修士論文

題目

計算量非均質なGPUスレッドの
スケジューリング最適化

指導教員

大野　和彦　講師

2018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

古居　鉄平 (416M519)

内容梗概

高性能計算の研究が現在，世界的に行われており，画像処理に用いられる GPU を汎用計算に用いる GPGPU が注目されている．GPGPU の現在主流な開発環境である CUDA はハードウェアのメモリ構造など低レベルな知識が必要となるため，コーディング難易度が高く，記述を簡易化するフレームワークや自動最適化に関しての研究が行われている．GPU は複数のデータに対して同じ命令を実行する SIMD 型実行を行なっている．SIMD 型実行は大量のデータに対し効率的に並列処理を行うことが可能である．しかし，GPU で実行する関数である GPU カーネルに条件分岐が含まれている場合に同時実行されるスレッドの実行パスが異なってしまう．実行パスが異なることで同時実行されるスレッドのうち一部がアイドルとなる時間が発生し，実行効率が低下してしまう分岐ダイバージェンスという問題がある．本稿では，上記の問題を軽減するスケジューリング手法を提案する．同時実行されるスレッド間の計算量や実行パスの違いを本スケジューリングによって抑えることで実行効率の改善を図る．そのため本研究では入力データによる計算量の非均質性や実行パスの分散を調べる．次に調べた結果をもとに各スレッドに対応付けられた入力データを変更する．しかし入力データの並べ替えはコストが高い．そのためスレッド ID ソートテーブルというデータアクセステーブルを作成し，各スレッドのデータアクセス時にスレッド ID ソートテーブルを経由することで各スレッドのデータとの対応付けを変更する．ランダムなデータ長に計算量が依存するプログラムに提案手法を適用した結果，最大で 51%の実行時間短縮が見られた．

Abstract

Research on high-performance computing is actively. Among them, GPGPU which parallelly executes general purpose computation making use of the parallelism possessed by GPU has drawn attention in the field of HPC. CUDA, which is the current mainstream development environment of GPGPU, requires low-level knowledge such as hardware memory structure. For this reason, simplification framework and automatic optimization research are being conducted. The GPU performs SIMD type execution. SIMD type execution can efficiently perform parallel processing on a large amount of data. However, when a conditional branch is included in the GPU kernel which is a function to be executed by the GPU, the execution path of the concurrently executed thread differs. A part of the concurrently executed threads becomes idle due to the execution path being different This problem is called branching divergence. In this paper, we propose a method to optimize scheduling This scheduling improves execution efficiency by suppressing the difference in the amount of computation and execution path between concurrently executed threads. For this reason, we investigate heterogeneity of computational complexity and variance of execution path by input data. Next, based on the examined result, the correspondence of input data of each thread is changed. However, sorting input data is expensive. Therefore, we create a data access table called a thread ID sort table We change the correspondence with the data of each thread by passing through the thread ID sort table at the time of data access of each thread. As a result of applying the proposed method to a program whose computational amount depends on random data length, the execution time is reduced by 51% at the maximum.

目 次

図 目 次

表 目 次

1　はじめに

近年 , GPU(Graphics Processing Unit) は CPU に比べめざましい演算性能の向上を見せている．そのため , GPU を本来の用途である画像処理以外の汎用的な計算に用いる GPGPU(General Purpose computation on GPUs) はその計算性能により様々な分野から注目を集めている．現状利用されている GPGPU 開発環境として CUDA や OpenCL が存在する. しかし, これらの開発環境は GPU の性能を引き出す上で低レベルのコーディングを要求するため, ユーザの負担が大きい問題がある. また,CPU(ホスト)・GPU(デバイス) はそれぞれ自身のみがアクセスできるホストメモリ・デバイスメモリを搭載する. ユーザはデバイスを扱う上でホストメモリ・デバイスメモリ間のデータ転送をプログラム中に記述する必要があり, ハードウェア構成を意識したプログラミングが求められる. そのため, 記述を簡易化するフレームワーク [1] [2] や自動最適化 [3] [4] に関しての研究が行われている

GPU ではコア数を超えるスレッドを生成し，各スレッドが同一命令をそれぞれスレッドに割り当てられたデータで実行する SIMD 型実行で並列処理を行う．このとき同時実行するスレッドの実行パスが異なる場合，実行効率が低下してしまう現象が発生する．この現象を分岐ダイバージェ

ンスと呼ぶ．同時実行するスレッドの実行パスが異なる条件はデバイス側で実行する命令の中に条件分岐が含まれており，かつ同時実行するスレッドのなかで条件分岐の真偽が異なる場合となる．条件分岐の真偽が異なる原因は殆どの場合がスレッドに割り当てられたデータによるものである．

そこで本稿ではデータが割り当てられたスレッドを実行時に適切にスケジューリングすることで，分岐ダイバージェンスの影響を抑える．本スケジューリング手法の狙いは同時実行されるスレッド間の計算量や実行パスの違いを抑えることである．これにより分岐ダイバージェンスによる実行効率の低下を軽減する．GPGPU プログラムを実行する際に GPU カーネル内の処理からあらかじめ各スレッドの計算量や実行パスを解析する．調査結果を元にスレッドのスケジューリングを行う．ランダムなデータ長に計算量が依存するプログラムに提案手法を適用し，適用しない場合と比較を行った．

以下，2 章では背景として GPU，CUDA，分岐ダイバージェンス について解説する．3 章では本研究で提案するスケジューリング手法と適用例を示す．4 章では提案手法の有無による CUDA プログラムの実行時間を比較した結果を示し，5 章で関連研究を紹介する．最後に，6 章でまとめ

を行う

2　背景

2.1　GPU

GPU [5] は演算を行うコアを大量に搭載し多数の処理を並列に実行できる．GPU ではコア数を超えるスレッドを生成でき，これらの大量のスレッドは 32 スレッド単位で分割され管理・実行される．この 32 スレッドのグループをワープという．ワープ内の 32 スレッドは同時に同じ命令を実行する SIMD 型の並列処理を行う．

ワープ内の各スレッドがアクセス命令を実行するとき，メモリ上で連続したアドレスへのアクセスは高速になる．GPU はキャッシュを搭載した階層型のメモリアーキテクチャを採用しており，GPU の主記憶であるデバイスメモリへのアクセスは 2 次キャッシュのラインサイズである 128byte 単位で行われる．ワープ内のスレッドが同時に同一キャッシュライン上のデータにアクセスすれば，複数のデータ転送を一度のデバイスメモリへのアクセスで行える．このようなアクセスをコアレッシングアクセスという．また, 同一ライン内のデータに対して時間的局所性のあるアクセスを行えば，キャッシュメモリ上にデータが存在するので高速にアクセスできる.

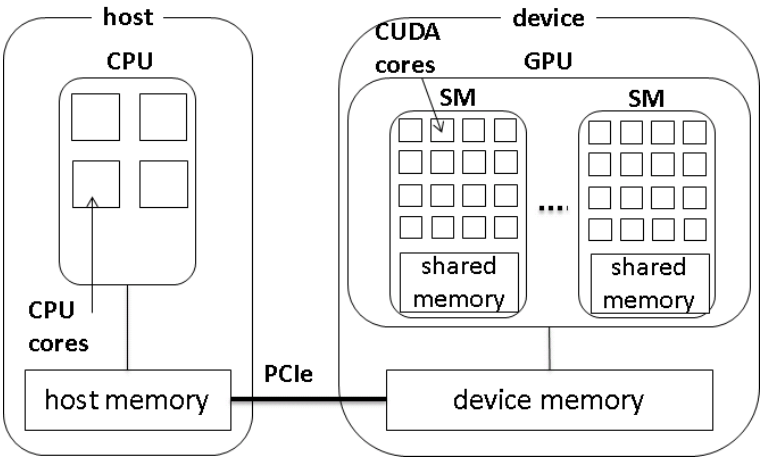

図 2.1: GPU アーキテクチャ

2.2 CUDA

CUDA [6] は nVIDIA 社が提供するコンパイラ・ライブラリを含めた GPGPU 統合開発環境であり, 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 CUDA プログラムを開発する.CUDA において, CPU 側はホスト, GPU 側はデバイスと呼ぶ. デバイスは PCI-Express を通じてホストにより制御され, ホストから与えられる処理を数千個の CUDA コアで並列実行する. ホスト・デバイスの各 CPU コア・CUDA コアは図 2.1 に示すように, 自身が接続するホストメモリ・デバイスメモリにのみそれぞれアクセスする. ホストメモリ・デバイスメモリ間のデータ転送はユーザ自身が CUDA ライブラリ関数を用いて記述する必要がある.

2.3　分岐ダイバージェンス

各スレッドの実行パスが分岐処理により異なる場合, ワープは分岐部分のそれぞれのパスを逐次実行する. 例えば, ワープ内のスレッドが if-else 文により 2 通りの実行パスに分かれた場合，ワープでは if 節のパスの各命令を実行した後に,else 節のパスの各命令を実行する．分岐内の片側のパスを実行中, そのパスを実行しないスレッドに対応するコアはアイドル状態となる．この現象を分岐ダイバージェンスと言う．分岐ダイバージェンスによって引き起こされるアイドルコアの増加は性能低下の要因になる.

図 2.2 では分岐ダイバージェンスの発生による実行効率低下の様子を示している．この図は GPU カーネルで if-else 文を利用した場合の実行の一例である．同一ワープ内の各スレッドの実行パスがすべて同じ場合，実行しない if, 又は else 部分はすべて省略される．しかしワープ内のスレッドのうち 1 つでも違う実行パスだった場合は if-else 文全体を実行するため，各スレッドにおいてアイドルとなる時間が発生し，実行効率が低下してしまう．GPGPU プログラムでは各スレッドに対応付けられたデータが GPU カーネル内の分岐に関わることが多く，分岐ダイバージェンスの発生と深く関わっていることが多い.

本稿では入力データによって引き起こされた分岐ダイバージェンスに

よる性能低下を防ぐために，スケジューリングの最適化を行う．

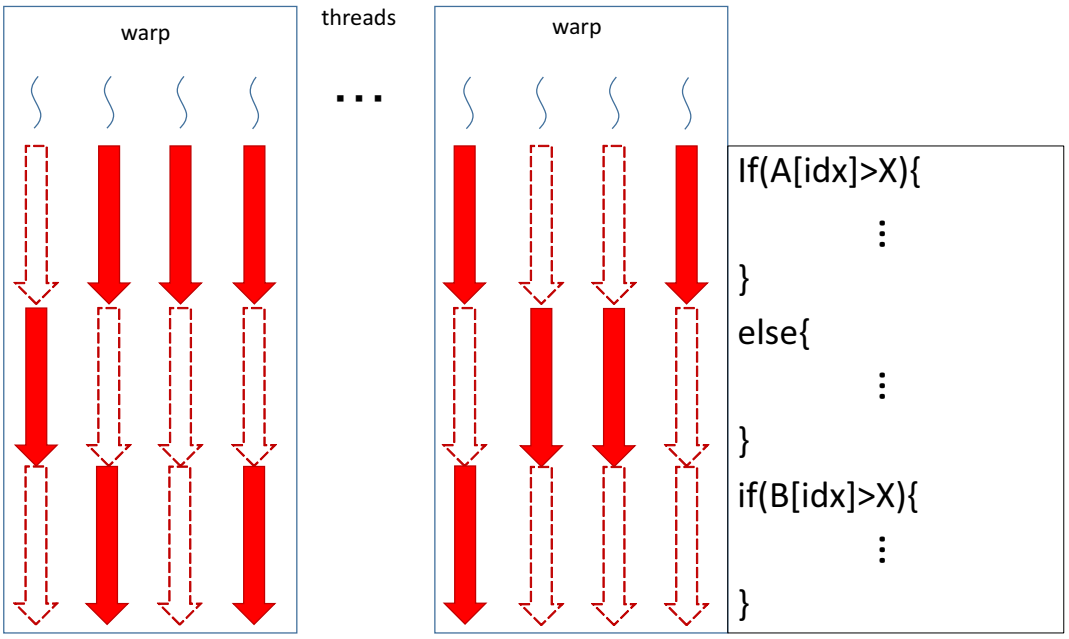

図 2.2: 分岐ダイバージェンスの発生例

3　提案手法

3.1　基本方針

処理量の違いや条件分岐によるアイドル時間を削減するようにスレッドスケジューリングを最適化する．計算量非均質な問題の場合，同一ワープ内の各スレッドの実行パスが異なるため，同時実行時に実行効率が低下する．この問題の改善のためには同一ワープ内のスレッドの実行パスをできる限り同一にする必要がある．今回のスケジューリングの基本方針は同一ワープ内の各スレッドの実行パスを可能な限り同一にすることである．

同一ワープ内の各スレッドの実行パスを可能な限り同一にするために，各スレッドを実行パスによってグルーピングする．各スレッドのグルーピング結果をもとに同一ワープ内の各スレッドの実行パスが均質になるようにスケジューリングすることで分岐ダイバージェンスの影響を低減する．

図 3.3 は提案手法による分岐ダイバージェンスの改善の様子を示した図である．for ループにネストされた if-else 文を実行した際の各スレッドの実行状態を表している．図形の実線矢印が実行されている時間，破線矢印がアイドル時間となっている．適用前は同一ワープ内の実行パスがバラ

バラになっているため，各スレッドに破線矢印のアイドルとなる時間が存在してしまう．適用後は各スレッドを割り当てられたデータをもとにグルーピングしたことにより, ワープ内のスレッド間の実行パスの統一が行われる．実行パスの統一により各スレッドのアイドルとなる時間が削減され，実行効率が改善される．

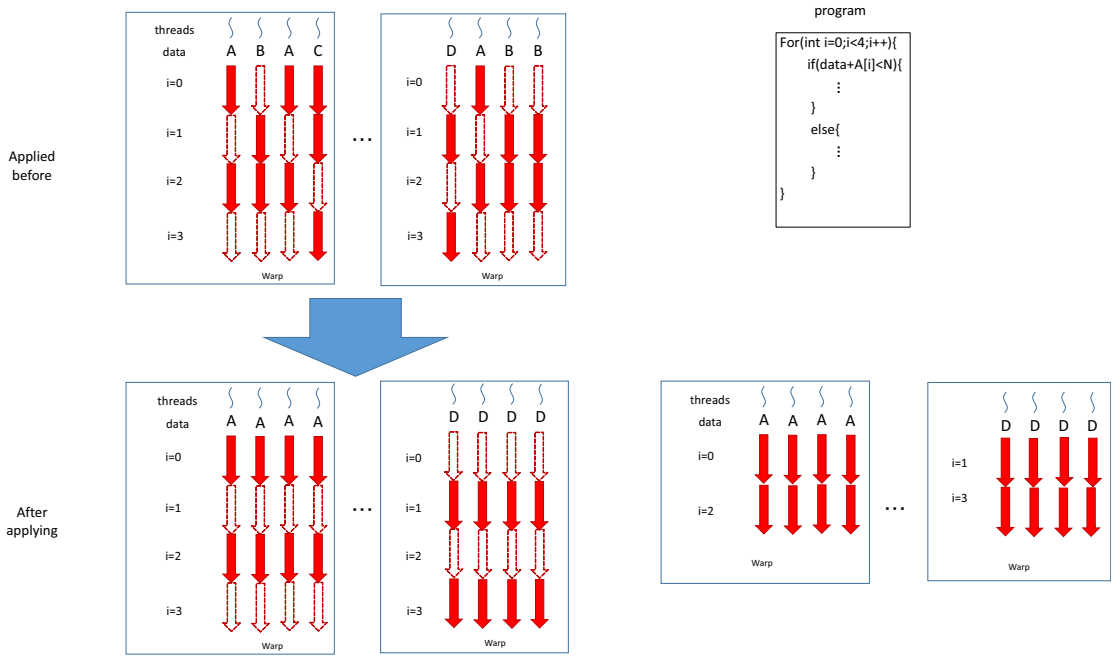

図 3.3: スケジューリングによる分岐ダイバージェンスの改善

3.2　生成されるコード

今回提案する手法について説明する．今回はGPUカーネル内で分岐命令に関わるデータが各スレッドに対応付けられたデータであると仮定している．その場合各スレッドに対応付けられたデータを用いて事前に実行パスを調べることが可能である．実行パスの違いから各スレッドをグルーピングすることで同一な実行パスを持つスレッドをまとめる．グルーピングしたスレッドを同一ワープのスレッドに割り当てることでワープ内の分岐ダイバージェンスを抑え，実行効率を改善させる．

今回の提案手法における手順は以下の3つとなっている．

手順1: 対応付けられたデータを用いて事前に各スレッドにおけるGPUカーネル内での実行パスを調べる

手順2: 手順1で調べた実行パスを元に各スレッドをグルーピングする

手順3: グルーピングしたスレッドをワープに割り当てる

実行パスの調査は元のGPUカーネルを変更することで可能である．GPUカーネルの分岐に関わる部分以外の処理をすべて取り除き，分岐命令内も分岐結果に関わる命令以外は取り除く．そして分岐の結果やルー

プの結果によって変動する変数である expid を作成する．expid は実行パスによって重複せずそれぞれ別の値をとるように GPU カーネルを書き換える．今回は if-else 文にネストされたループや二重ループなどの分岐パターンが多く expid による実行パスの分類が困難な場合は考慮してない．実行後各スレッドにおける expid を調べることでこれから実行する GPU カーネルの実行パスによってスレッドを分類することが可能となる．

スレッドのグルーピングを行うコードでは先程調べた expid をもとにスレッドのグルーピングを行う．expid をキーとしてソートすることで実行パスによるグルーピングは完了するが，手順 3 と同時に行うために本手法ではスレッド ID ソートテーブルというデータアクセステーブルを作成する．

3.3 スレッド ID ソートテーブル

スレッド ID ソートテーブルはスケジューリング後に各スレッドが対応付けられたデータにアクセスするためのデータアクセステーブルである．図 3.4, 3.5 ではスレッド ID ソートテーブルの作成過程を示している．図では 99 スレッドに対して実行パスを調べ，各スレッドに expid が 1 から 4 で割り振られている．はじめに expid と Table index を結びつける．こ

れは後にこの配列をソートした後でも元々の index にアクセス可能にしておくためである．図 3.4 の上の図が expid と Table index を結びつけた後の配列の状態となっている．次に expid をキーとしてこの配列をソートする．expid をキーに配列をソートすることで実行パスによるグルーピングを行う．図 3.4 の下の配列はソート後の配列の状態を示している．その後この配列から expid を取り除くことで Data Index と Table Index からなる配列を作成する．この配列がスレッド ID ソートテーブルとなる．

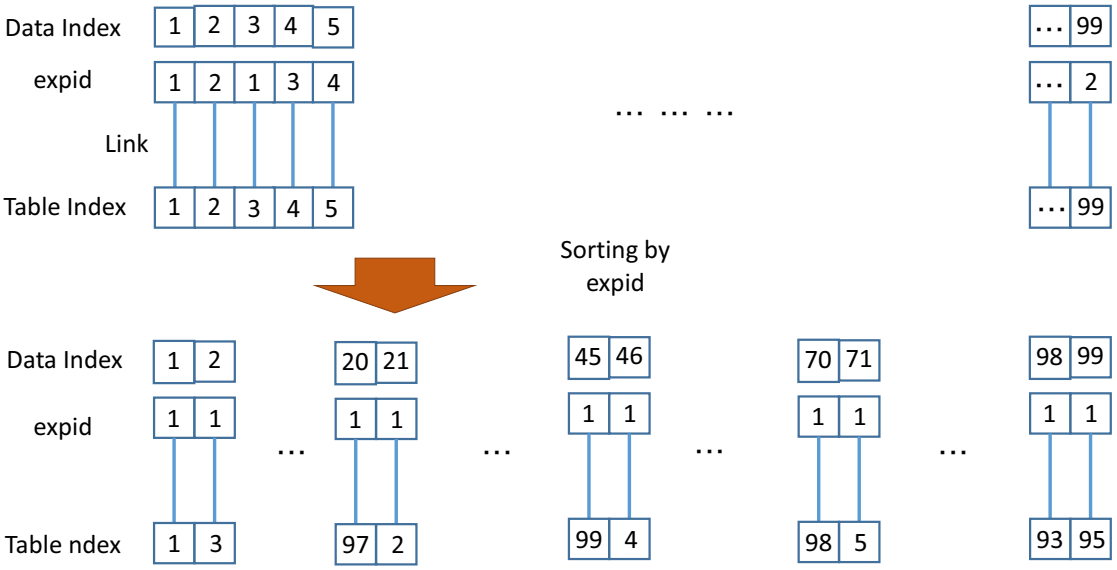

図 3.4: スレッド ID ソートテーブルの作成 1

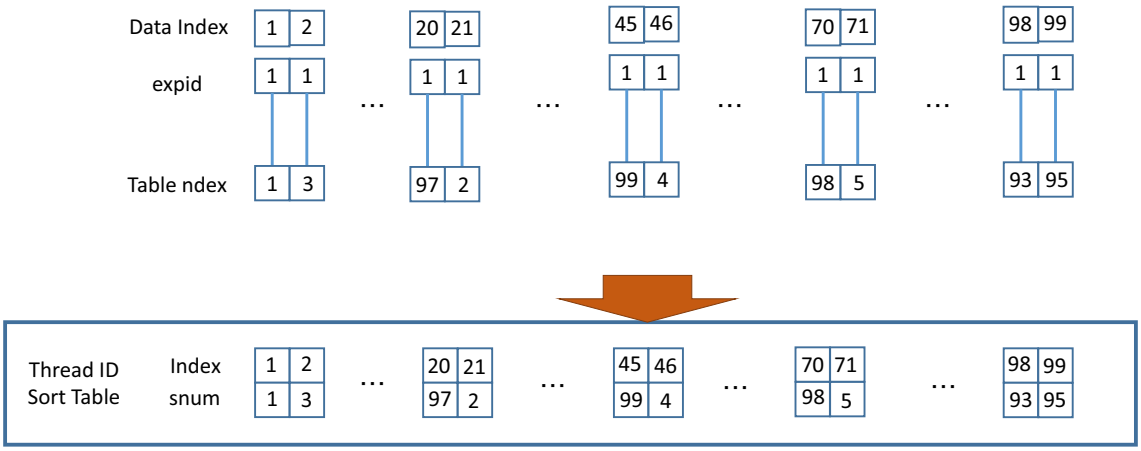

図 3.5: スレッド ID ソートテーブルの作成 2

スレッド ID ソートテーブルは GPU カーネル内で各スレッドがデータにアクセスする際に利用される．各スレッドはデータにアクセスする際，基本的にスレッドの index を配列要素としてアクセスする．この時にスレッド ID ソートテーブルを間に挟み，関節参照する．スレッド ID ソートテーブルを関節参照することで各スレッドは手順 2 でグルーピングされた後のデータにアクセスする．結果的に各スレッドの対応付けられたデータをスケジューリング後のデータに変更することで分岐ダイバージェンスの影響を抑える．

3.4 メモリ配置上の問題点と改善案

スレッド ID ソートテーブルの実装によって分岐ダイバージェンスの問題は解決するが，ワープ内スレッドの担当データは元々メモリ上でも連続配置になっていることが多く，ソートしたことによって不連続なアクセスとなりアクセス効率が低下してしまう問題が浮上する．これを防ぐために，スレッドのアクセス部分を変更させるのではなく入力データに対してソートを行う方法が考えられる．しかし入力データのソートはコストが大きいため不連続配置による性能低下とのトレードオフとなる．CUDA ではメモリアクセス部分のオーバーヘッドによる実行効率の低下が発生

しやすいため，特に注意しなければならない問題である．

4　評価

4.1　性能評価

提案手法の評価を行った．

提案手法の評価をとる際に使用した環境を以下に表で示す．

表 4.1: 評価環境

	環境
GPU	Geforce GTX 980
Memory	4GB
CPU	Xeon E5-1620 3.6GHz
Memory	16GB

はじめにループ回数がデータの長さに依存するプログラムを作成した．データは GPU カーネル実行前に事前に作成され，一様分布の乱数で定められている．データ長の測定部分はプログラムに事前に実装し，実際の計算 GPU カーネル関数とは別に GPU カーネルを作成しデータ長を測定している．各スレッドは与えられたデータの長さだけ GPU カーネル内でループし，1 回のループあたりの計算量は 2^{13} となっている．

何も手を加えず，分岐ダイバージェンスが発生するプログラムとデータ長の測定からスレッド ID ソートテーブルの作成部分を実装した提案手法プログラムの 2 つの実行時間を比較した．スレッド ID ソートテーブル

の作成時のソート部分はCUDAライブラリのThrust[7]を使用している．データ長の最大値は2^{13}で固定となっているが，データ長の最低値を変動させることで分岐ダイバージェンスの影響規模を変動させ，両プログラムの実行時間の変動を測定した．

実行時間の表を以下に示す．

表 4.2: 実験1評価結果-1

手法適用 / 問題サイズ	1	512	1024	1536	2048
手法適用なし (s)	17.92	17.29	17.20	17.94	17.26
手法適用あり (s)	8.81	9.32	9.47	10.05	10.53
短縮割合 (%)	50.83	46.10	44.97	43.96	38.97

表 4.3: 実験1評価結果-2

手法適用 / 問題サイズ	2560	3072	3584	4096
手法適用なし (s)	18.16	17.41	18.07	17.40
手法適用あり (s)	10.88	11.35	11.77	12.01
短縮割合 (%)	40.12	34.84	34.85	30.94

また，スレッド ID ソートテーブルの作成時にデータ長によってソートを行うが，ソートの昇順，降順による実行時間の差異を測定した．

表 4.4: 昇順降順比較-1

ソート順 / 問題サイズ	1	512	1024	1536	2048
昇順 (s)	8.81	9.32	9.47	10.05	10.53
降順 (s)	8.63	8.99	9.28	9.56	9.94
全体短縮割合 (%)	51.84	47.98	46.02	46.69	42.37

表 4.5: 昇順降順比較-2

手法適用 / 問題サイズ	2560	3072	3584	4096
昇順 (s)	10.88	11.35	11.77	12.01
降順 (s)	10.23	10.60	10.88	11.19
全体短縮割合 (%)	43.66	39.16	39.78	35.67

最後にこの手法のコストとなるデータ長測定とスレッド ID ソートテーブルの作成時間，実計算時間の比率を以下に示す

表 4.6: 実験 1 実行時間割合

工程	データ長測定	ID ソートテーブル作成	実計算
実行時間 (s)	0.16	0.42	8.05
実行時間割合 (%)	1.83	4.88	93.29

提案手法の有無による実行時間の表より提案手法による効果は計算量の最大，最小が離れているほど効果を発揮することがわかる．今回のプログラムでは最大で約 50%の実行時間短縮となった．データ長の昇順降

順による実行時間の違いはデータ長を降順に，つまり計算量の小さいスレッドを後に処理することによってワープごとの計算量のバラつきが昇順の時より抑えられたものだと考えられる．工程ごとの実行時間の割合はデータ長の長さによって変動する部分は最後の実計算部分のみであり，今回のプログラムでは提案手法部分の処理時間は変動しない．今回のプログラムに関しては提案手法の実行時間の割合は 10%未満となり，提案手法によるスケジューリング最適化の効果が勝った形となった．

次に if-else 文のような分岐ダイバージェンスの規模が大きくない問題においてどの程度の処理量があると手法が有効なのか調査を行った．

if 文の条件分岐が各スレッドのデータ内容に依存するプログラムを作成した．スレッド数は 2^{26} となっており，各スレッドは与えられたデータによって if-else 文のどちらかに分岐をする．分岐の確率はそれぞれ 1/2 となっている．分岐の中ではそれぞれ別の処理が行われているが，計算量は同一になるようにプログラムされている．

こちらの評価も同様に提案手法の適用の有無による実行時間の差異を調査した．今回の評価では分岐内での計算量を変動させ，実行時間がどのように推移していくかについても調査を行った．評価環境と評価結果を示した表を以下に示す．

表 4.7: 実験 2 評価結果-1

手法適用 / 計算量	1	1024	2048	3072	4096
手法適用なし (s)	0.13	0.89	1.24	1.81	2.25
手法適用あり (s)	0.72	1.09	1.27	1.57	1.80
短縮割合 (%)	—	—	—	13.57	20.21

表 4.8: 実験 2 実行時間-2

手法適用 / 問題サイズ	5120	6144	7168	8192
手法適用なし (s)	18.16	17.41	18.07	17.40
手法適用あり (s)	10.88	11.35	11.77	12.01
短縮割合 (%)	25.65	28.30	31.09	32.91

最後に最大の計算量となる 8192 の場合における各工程ごとの処理時間と処理時間割合を示す．

表 4.9: 実験 2 実行時間割合

工程	条件分岐解析	スレッド ID ソートテーブル作成	実計算
実行時間 (s)	0.13	0.43	2.25
実行時間割合 (%)	4.67	15.30	80.03

計算量が最小の 1 から 2048 の間において提案手法を適用したプログラムのほうが実行時間が長いという結果になった．これはデータ長測定と ID ソートテーブルのコストが提案手法適用による実行効率改善の効果を上回ったためだと考えられる．計算量が 3072 以上の場合は提案手法適用の実行効率改善の効果が上回り，実行時間の短縮が見られる．最大で約 32%の実行時間の短縮となった．この評価プログラムにおける提案手法の処理工程は合計で 0.56 秒ほどである．提案手法の処理工程の処理時間は GPU カーネルの分岐の複雑さとスレッド数にある程度依存しているものと考えている．条件分岐の解析は GPU カーネルの複雑さに，スレッド ID ソートテーブル作成はスレッド数と GPU カーネルの複雑さの両方が関与している．そのため，問題サイズによっては処理時間が大きくなることは想定できる．手法による効果は GPU カーネル内の分岐内の計算量

と分岐ダイバージェンスの影響規模への依存度が大きいため，自動化する際に適用するべきかどうか，の判定にはこのパラメータを使用するものと考えられる．

5　関連研究

Han Tianyi David ら [10],[11] はループにネストされた if-else 文やループに対する分岐ダイバージェンスの影響減少の手法について研究を行なっていた．if-else 文に対しては同時実行されるスレッド間の if-else の分岐方向に対して多数決をとる．多数派である分岐方向のみを実行し，分岐を実行したスレッドのみループを 1 つ進める．少数派である分岐方向のスレッドを実行せず，多数派になるまで待つことでアイドルとなるスレッドの割合を減少させている．ループにネストされたループに対しては Loop Merging という手法を提案した．各スレッドの各ループの終了時に他のスレッドのループの終了を待たずにループの終了処理から次のループの開始処理を行うことでスレッド全体のループ終了待ちによるアイドル時間の減少させた．W. Fung ら [13] は GPU カーネル内の命令に対して実行されるスレッドのみを集めてグルーピングを行い，動的なワープ生成を行なっている．しかしこの手法ではレジスタアクセスの際に競合が発生して実行不可になる可能性がある．Vaidya, Aniruddha S ら [12] は Intel の組み込み GPU による並列実行時に発生する分岐ダイバージェンスの影響軽減のために，マイクロアーキテクチャの最適化の研究を行なっていた．同時に実行されるスレッドの分岐方向によってスレッドをグルーピング

し，アーキテクチャ内で同時実行するスレッドの並び替えを行なっている．JABLIN, James A ら [14] はトレーススケジューリングを GPGPU プログラムに適用する上で弊害となる問題を解決し，GPGPU プログラムの最適化を行った．この手法は分岐に偏りのあるプログラムの最適化には向いているが，偏りのないプログラムに対してはあまり効果が見られない問題がある．

6　おわりに

本研究では分岐ダイバージェンスが発生するプログラムを最適化するスケジューリング手法を提案した．評価の結果，ランダムなデータ長に計算量が依存するプログラムに対し，実行パスや計算量によってスレッドをグルーピングすることでスレッドのアイドル時間を削減できた．今後の課題として，今回提案した手法の自動化と自動化にあたり，手法適用のためのしきい値の測定などがあげられる．

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] 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.

[2] 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.

[3] RYOO, Shane, et al. *Optimization principles and application performance evaluation of a multithreaded GPU using CUDA.* In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. ACM, 2008. p. 73-82.

[4] LEE, Daren, et al. *CUDA optimization strategies for compute-and memory-bound neuroimaging algorithms.* In: Computer methods and programs in biomedicine, 2012, 106.3: 175-187.

[5] NVIDIA Corporation, `http://www.nvidia.co.jp`(参照 2018-02-05).

[6] NVIDIA Corporation: CUDA Zone, https://developer.nvidia.com/cuda-zone(参照 2018-02-02).

[7] NVIDIA Corporation: DEVELOPER ZONE, http://docs.nvidia.com/cuda/thrust/index.html(参照 2018-02-05).

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

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

[10] HAN, Tianyi David; ABDELRAHMAN, Tarek S. *Reducing branch divergence in GPU programs.* In: Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units. ACM, 2011. p. 3.

[11] HAN, Tianyi David; ABDELRAHMAN, Tarek S. *Reducing divergence in gpgpu programs with loop merging.* In: Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units. ACM, 2013. p. 12-23.

[12] VAIDYA, Aniruddha S., et al. *SIMD divergence optimization through intra-warp compaction.* In: ACM SIGARCH Computer Architecture News. ACM, 2013. p. 368-379.

[13] W. Fung, I. Sham, G. Yuan, and T. Aamodt, *Dynamic warp formation and scheduling for efficient GPU control flow.* In: Proceedings of International Symposium on Microarchitecture, 2007, pp. 407420.

[14] JABLIN, James A., et al. Warp-aware trace scheduling for GPUs. In: Proceedings of the 23rd international conference on Parallel architectures and compilation. ACM, 2014. p. 163-174.