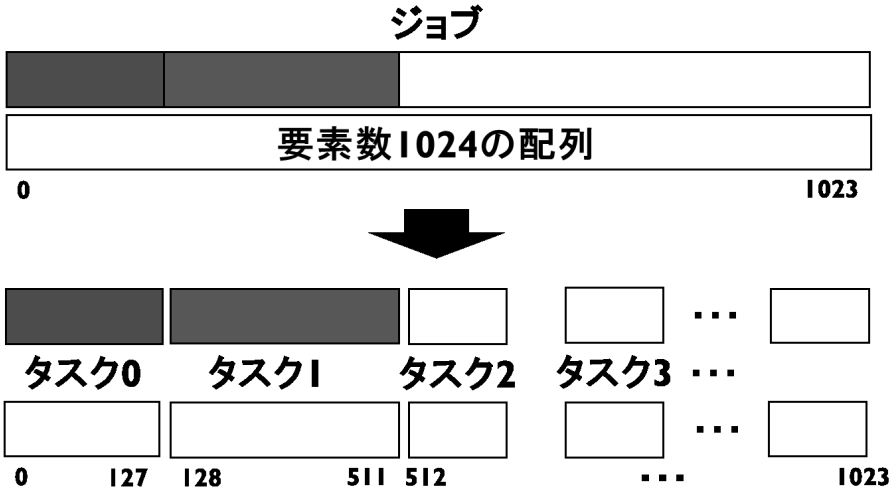

fig . 3.7: 配列領域の例

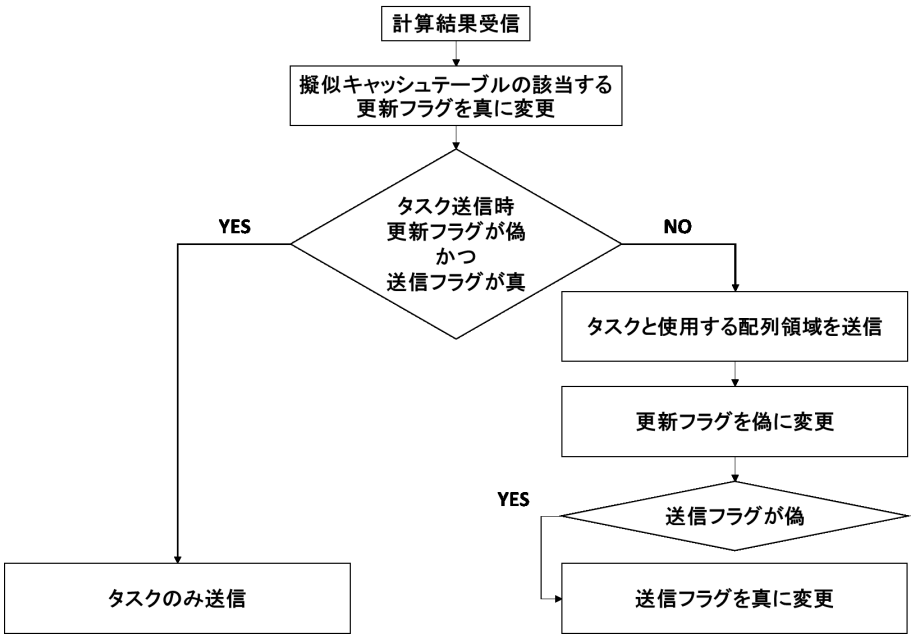

fig . 3.8: 擬似キャッシュテーブルを用いる一連の流れ

4　評価

予備実験として，以下のような 3 つの環境でスケジューラの性能評価を行った．

1. シングルホスト・シングル GPU(S・S)

2. シングルホスト・マルチ GPU(S・M)

3. マルチホスト

具体的な評価環境を Table4.1 に示す．メモリはすべて 16GB である．

評価プログラムは 4096 × 4096 の正方行列の行列積，4096 × 4096，10 ステップの hotspot の 2 つのプログラムを用いた．結果を Table4.2，4.3 に示す．

表 4.1: 評価環境

	シングルホスト		マルチホスト (環境 3)	
	シングル GPU(環境 1)	マルチ GPU(環境 2)	マスターホスト	リモートホスト
CPU	Xeon E5-1620 0 3.6GHz		Xeon E5-1620 0 3.6GHz	Xeon E5-2620 v2 2.10GHz
GPU ボード	GeForce GTX 980	GeForce GTX 980,GeForce GTX 750 Ti	GeForce GTX 980	GeForce GTX 750 Ti

表 4.2: シングル/マルチホストでの実行時間

実行環境	行列積 (s)	hotspot(s)
S・S	1.58	1.21
S・M	0.92	0.47
マルチホスト	1.28	2.58

表 4.3: 擬似キャッシュ機能の有/無での実行時間

実行環境	行列積 (s)	hotspot(s)
擬似キャッシュ機能無し	1.89	3.76
擬似キャッシュ機能有り	1.28	2.58

Table4.2 により，行列積の場合，マルチホストは S・S より約 19%速くなっていることが分かる．一方，S・M は S・S より約 41%速くなっていることが分かる．この差の原因は，ネットワーク通信のオーバーヘッドの影響である．S・S と S・M の実行時間差の比率から 45%のオーバーヘッドが発生する．hotspot の場合，マルチホストは S・S より約 113%遅くなっている．行列積と同じ条件でネットワーク通信のオーバーヘッドを求め

るとの約 285% のオーバーヘッドが発生する．この原因は，現状では擬似キャッシュ機能を考慮しないタスクの割り振りを行っているため，キャッシュがなされなかった配列領域が多かったと考えられる．また，繰り返し計算を行うために，1 ステップ毎に更新された配列領域が発生し，リモートホストへ転送しなければならないため，ネットワーク通信のオーバーヘッドが発生したことも実行時間に影響したと考えられる．これらの影響がネットワーク通信のオーバーヘッドを増加させた結果，S・S 及び S・M よりも実行時間が大幅に遅くなった原因と考える．

Table4.3 より，行列積，hotspot 共に擬似キャッシュ有りの実行時間が擬似キャッシュ無しより約 30%速くなっていることが分かる．

以上の結果より，一部のプログラムではマルチホストが S・S より速くなったことが分かった．また，擬似キャッシュ機能を追加した結果，追加しない場合よりも速くなり，ネットワーク通信のオーバーヘッドを削減することができた．

5　おわりに

本研究では，MESI-CUDA のランタイムスケジューラにリモートホスト上の GPU を扱う機構を設計・実装し評価を行った．その結果，擬似キャッシュテーブルを用いることによって，ホスト間通信における実行時間のオーバーヘッドを改善できた．今後は，タスク分配方法の改善として，依存性のあるタスク同士を違うホストに送信することを回避する，同じ配列領域を使用するタスクを同じホストに送ることで擬似キャッシュ機能を活用するなど，ホスト間通信のオーバーヘッドのさらなる改善と高速化を行っていく．

謝辞

本研究を遂行するにあたり，日頃からご指導，ご助言をいただきました大野和彦講師に感謝いたします．また，お世話になった山田俊行講師，コンピュータソフトウェア研究室の方々に感謝の意を表します．

参考文献

[1] NVIDIA Developer Zone,https://developer.nvidia.com/category/zone/cuda-zone

[2] 丸山剛寛，田中宏明，水谷洋輔，神谷智晴，大野和彦.
GPGPU フレームワーク MESI-CUDA におけるマルチ GPU へのスレッドマッピング機構.
情報処理学会研究報告 2014-HPC-145(44),1-8,(2014)

[3] Kazuhiko Ohno and Rei Yamamoto.
Dynamic Task Scheduling Scheme for a GPGPU ProgrammingFramework.
CANDAR2015, pp. 181-187,(2015)