目 次

図 目 次

表 目 次

1 はじめに

1.1 研究背景

GPU コンピューティングのための開発環境である CUDA[1] が 2006 年に NVIDIA 社から発表されて以降、GPU は従来の 3D グラフィックス処理に限らず、科学技術計算をも、安価かつ低いプログラミングコストで行うことが可能になった。さらに近年では、対応する GPU の種類の増加やプログラムの開発環境の発展、GPU の省電力化などによって、銀行業界といった科学産業以外の業界の会社も含めて 500 社以上の会社のクラスターシステムに CUDA 環境がインストールされており [1]、また、一般消費者向けのソフトウェア開発会社の開発する製品 [2][3] でも GPU コンピューティングが利用することが可能となった。そのため今後も、幅広い分野で GPU コンピューティングが普及していくと考えられる。

しかし、CPU と GPU の両方を利用するシステムにおいて高い計算性能を得るためには、GPU に内蔵されている数百から数千の演算コアやメモリなどのハードウェアアーキテクチャの特性、および実行させるプログラムの特性を把握した上で、開発を行う必要がある。そのためには、GPU プログラムのプログラミングの際に、GPU に割り当てる実行スレッド数や、GPU に内蔵されているマルチプロセッサあたりに割り当てる実行スレッド数といった様々なパラメータの値に対してチューニングを行わなければならない。このチューニングはプログラムの実行時間に大きく影響するため、チューニング次第では CPU のみを利用する場合よりも、実行させるプログラムの実行時間が増大してしまうこともありえる。最適なチューニングの内容は GPU のハーウェアスペック、および作成するプログラムによって異なり、一貫する指針は存在しない。そのため、プログラマはパラメータの値の調整を繰り返しながら適切な値を見つけることになり、開発コストが増大してしまう。

利用するシステムのもつ GPU の数や種類が増加した際に、それらを並列に使用するには各 GPU の演算コア数やその性能、メモリ性能の差異などにも着目してパラメータの値を決定しなければならない。さらに、GPU が搭載されている複数のコンピュータノードがネットワークなどを介して接続されたマルチノードなヘテロジニアス構成のシステムの場合には、それぞれの GPU に割り当てる処理量の分割方

法やノード間をつないでいるネットワークの速度といった要素が新たに増えるため、チューニングコストが上昇してしまうと予想される。

1.2 研究目的

上述の背景のもと、本研究では、高速な GPU プログラムの開発コスト低減を目的とし、GPU に割り当てるスレッド数などのワークサイズを自動的に決定し、この値を未チューニングの GPU プログラムのソースコードに対して反映させるためのソースコード自動変換機構の作成を行う。

特に、GPU の種類や、GPU が搭載されているノードごとのハードウェア構成やシステム構成が異なるようなヘテロジニアスなシステムにおいて、使用する GPU の選択手法や適切なワークサイズへのチューニング方針に着目することで、そのようなシステムを想定したプログラムの開発コストの低減や、単一 GPU のみを使用する設計の GPU プログラムなどの資産の活用を図る。

1.3 本論文の構成

本論文は 6 つの章で構成される。第 1 章は、本章である。第 2 章では、GPU プログラミングの方法と本研究で利用する GPU コンピューティングのためのフレームワークである OpenCL について述べる。第 3 章では、本研究が対象とするパラメータの種類と OpenCL における性能チューニングの一般的な課題、および関連研究について述べる。第 4 章では、本研究が対象とするパラメータのチューニング方針について述べる。第 5 章では、提案するパラメータ調整機構について述べる。第 6 章では、本論文におけるまとめ、および今後の課題を述べる。

2　GPU と OpenCL プログラミング

本章では、GPU の概要と GPU プログラミングの方法、および本研究で対象にする GPU コンピューティングのためのフレームワークである OpenCL について述べる。

2.1　GPU と GPU コンピューティング

CPU に搭載されている演算ユニット数が 1 個から数十個程度である一方で、GPU は並列処理を効率的に実行するために設計された数百から数千個のストリーミングプロセッサ（以下、SP と呼ぶ）を搭載している。SP は演算ユニットであり、複数の SP を用いて同時並列的に演算を行うことが可能である。複数の SP とそれらの制御用のユニットをまとめたストリーミングマルチプロセッサ（以下、SM と呼ぶ）内には数十から数百個の SP が搭載されていて、SM 単位でローカルアクセスすることが可能な共有メモリが接続さている。これらの一般的な CPU のコアは分岐予測や投機実行を行うための機能を備えているが、GPU にはこれらの機能は基本的に備えられていない。しかしそのため、集積するトランジスタを演算器に多く割り当てられるため、ダイ面積あたりの性能は CPU に比べて GPU の方が高い。

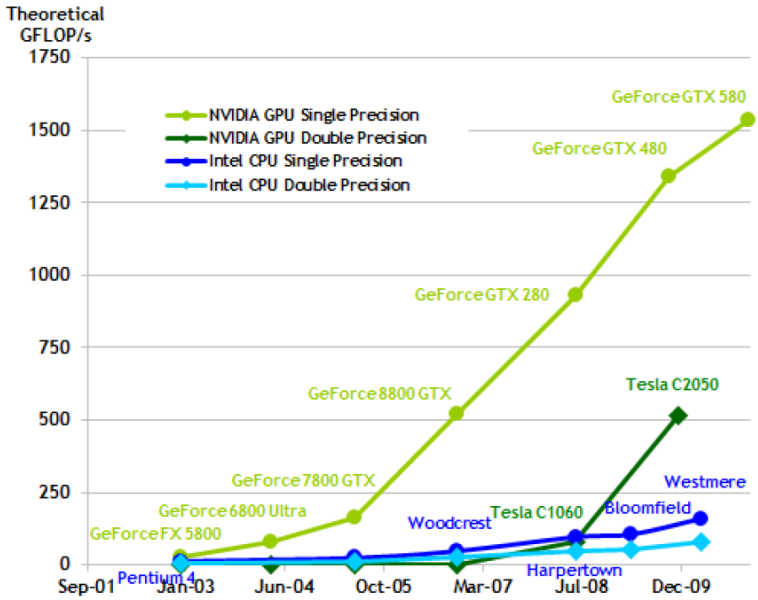

図 1: CPU と GPU の演算能力の比較 [8]

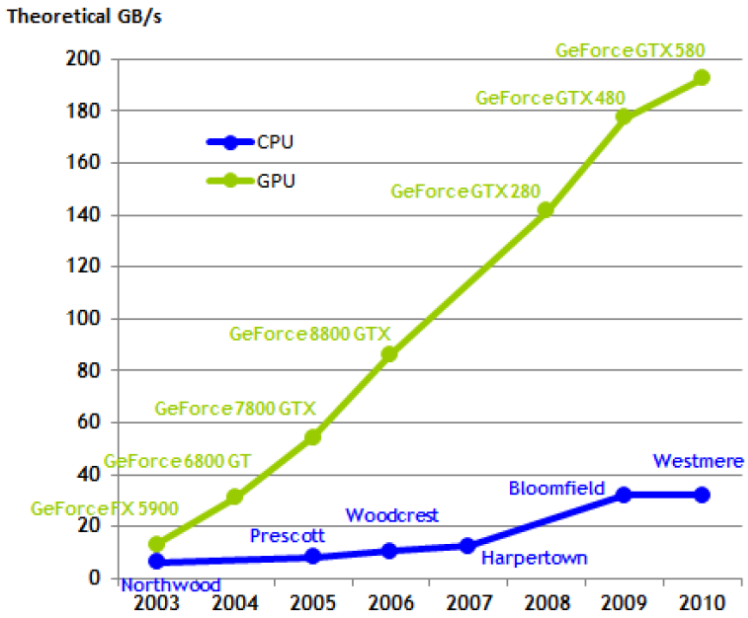

図 2: CPU と GPU のメモリアクセス速度の比較 [8]

近年では、GPU は 3 次元グラフィックスの画像処理だけに留まらず、数値計算や物理シミュレーションなどの一般的な計算にも使用され始めている。GPU は前述のような高い並列性と演算性能を持つため、異なるデータに対して同じ命令を同時に実行する SIMD 型のデータ並列処理やマルチメディア処理に対して有効的である。GPU は CPU に比べて高い演算性能と高いメモリバンド幅を持つ（図 1、図 2）。また、GPU の省電力化に関する研究や開発が進んだことによる GPU の消費電力低下や、GPU で実行するプログラムの開発環境の発展も相まって、世界中の企業や研究機関において GPU コンピューティングが注目されている。

GPU で実行するプログラムの開発環境は、プログラム可能な GPU に関する初期研究 [4] が 1980 年代後半に発表され、この頃から注目が集まり始めた。当初は、アセンブリ言語で記述していたが、その後、GPU に搭載されたプログラマブルシェーダを利用する Cg や HLSL などの画像処理用言語で記述していた時期を経て、今日では GPU プログラムを通常の CPU プログラムと同じようなスタイルで記述することが可能な CUDA や Brook+[5] が利用されるようになった [6]。アセンブリ言語や画像処理用言語に比べて、CUDA や Brook+は C 言語や C++の言語拡張であるた

め習得コストが低く、プログラマにとって GPU プログラムの開発は格段に容易になった。現在のところ、CUDA は NVIDIA 社の GPU 上でのプログラミング環境として、Brook+は AMD 社の GPU 上でのプログラミング環境である ATI Stream でのプログラミング言語として利用されている。また、Microsoft 社の GPU コンピューティング技術である DirectCompute では、同社が発表した C++の拡張である C++ AMP が利用されている [7]。

2.2 OpenCL

GPU コンピューティングのための開発環境が GPU メーカーごとに異なっているため、プログラマは使用する GPU プログラミング言語や API を、GPU メーカーによって使い分けなければならず、多数の開発環境を習得しなければならなかった。そのため、プログラムの移植性が低く、作成した GPU プログラムを他のメーカーの GPU で使用できるようにするためには、そのメーカーの開発環境でプログラムを作り直さなければならないという問題があった。

この問題を解決するために GPU コンピューティングの統一規格として、ヘテロジニアスコンピューティングのための標準フレームワークである OpenCL(Open Common Language)[9] が策定された。

2.2.1 OpenCL の概要とプラットフォームモデル

OpenCL はヘテロジニアスコンピューティングに適した並列プログラミングのための標準フレームワークであり、演算デバイスで動作するプログラムを記述できる OpenCL C 言語、および C 言語や C++から呼び出すことが可能な OpenCL ランタイム API などが含まれている。これらの仕様は業界標準化団体であるクロノスグループによって策定されている。クロノスグループには NVIDIA 社や AMD 社を含む主要なプロセッサベンダらが加盟している。

OpenCL では、次の 3 つの特徴が挙げられる。

- GPU やマルチコア CPU、DSP、Cell/B.E. など、メーカーやチップの種類などを問わず、統一的な記述によって並列プログラミングが可能である。

- ハードウェアに近いレベルでAPIを共通化した仕様になっているため、OpenCLのAPIの利用がパフォーマンスの低下につながるオーバーヘッドになることが少ない。

- OpenCLランタイムAPIを利用する制御用プログラムは通常のC言語、またはC++、Java、C#などによる開発が可能であり、特別なツールは必要としない。また、演算デバイス上で実行するプログラムを記述するOpenCL C言語はプリミティブなC言語の拡張である。そのため、他のメーカー独自の開発環境に比べて習得コストが低い。

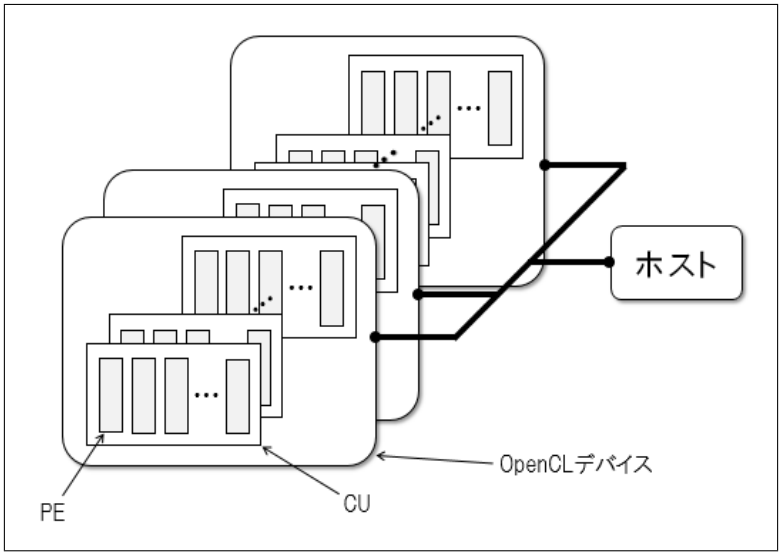

図 3: OpenCLのOpenCLのプラットフォームモデル

プラットフォームモデル

OpenCLが対象としているプラットフォームは、CPUなどの「ホスト」に対して、GPUなどの演算デバイスである「OpenCLデバイス」が1つ、あるいは複数個が接続されている形態である（図3）。OpenCLデバイスは、1つ、あるいは複数個のコンピュートユニット（以下、CUと呼ぶ）を含み、かつ、CUは1つ、あるいは複数個のプロセッシングエレメント（以下、PEと呼ぶ）を含んでいる。

例として、OpenCL デバイスを GPU とした場合、SM が CU にあたり、SP が PE にあたる。

このプラットフォームモデルにおいて、データ並列処理を行う場合には PE が最小単位となり、PE を単位として SIMD、あるいは SPMD 方式で実行されることが一般的である。

2.2.2 OpenCL のプログラミングモデル

同じカーネルプログラムを実行するデータ並列モデルと、異なるカーネルプログラムを実行するタスク並列モデル、あるいは両者の混在したプログラミングモデルで並列処理を実行することができる。ただし、本論文ではデータ並列モデルのみを扱うものとする。

ソースコード 1: カーネルプログラムのソースコード例

```
1  // kernel.cl
2  // int 型の配列の各要素に 1 を加算するカーネル関数
3  __kernel void addOne(__global int* data, int data_size)
4  {
5          int index = get_global_id(0);
6          data[index] = data[index] + 1;
7  }
```

OpenCL では、ホスト上で実行するホストプログラムと、OpenCL デバイス上で実行するカーネルプログラムのそれぞれを作成する必要がある。ホストプログラムは、OpenCL の API を用いてカーネルプログラムのセットアップや実行、演算デバイスとのデータの転送を実行する。一方、カーネルプログラムは OpenCL デバイス上で行う演算処理が中心になる。

カーネルプログラムのソースコードは、標準 ANSI C(C99) をベースとして並列処理を抽出するように拡張した OpenCL C 言語で記述する。カーネルプログラムのソースコードの例をソースコード 1 に示す。各ワークアイテムは、5 行目のように、

NDRange に対して一意に識別可能な ID を取得することで、データ並列モデルを実現することができる。

ホストプログラムのソースコードには、カーネルプログラムの実行命令の発行に加え、ホストと OpenCL 間で、計算に必要なデータを転送する処理などを、OpenCL の API を利用して記述する必要がある。単一の GPU を使用する場合の基本的な処理は、下記の流れが一般的である。

1. ホストの OpenCL 環境、および OpenCL デバイス情報の取得
2. 使用する OpenCL デバイス用のコマンドキューやバッファオブジェクトなどの必要なオブジェクトを生成
3. ホスト内のメモリから OpenCL デバイス内のメモリへの計算に必要なデータのコピー
4. カーネルプログラムの実行命令の発行
5. OpenCL デバイス内のメモリからホスト内のメモリへの計算結果のコピー
6. 2 で生成したオブジェクトのメモリ開放などの終了処理

2.2.3 実行単位の管理モデル

GPU は、高い並列性を活かすために数百から数万に及ぶスレッドを扱う。一般的な CPU ではコア数と同程度かその数倍程度のスレッドしか扱わないため、GPU は膨大な数のスレッドを扱うと言える。そのため、OpenCL では下記に挙げる管理モデルを用いて、スレッドの管理を行う。

NDRange

NDRange は、カーネルプログラムが実行される際に用意されるカーネルプログラムの実行スレッドが属する空間である。この空間は、カーネルプログラムの実行命令をホストプログラム内で呼び出す際に 1、2、3 次元の次元数が指定され、この次元数の次元空間になる。

ワークアイテム

ワークアイテムは、カーネルプログラムを実行するスレッドのことである。ワークアイテムはNDRangeにマッピングされ、個々のワークアイテムは空間内の絶対位置によって一意に識別するためのIDが割り当てられ、1つのPE上で実行される。

ワークグループ

ワークグループ、1つ以上のワークアイテムをグループ化したものである。ワークグループは1つのCU上で実行され、グループ内のワークアイテムはそのCU内のPEで実行されることになる。ワークグループはNDRangeの次元数を最大として、任意の次元でワークアイテムをマッピングすることが可能である。その際、1つのNDRange内では、すべてのワークグループは、各次元へマッピングしたワークアイテムの個数が等しくなる。

上述のOpenCLの実行モデルの概念図を図4に示す。図4では、4096個のワークアイテムをNDRangeに対して、x方向、y方向、z方向の3方向すべてに16個ずつマッピングしている様子を表している。また、図4でのワークグループは、x方向、y方向、z方向の3方向すべてに4個ずつのワークアイテムをマッピングしたグループ構成を表していて、合計で64個がつくられている。

また、HPC分野で頻繁に利用されているGPUコンピューティングの実行環境であるCUDAとOpenCLの用語の対応表を表1に示す。

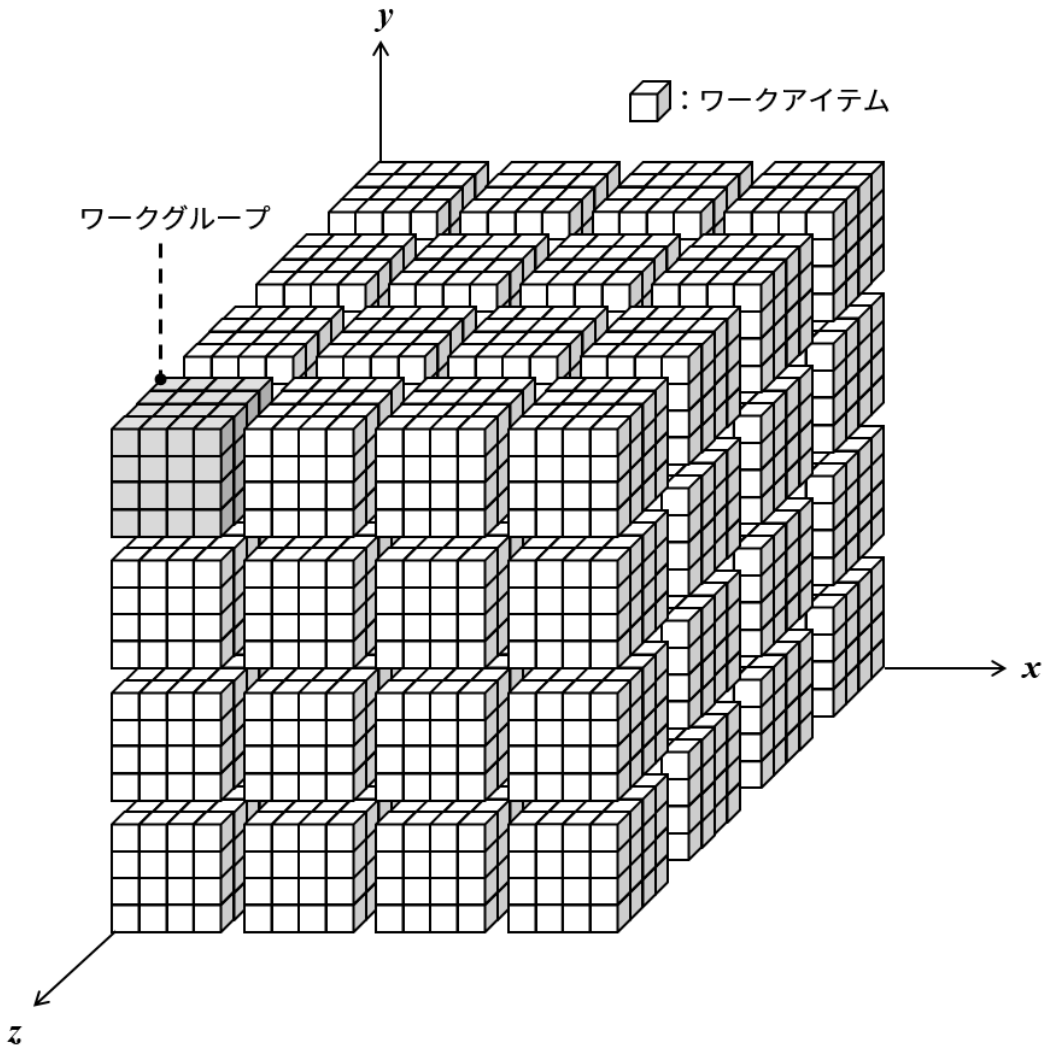

図 4: OpenCL の実行モデルの概念図

表 1: CUDA と OpenCL の用語対応表

CUDA	OpenCL
grid	NDRange
block	ワークグループ
thread	ワークアイテム
warp	該当する概念なし

3 OpenCLプログラミングにおける性能チューニングの一般的な課題

本章では、OpenCL プログラミングにおけるパラメータとパラメータチューニングの一般的な課題について述べる。次に、パラメータチューニングについて本研究での着眼点を述べ、その効果を実機を用いて実験することで検証する。また、OpenCL プログラミングについての関連研究を挙げる。

3.1 本研究で対象とするパラメータ

本研究では、OpenCL プログラミングにおける性能パラメータの中から、チューニング対象を次の 3 つのパラメータとする。その理由は、OpenCL プログラミングにおいてこれらのパラメータチューニングが、プログラムの実行時間に大きな影響を与えるためである。

- グローバル・ワークサイズ
- ローカル・ワークサイズ
- ワークグループ内でのワークアイテムのグループ構成方法

グローバル･ワークサイズは、カーネルプログラムを実行する際に、1 台の OpenCL デバイスで実行されるワークアイテムの総数で、カーネルプログラムの設計内容と使用する OpenCL デバイス数、各 OpenCL デバイスへの処理量の分配比によって決定される。

例えば、配列の 1 要素を 1 ワークアイテムが演算によって求めるカーネルプログラムでかつ、使用する OpenCL デバイスが 1 台の場合には、グローバル・ワークサイズは配列要素数と等しくなる。同様の条件で N 台の OpenCL デバイスで処理を均等に分配する場合には、すべての OpenCL デバイスのグローバル・ワークサイズは、

$$\text{配列要素数}/N \tag{1}$$

になる。

ローカル・ワークサイズは、1 つのワークグループ内のワークアイテムの総数、すなわち、1 つの CU 上で実行されるワークアイテム数である。したがって、グローバル・ワークサイズは式 (2) となる。

$$\text{グローバル・ワークサイズ} = \text{ローカル・ワークサイズ} \times \begin{array}{r} OpenCL \text{ デバイス} \\ \text{あたりのワークグループ数} \end{array} \quad (2)$$

式 (2) より、グローバル・ワークサイズが一定の場合には、ローカル・ワークサイズを決定すると OpenCL デバイスあたりのワークグループ数が求まることになり、逆に、OpenCL デバイスあたりのワークグループ数を決定するとローカル・ワークサイズが求まることになる。

ローカル・ワークサイズはグローバル・ワークサイズの約数としなければならない。厳密には、NDRange の各次元方向にマッピングしたワークアイテムの個数が、ワークグループの対応する各次元方向にマッピングしたワークアイテムの個数がすべて約数としなければならない。

さらに、ワークグループ内でのワークアイテムのマッピング方法によって、複数の種類のグループ構成方法が考えられる。1 つのローカル・ワークサイズに対して、そのワークグループ内のワークアイテムのグループ構成方法には複数の種類が候補として挙げられる。

例えば、ローカル・ワークサイズ=8 の場合を考えると、次の 4 種類のグループ構成が存在する。

(x 方向へのワークアイテムのマッピング個数, y 方向へのワークアイテムのマッピング個数) = (1, 8), (2, 4), (4, 2), (8, 1)

3.2 パラメータチューニングの課題

本研究で対象とする 3 つのパラメータの適切な値は、演算に使用する OpenCL デバイスなどのハードウェア特性、および実行するプログラムの特性によって決定さ

れ、そのためには、プログラマはそれぞれのパラメータに対して繰り返しチューニングを行い、プログラムの実行時間の測定を行うことで適切な値を調べていく必要がある。さらに、使用可能な OpenCL デバイスの種類が多い環境ほど、各 OpenCL デバイスのハードウェアスペックやアーキテクチャによってパラメータの選択肢が増えるため、パラメータの適切な値の発見は難しくなる。

3.2.1 単一ノード上での課題

単一ノード上で、1 つのホストに対して GPU などの OpenCL デバイスが 1 台以上接続されている環境 (図 3) での課題を述べる。

グローバル・ワークサイズ

OpenCL プログラムを設計する際に、通常は設計者によってカーネルプログラムの並列数が決められている。

そのため、使用可能な OpenCL デバイスが 1 台のみの場合は、設計者が意図した並列数をグローバル・ワークサイズとして設定する必要がある。しかし、使用可能な OpenCL デバイスが複数台存在し、それらに並列処理部を分割して実行させるためには、各 OpenCL デバイスに対してのワークアイテムの割り当て量を調整することになる。その場合には、各 OpenCL デバイスに搭載されている CU 数や PE 数、CPU-GPU 間のデータ転送速度などのハードウェア情報、および実行するカーネルプログラムに転送する演算に必要なデータサイズの情報によって、適切な OpenCL デバイスに対して、適切なグローバル・ワークサイズでワークアイテムを割り当てなければならない。

ローカル・ワークサイズ

一般的に演算を行う OpenCL デバイスに搭載されている CU 数、および CU あたりの PE 数をもとに決定される。CU あたりの PE 数は、1 つの CU 上で同時実行可能なワークアイテムの最大数を表している。つまり、CU あたりの PE 数をローカル・ワークサイズが上回った場合、実行されていないワークアイテ

ムは待機状態になり、PE が空くのを待機する時間と、先に処理されたワークアイテムとの切り替え時にレイテンシが発生する。一方、CU あたりの PE 数をローカル・ワークサイズが下回った場合、待機するワークアイテムは発生しないが、ワークアイテムを実行していないアイドル状態の PE が発生してしまい、GPU の並列性を活かせない。

さらに式 (2) より、グローバル・ワークサイズが一定の場合に、ローカル・ワークサイズを小さく設定すると、ワークグループ数が増加し、CU に割り当てられたワークアイテムが終了するのを待機する時間と、ワークグループの切り替え時にレイテンシが発生する。一方、ローカル・ワークサイズを大きく設定した場合、ワークグループ数が減少し、ワークグループ数が CU 数よりも下回った場合にはアイドル状態の CU が発生してしまう。

すなわち、演算を行う各 OpenCL デバイスに搭載されている CU 数、および CU あたりの PE 数をもとに、適切なローカル・ワークサイズは決定される。

ワークグループ内でのワークアイテムのグループ構成方法

ワークグループ内での各次元方向へのワークアイテムのマッピングによるワークアイテムのグループ構成は、カーネルプログラムのメモリアクセスパターン、およびワークグループ内のワークアイテムの参照するデータの重複度によって最適な構成が異なる。

NVIDIA 社の Fermi アーキテクチャや Kepler アーキテクチャを採用している GPU では、各 SM が L1 キャッシュを搭載しているため、SM 内のすべての SP が共有キャッシュを使用することが可能である。そのため、ワークアイテムが参照するデータの重複度が高くなるようにワークグループを構成した場合には、キャッシュヒット率が上がるため、メモリアクセスレイテンシが削減される [10]。

さらに、メモリアクセスパターンが、メモリの連続したアドレスの方向である行方向にアクセスをするパターンの場合には、メモリから行方向に連続したデータでバーストが行われるとすると、ワークアイテムを x 方向に多くマッピングした方がメモリアクセスレイテンシの削減になる。

したがって、GPU に搭載されているハードウェアの空間的局所性を活用できるように、ワークグループ内での各次元方向へのワークアイテムのマッピングを行わなければならない。

3.2.2 複数ノード上での課題

複数ノード上、つまり、ネットワークなどを介して複数のノード上の OpenCL デバイスが使用できる環境の場合には、3.2.1 節で述べた課題に加えて、次のような課題が存在する。

第一に、グローバル・ワークサイズを調整する際に、リモートノード上の OpenCL デバイスにはノード間通信にかかる時間的オーバーヘッドが発生することが課題になる。その場合には、ノード間のネットワークの通信速度と、ノード間で転送しなければならないデータのサイズによっても、適切なグローバル・ワークサイズは決定されることになる。

第二に、OpenCL にはノード間通信をサポートする機能が存在しないため、ノード間通信には MPI などの別の API を使用しなければならないため、習得コストが高くなってしまうことが課題になる。また、その場合には MPI を使用することによる時間的オーバーヘッドが発生するため、MPI ルーチンの利用や通信にかかる時間が、カーネルプログラムの実行時間を上回ってしまう場合も考えられる。

3.3 関連研究

GPU コンピューティングに関する研究では、一般的な GPU プログラムの実行時間短縮に対する研究が多数行われている。GPU コンピューティングではハードウェアの性質上、次の 3 つの動作に時間がかかってしまうことが知られている。実行時間の短縮に関する研究では、これらの動作による時間的オーバーヘッドに着目している研究が多い。

- GPU 内のメモリと CPU 側のメモリとの間でのデータコピー
- SP による GPU のグローバルメモリへのアクセス

- カーネルプログラム内の条件分岐

本研究では、パラメータチューニングのみによってプログラムの実行時間の削減を図っているが、本間らの研究 [13] や Hyeran Jeon らの研究 [14] では、GPU 内のメモリと CPU 側のメモリとの間でのデータコピーとカーネルプログラムの実行をオーバーラップさせるためのスケジューリング手法を提案し、CPU-GPU 間のデータ転送にかかるレイテンシの削減を達成していた。また、Snaider Carrillo らの研究 [15] では、条件分岐部の分割手法の提案によって、GPU プログラムの実行時間短縮を達成していた。

逆にこれらの研究では、SM あたりのスレッド数は単純な値に固定している場合が多く、ワークサイズのパラメータチューニングを適用した際に得られる効果については述べられていない。また、複数のノード上に搭載されている異なる種類の GPU を対象にしている研究も少ない。

GPU プログラムのチューニングのために、GPU のハードウェア特性や実行するプログラムの特性に着目した研究も多く発表されている。伊藤らの研究 [16] では、主記憶や GPU メモリ、GPU の演算器間の転送バンド幅と転送遅延の組みをもとに、実行時間を予測する性能モデルを提案している。島田らの研究 [17] では、プログラムの GPU メモリへのアクセス回数や命令数などのパフォーマンスカウンタを用いて実行時間を予測するモデルを作成している。本研究ではパラメータチューニングを自動で行うことで GPU プログラムの開発コストの削減を狙っているが、これらの研究では、作成したモデルをもとに、GPU プログラムの開発者がプログラムを手動でパラメータなどのチューニングを行うことを目的とされている。

本研究でも行っている GPU プログラムのパラメータチューニングに関しても、特定の数値計算プログラムに対しての最適パラメータの決定に関する研究 [18] や、一般的な GPU プログラムを対象とする最適パラメータの決定に関する研究 [19][20] が行われている。研究 [19] や研究 [20] では、CUDA プログラミングにおいて、ブロック数やブロックあたりのスレッド数に加えて、スレッドあたりの使用レジスタ数をもとにこれらのパラメータチューニングを行うことで、GPU プログラムの実行時間の短縮を達成している。

4　パラメータチューニング方針

本章では、3.1 節で述べた 3 つのパラメータの値をどのように決定すればよいのかを、各パラメータとプログラムの実行時間の関係を調べることで、本研究で行うパラメータチューニングの方法を決定する。

まず、使用する各 GPU に対してカーネルプログラムの処理をどのように分割するのかを決定するために、GPU のハードウェア特性とカーネルプログラムの処理の大きさ、およびノード間のデータ転送速度がプログラムの実行時間にどのように影響するのかを調べるための実験を行うことで、適切なグローバル・ワークサイズの決定方法を決定する。

次に、演算に使用する各 GPU で、一定のグローバル・ワークサイズの場合に、ローカル・ワークサイズ、およびワークグループのワークアイテムのマッピング方法がプログラムの実行時間にどのように影響するのかを調べるための実験を行うことで、これらのパラメータの決定方法を決定する。

4.1　実験環境

各実験では、ホストプログラムのコンパイルに使用するコンパイラと、OpenCL、ノード間通信に利用した MPI の実装は、表 2 に示すものを使用する。

また、各実験は表 3 に示す GPU が別々のノードに搭載されている計算機環境で行う。

各ノード間のネットワークの通信速度の差は、平均で 5%程度であるので、大きな差はないと言える。

表 2: 実験環境

コンパイラ	GNU gcc 4.4.7
OpenCL	OpenCL 1.1
MPI	MPICH2 1.2.1

表 3: 実験に使用した GPU のスペック表

	Tesla K20	Quadro 2000D	GeForce 8800GTS
CU 数	13	4	16
1CU あたりの PE 数	192	48	8
総 PE 数	2496	192	128
最大クロック周波数 [MHz]	1625	1251	1625
設定可能な最大 ローカル・ワークサイズ	1024	1024	512

4.2 使用する GPU の選択と各 GPU に割り当てるワークアイテム数

使用可能な GPU が複数台存在する場合において、それらに対してカーネルプログラムの処理量であるグローバル・ワークサイズの分割比を変化させた場合のプログラム全体の実行時間を測定することで、本研究でのグローバル・ワークアイテムのチューニング方法を策定する。

実験方法は、Quadro 2000D が搭載された Node1 と GeForce 8800GTS が搭載された Node2 が、それぞれネットワークを介して Node0 に接続されている環境で、Node0 をホストプログラムの起点として、MPI を使用したマスタースレーブ方式で実行する。ホストプログラムの手順は、MPI を利用して Node0 で作成した計算に必要なデータを Node1 と Node2 に送信し、Node1 と Node2 はそのデータを自身に搭載されている GPU のメモリにコピーし、カーネルプログラムのサブセットを行った後、GPU から計算結果をコピーし、そのデータを Node0 に転送する。カーネルプログラムのサブセット時には、Node1 と Node2 で、問題サイズを指定した分割比で分割したグローバル・ワークサイズを設定する。また、ローカル・ワークサイズは、2 台の GPU で使用しない PE が発生しないように、50 に固定する。

この実験では、計算量が大きく異なる次の 3 つのカーネルプログラムを用いた。

- 行列積カーネル （計算量：大）
- 画像拡大・縮小のための線形補完法カーネル （計算量：中）
- 台形公式による区分求積カーネル （計算量：小）

また、CPU の違いなどのノードごとのハードウェア差異を隠蔽するために、カーネルプログラムは全体の計算処理をそれぞれ 10 回、10000 回、10000 回実行することで、計算量を増やした。

NDRabge の次元数が 1 で実行することが想定された区分求積カーネルと、次元数が 2 で実行することが想定された行列積カーネルと線形補完法カーネルの両次元数の場合とも、処理量の分割はブロック分割で行った（図 5）。

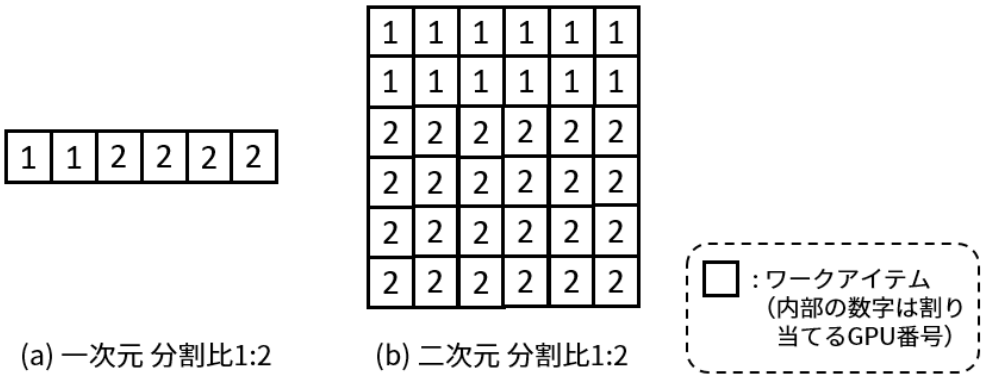

図 5: NDRange にマッピングされたワークアイテムのブロック分割例

この実験によって得られたグローバル・ワークサイズとプログラム全体の実行時間の関係を、図 6、図 7、図 8 に示す。

図 6、図 7、図 8 から、計算量が多いカーネルプログラムほど GPU の性能差が実験結果に顕著に現れていることが分かる。ここでは、GPU の性能を、その GPU に搭載されている総 PE 数を指標とすることにする。その理由として、ワークアイテムは PE 上で実行されるため、PE 数が多いほど高い並列性が得られるからである。

計算量が多い行列積カーネルでは、総 PE 数が多い Quadro 2000D に 9 割のワークアイテムを割り当てた場合が、すべての問題サイズで実行時間が最小となった。

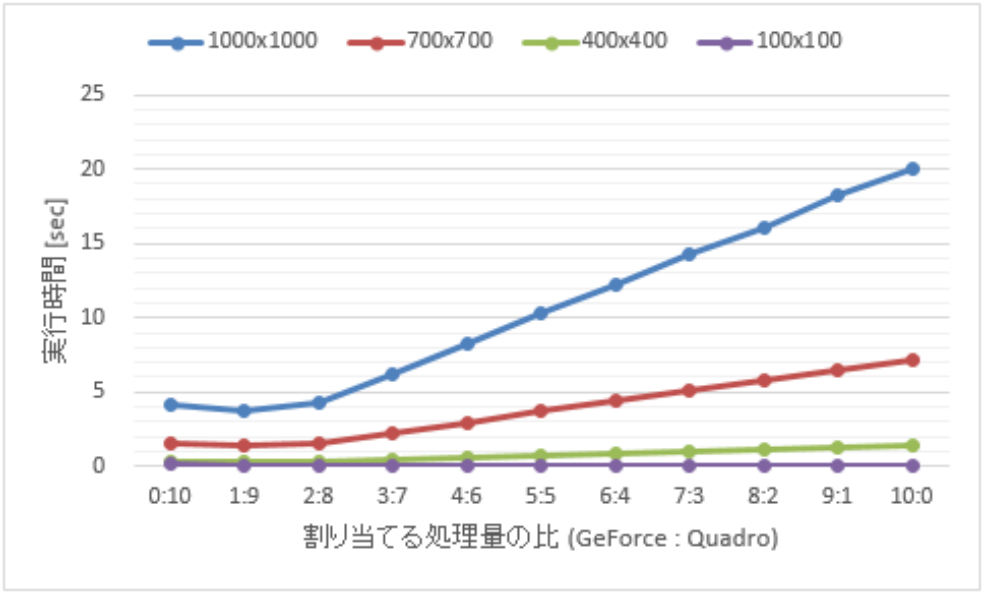

図 6: 行列積カーネル（計算量：大）の割り当てる処理量の比と実行時間の関係

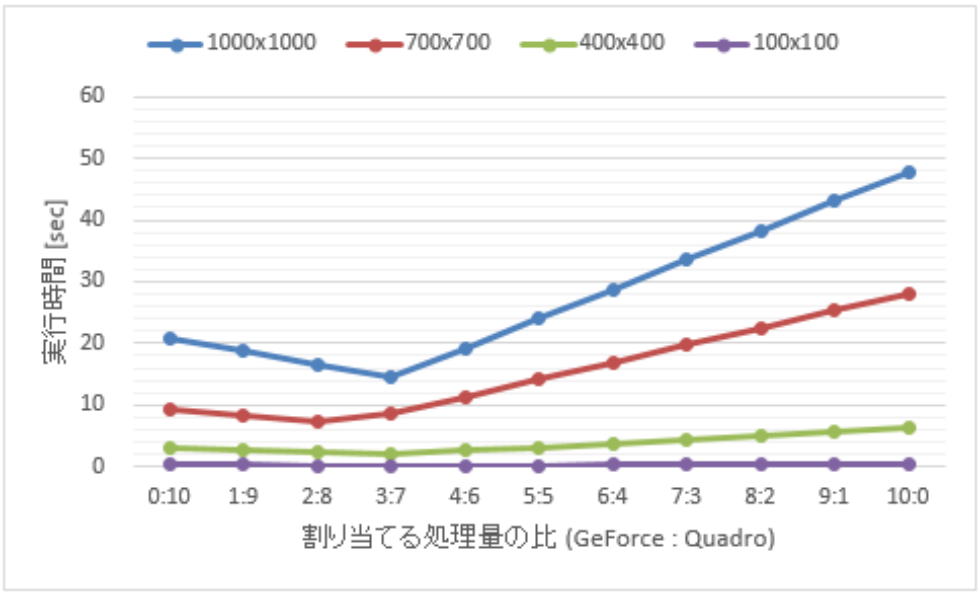

図 7: 画像拡大・縮小のための線形補完法カーネル（計算量：中）の割り当てる処理量の比と実行時間の関係

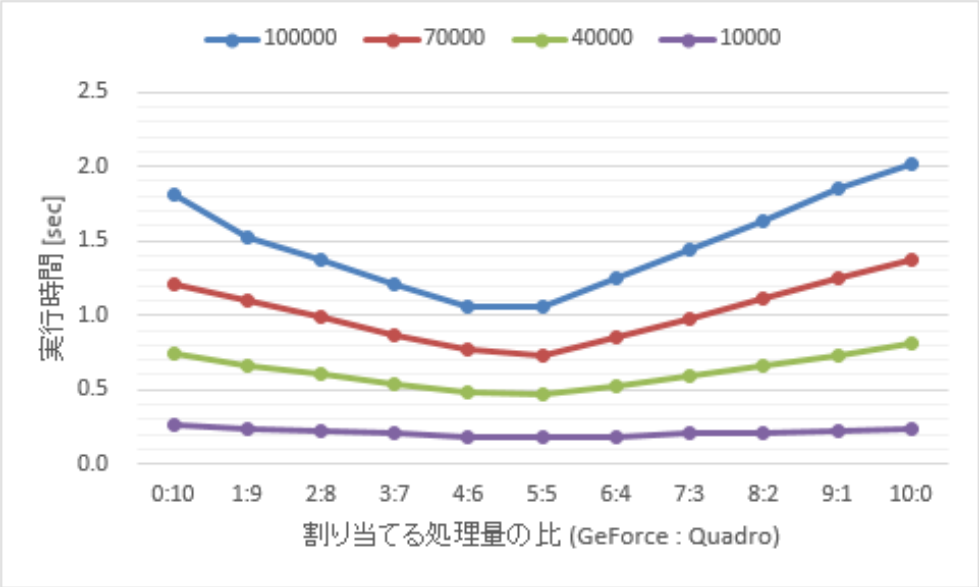

図 8: 台形公式による区分求積カーネル（計算量：小）の割り当てる処理量の比と実行時間の関係

一方で、計算量が少ない区分求積カーネルでは均等に分配した場合で、実行時間が最小となった。この理由としては、計算量が多いカーネルプログラムほど、1 ワークアイテムが PE 上で実行される時間が多くなるため、多くのワークアイテムを並列に実行できる GPU の方が、カーネルプログラムの実行時間が短くなるためであることが考えられる。

表 4: 処理量の分割比に関する実験の結果

カーネルプログラムの計算量	実行時間が最小となる処理量の比 （PE 数が少ない GPU：PE 数が多い GPU）
大	1：9
中	3：7
小	5：5

この実験によって得られたグローバル・ワークサイズの分割比とプログラム全体の実行時間の関係から、総 PE 数の比が 2 : 3 である 2 台の GPU を使用した場合、カーネルプログラムの計算量と、実行時間が最小となる処理の分割比は表 4 となることが分かったため、この結果をもとに、GPU のもつ総 PE 数に対しての割り当てるワークアイテムの割合を求めるためにモデル化を行う。

GPU のもつ総 PE 数の少ない方の GPU を GPU_{few}、GPU のもつ総 PE 数の多い方の GPU を GPU_{many} とし、GPU_{few} と GPU_{many} のもつ総 PE 数の和に対する、GPU_{few} のもつ総 PE 数の割合を PE_ratio_{few} とし、GPU_{many} のもつ総 PE 数の割合を PE_ratio_{many} とする。

2 台の総 PE 数の比が 2:3 以上の開きがある場合、GPU_{few} に割り当てるワークアイテムの割合は、カーネルプログラムの計算量が多い場合から順に、0.1、0.3、0.5 となるようにする。そこで、式 (3)、式 (4)、式 (5) を満たす 3 種類の $ThreadDetermine_{size}$ を求めると、式 (6) が得られる。

$$PE\_ratio_{few} \times\ ThreadDetermine_{big} = 0.1 \tag{3}$$

$$PE\_ratio_{few} \times\ ThreadDetermine_{middle} = 0.3 \tag{4}$$

$$PE\_ratio_{few} \times\ ThreadDetermine_{small} = 0.5 \tag{5}$$

$$\begin{aligned} ThreadDetermine_{big} = 0.25, \quad ThreadDetermine_{middle} = 0.75, \\ ThreadDetermine_{small} = 1.25 \quad (0 < PE\_ratio_{few} \leq 0.4) \end{aligned} \tag{6}$$

この 3 種類の $ThreadDetermine_{size}$ の値を用いて、GPU_{few} に割り当てるワークアイテムの全体に対する割合は、

$$\begin{aligned} (GPU_{few}\text{に割り当てるワークアイテムの割合}) = PE\_ratio_{few} \\ \times\ ThreadDetermine_{size} \quad (0 < PE\_ratio_{few} \leq 0.4) \end{aligned} \tag{7}$$

で表される。この比によって割り当てるワークアイテム数が整数にならない場合には、その数値の小数第一位の値を四捨五入して整数値にする。GPU_{many} に割り当てるワークアイテム数は、未割り当てのワークアイテム数とする。

例えば、総 PE 数の比が 2 : 5 である 2 台の GPU が同じノード上で使用できるシステムで、カーネルプログラムの計算量が $middle$ サイズであり、1000 個のワークアイテムを発行するプログラムを実行する場合、

$$(GPU_{few}\text{に割り当てるワークアイテムの割合}) = 2/7 \times\ 0.3 \fallingdotseq 0.0857$$

となり、GPU_{few} では 86 個のワークアイテムを、GPU_{many} では 914 個のワークアイテムを実行することになる。

また、2 台の GPU の総 PE 数の比が 2 : 3 よりも開きがない場合には、GPU の性能に差異がないと判断して、処理量を均等に分割するために、

$$\begin{aligned} &(GPU_{few}\text{に割り当てるワークアイテムの割合}) \\ =\ &(GPU_{many}\text{に割り当てるワークアイテムの割合}) \\ =\ &0.5 \quad (0.4 < PE\_ratio_{few} \leq 0.5) \end{aligned} \tag{8}$$

とする。

さらに、3 台以上の GPU を使用する場合には、はじめに、GPU のもつ総 PE 数の少ない GPU を GPU_{few}、その次に少ない GPU を GPU_{many} とし、式 (7) または式 (8) を用いて GPU_{few} に割り当てるワークアイテムの割合を求め、

$$\begin{aligned} &(GPU_{many}\text{に割り当てるワークアイテムの割合}) = \\ &1 - (GPU_{few}\text{に割り当てるワークアイテムの割合}) \end{aligned}$$

とする。次に、GPU_{many} の次に総 PE 数の少ない GPU を新たな GPU_{many} とし、もとの GPU_{many} を GPU_{few} とする。新たな GPU_{few} と GPU_{many} に割り当てるワークアイテムの割合をそれぞれ前と同様の手順で求める。以降も同様の手順で各 GPU に割り当てるワークアイテムの割合を求めていき、最終的にすべての割合の連比を求めることで、全ての GPU に割り当てるワークアイテムの割合を決定する。

4.3 各 GPU 内の CU あたりに割り当てるワークアイテム数とワークアイテムのグループ構成方法

各 GPU 内の CU あたりに割り当てるワークアイテム数であるローカル・ワークサイズの決定方法を策定するために、NDRange の次元数を 1 として実行するカーネルプログラムと、次元数を 2 として実行するカーネルプログラムとに分けてローカル・ワークサイズとカーネルプログラムの実行時間の関係を測定する。

本実験では表 4 に示す 4 種類の GPU カーネルを用いる。

表 5: 各カーネルプログラムの問題サイズ

カーネルプログラム名	問題サイズ
行列積カーネル	1000 × 1000
画像拡大・縮小のための線形補完法カーネル	1000 × 1000
実数ソートカーネル	100000
台形公式による区分求積カーネル	100000

まずは、ワークグループへのワークアイテムのマッピングを x 方向のみの 1 次元を想定して作成した実数ソートカーネルと台形公式による区分求積カーネルを用いて、ローカル・ワークサイズを変化させた際のカーネルプログラムの実行時間を測定した。

実験方法は、各 GPU を単体で使用するシングル GPU プログラムのソースコード中で、グローバル・ワークサイズは表 5 に示す問題サイズに固定し、ローカル・ワークサイズを、4 から 1000 までの数値で、問題サイズであるグローバル・ワークサイズを割り切れる整数で変化させる。ただし、GeForce 8800GTS だけは、設定可能なローカル・ワークサイズの最大値が 512 であるため 4 から 500 までとする。その際のカーネルプログラムの実行時間を計測した。この実験によって得られたローカル・ワークサイズとカーネルプログラムの実行時間の関係を図 9 と図 10 に示す。

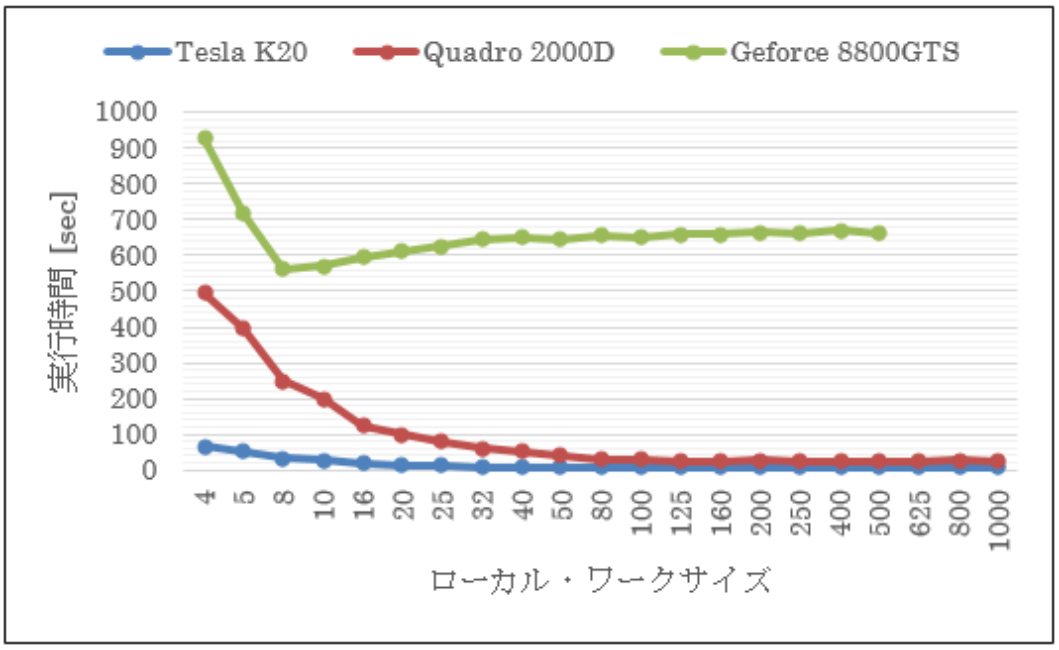

図 9: 実数ソートカーネルのローカル・ワークサイズと実行時間の関係

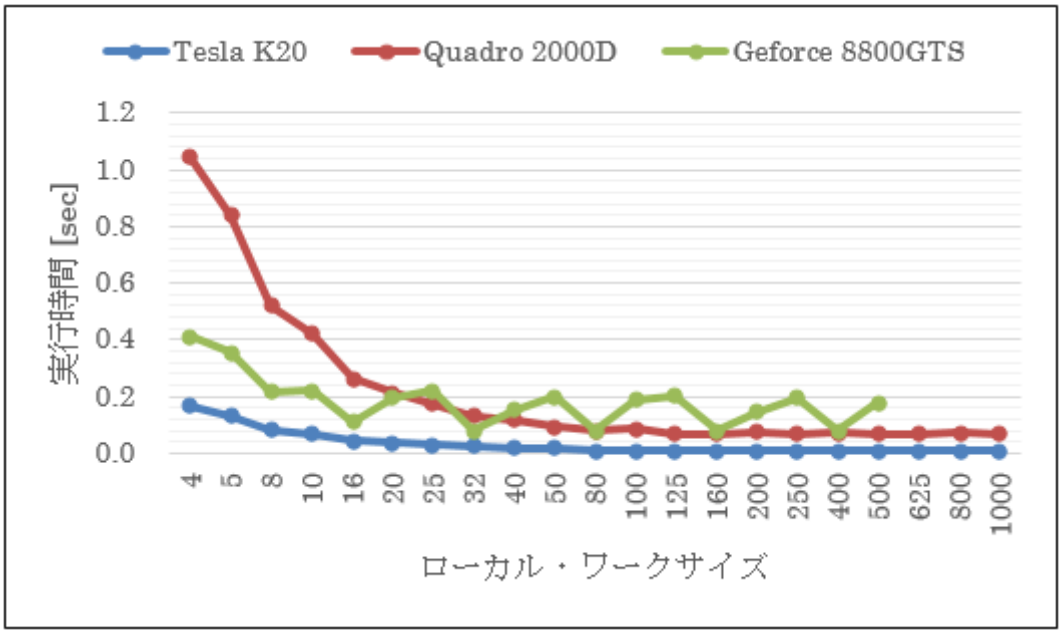

図 10: 台形公式による区分求積カーネルのローカル・ワークサイズと実行時間の関係

図 9 と図 10 から、カーネルプログラムの実行時間は、ローカル・ワークサイズが CU あたりの PE 数に等しい、または初めに超えた値で実行時間が小さい傾向が確認できる。さらに、3 種類の GPU を比較すると、CU 数と CU あたりの PE 数の両方の値が大きい GPU ほど、カーネルプログラムの実行時間が小さい傾向が確認できる。

次に、ワークグループへのワークアイテムのマッピングを x 方向と y 方向の 2 次元を想定して作成した行列積カーネルと線形補完法カーネルを用いてローカル・ワークサイズ、およびワークグループの各次元方向へマッピングしたワークアイテムの個数を変化させた際のカーネルプログラムの実行時間を測定した。

実験方法は、前に述べた 1 次元のカーネルプログラムでの実験方法と同じであるが、ワークグループの各次元へマッピングするワークアイテムの個数の組み合わせは、x 方向、y 方向ともに 1 から 1000 までの数値で、問題サイズであるグローバル・ワークサイズを割り切れる整数の組合せとする。

行列積カーネルでのローカル・ワークサイズを変化させた際の実行時間を、表 6、表 7、表 8 に、線形補完法カーネルでのローカル・ワークサイズを変化させた際の実行時間を、表 9、表 10、表 11 に示す。これらの表は、x 方向へマッピングしたワークアイテムの個数を行に示し、y 方向へマッピングしたワークアイテムの個数を列に示している。また、セルの色が濃いデータは、実行時間が短かった上位 10 件を表しており、最も色の濃いセルが実行時間が最速であることを表している。

これらのローカル・ワークサイズを変化させた際の実行時間の測定結果からは、2 種類のカーネルプログラムで、実行時間が短かった上位 10 件の組み合わせについて、次に挙げる特徴が確認できる。

- 行列積カーネルでは、ワークグループの x 方向にマッピングしたワークアイテムの個数が、y 方向にマッピングしたワークアイテムの個数と比べて多い傾向にあるが、線形補完法カーネルでは、その逆の傾向がある。

  これは 3.2 節で述べたように、カーネルプログラムのメモリアクセスパターンが異なるためであると考えられる。行列積カーネルでは、x 方向に連続したメモリのアドレス領域にアクセスを行う。同じ領域にアクセスを行うワークアイテムが同一のワークグループに多くマッピングされている場合、メモリから

表 6: 行列積カーネルのローカル・ワークサイズと実行時間の関係 (Tesla K20)

(単位：秒)

＼	1	2	4	5	8	10	20	25	40	50	100	125	200	250	500	1000
1	1.64E+02	1.11E+02	5.81E+01	4.67E+01	2.90E+01	2.38E+01	1.22E+01	1.01E+01	6.53E+00	5.69E+00	4.13E+00	3.60E+00	3.44E+00	3.53E+00	3.41E+00	3.20E+00
2	9.33E+01	6.09E+01	3.12E+01	2.52E+01	1.56E+01	1.30E+01	6.23E+00	5.24E+00	4.02E+00	5.08E+00	4.10E+00	3.77E+00	3.13E+00	3.54E+00	2.98E+00	
4	5.13E+01	3.36E+01	1.73E+01	1.39E+01	8.64E+00	8.49E+00	4.99E+00	5.51E+00	4.08E+00	4.86E+00	3.38E+00	3.46E+00	3.15E+00	3.31E+00		
5	4.17E+01	2.78E+01	1.42E+01	1.14E+01	7.50E+00	6.89E+00	5.52E+00	4.98E+00	4.17E+00	4.59E+00	3.85E+00	3.42E+00	2.85E+00			
8	3.16E+01	2.01E+01	1.02E+01	1.29E+01	4.64E+00	6.61E+00	4.83E+00	5.17E+00	3.68E+00	4.05E+00	3.33E+00	3.46E+00				
10	3.27E+01	1.75E+01	1.35E+01	1.06E+01	5.94E+00	7.52E+00	5.06E+00	4.79E+00	3.47E+00	4.15E+00	2.99E+00					
20	2.61E+01	2.84E+01	1.22E+01	1.21E+01	6.62E+00	7.48E+00	4.28E+00	4.49E+00	3.44E+00	3.84E+00						
25	3.29E+01	2.52E+01	1.27E+01	1.18E+01	6.81E+00	7.35E+00	4.34E+00	4.39E+00	3.21E+00							
40	6.01E+01	2.46E+01	1.28E+01	1.20E+01	6.74E+00	7.21E+00	4.51E+00	4.62E+00								
50	5.34E+01	2.51E+01	1.29E+01	1.20E+01	6.62E+00	7.36E+00	5.08E+00									
100	4.97E+01	2.53E+01	1.30E+01	1.22E+01	6.81E+00	7.40E+00										
125	4.98E+01	2.55E+01	1.32E+01	1.23E+01	6.88E+00											
200	4.99E+01	2.56E+01	1.35E+01	1.25E+01												
250	5.07E+01	2.59E+01	1.33E+01													
500	6.29E+01	2.68E+01														
1000	1.00E+02															

表 7: 行列積カーネルのローカル・ワークサイズと実行時間の関係 (Quadro 2000D)

(単位：秒)

＼	1	2	4	5	8	10	20	25	40	50	100	125	200	250	500	1000
1	1.06E+03	5.82E+02	3.74E+02	3.13E+02	2.02E+02	1.64E+02	8.29E+01	6.88E+01	4.65E+01	3.85E+01	2.36E+01	1.89E+01	1.73E+01	1.50E+01	1.49E+01	1.85E+01
2	5.66E+02	3.15E+02	1.98E+02	1.64E+02	1.04E+02	8.42E+01	4.17E+01	3.55E+01	2.45E+01	2.27E+01	1.72E+01	1.59E+01	1.57E+01	1.56E+01	1.81E+01	
4	3.45E+02	1.94E+02	1.16E+02	9.48E+01	6.02E+01	4.19E+01	2.32E+01	1.99E+01	1.67E+01	1.80E+01	1.64E+01	1.69E+01	1.95E+01	1.79E+01		
5	3.01E+02	1.70E+02	9.90E+01	8.10E+01	4.16E+01	3.54E+01	1.89E+01	1.81E+01	1.67E+01	1.75E+01	1.67E+01	1.75E+01	1.64E+01			
8	2.42E+02	1.35E+02	7.39E+01	5.31E+01	3.09E+01	2.39E+01	1.68E+01	1.77E+01	1.71E+01	1.77E+01	1.97E+01	1.81E+01				
10	2.30E+02	1.24E+02	5.17E+01	4.31E+01	2.28E+01	2.25E+01	1.70E+01	1.77E+01	1.67E+01	1.77E+01	1.66E+01					
20	2.19E+02	9.32E+01	3.33E+01	3.51E+01	1.97E+01	2.26E+01	1.64E+01	1.71E+01	2.01E+01	1.76E+01						
25	2.16E+02	7.73E+01	3.33E+01	3.83E+01	2.05E+01	2.47E+01	1.63E+01	1.69E+01	1.66E+01							
40	1.88E+02	8.42E+01	4.65E+01	4.31E+01	2.16E+01	2.39E+01	2.06E+01	1.89E+01								
50	1.63E+02	9.18E+01	4.76E+01	4.34E+01	2.16E+01	2.54E+01	1.74E+01									
100	2.71E+02	1.42E+02	4.80E+01	4.65E+01	2.26E+01	2.31E+01										
125	3.49E+02	1.71E+02	5.90E+01	4.71E+01	2.16E+01											
200	4.15E+02	1.77E+02	4.82E+01	5.13E+01												
250	4.79E+02	2.25E+02	6.67E+01													
500	5.46E+02	2.39E+02														
1000	5.45E+02															

表 8: 行列積カーネルのローカル・ワークサイズと実行時間の関係 (GeForce 8800GTX)

(単位：秒)

＼	1	2	4	5	8	10	20	25	40	50	100	125	200	250	500
1	4.35E+02	2.53E+02	1.95E+02	2.17E+02	1.45E+02	1.87E+02	1.77E+02	1.98E+02	1.47E+02	1.96E+02	1.83E+02	2.05E+02	1.51E+02	1.99E+02	1.90E+02
2	3.38E+02	2.18E+02	2.00E+02	2.08E+02	1.99E+02	2.00E+02	1.88E+02	2.02E+02	1.53E+02	1.96E+02	1.79E+02	2.05E+02	1.46E+02	2.02E+02	
4	2.79E+02	2.09E+02	2.04E+02	2.05E+02	2.04E+02	2.04E+02	1.94E+02	2.06E+02	1.55E+02	1.95E+02	1.78E+02	2.06E+02			
5	2.61E+02	2.12E+02	2.01E+02	2.04E+02	1.95E+02	2.06E+02	1.94E+02	2.06E+02	1.56E+02	1.98E+02	1.81E+02				
8	2.18E+02	1.99E+02	2.00E+02	2.03E+02	2.03E+02	2.05E+02	1.93E+02	2.05E+02	1.55E+02	1.95E+02					
10	2.01E+02	1.95E+02	2.00E+02	2.01E+02	2.02E+02	2.06E+02	1.92E+02	2.06E+02	1.53E+02	1.94E+02					
20	2.04E+02	2.05E+02	2.02E+02	2.05E+02	2.04E+02	2.06E+02	1.93E+02	2.04E+02							
25	2.13E+02	2.06E+02	2.04E+02	2.06E+02	2.03E+02	2.08E+02	1.94E+02								
40	2.21E+02	2.11E+02	2.05E+02	2.06E+02	2.06E+02	2.07E+02									
50	2.29E+02	2.16E+02	2.06E+02	2.08E+02	2.05E+02	2.07E+02									
100	2.74E+02	2.24E+02	2.08E+02	2.08E+02											
125	2.88E+02	2.27E+02	2.08E+02												
200	3.15E+02	2.30E+02													
250	3.26E+02	2.33E+02													
500	3.54E+02														

表 9: 線形補完法カーネルのローカル・ワークサイズと実行時間の関係 (Tesla K20)

(単位：秒)

＼	1	2	4	5	8	10	20	25	40	50	100	125	200	250	500	1000
1	2.95E-01	1.60E-01	1.04E-01	9.39E-02	7.45E-02	6.93E-02	6.33E-02	6.56E-02	8.25E-02	7.16E-02	7.84E-02	8.17E-02	8.04E-02	8.10E-02	8.20E-02	8.47E-02
2	1.48E-01	7.99E-02	5.21E-02	4.70E-02	3.73E-02	3.47E-02	4.84E-02	5.42E-02	6.97E-02	7.30E-02	7.48E-02	7.69E-02	7.87E-02	7.64E-02	7.88E-02	
4	7.39E-02	4.00E-02	2.61E-02	2.36E-02	1.87E-02	2.29E-02	3.57E-02	4.54E-02	6.08E-02	5.85E-02	6.57E-02	6.29E-02	6.70E-02	6.91E-02		
5	5.92E-02	3.20E-02	2.09E-02	1.88E-02	1.87E-02	1.88E-02	3.16E-02	4.20E-02	5.09E-02	5.23E-02	5.50E-02	5.43E-02	5.93E-02			
8	3.70E-02	2.00E-02	1.31E-02	1.39E-02	1.17E-02	1.63E-02	2.80E-02	3.57E-02	4.40E-02	4.51E-02	4.52E-02	4.88E-02				
10	2.96E-02	1.60E-02	1.15E-02	1.12E-02	1.29E-02	1.45E-02	2.62E-02	3.36E-02	4.12E-02	4.15E-02	4.25E-02					
20	1.48E-02	9.06E-03	7.25E-03	9.34E-03	9.87E-03	1.30E-02	2.28E-02	2.87E-02	3.57E-02	3.51E-02						
25	1.19E-02	7.26E-03	7.47E-03	7.69E-03	1.09E-02	1.20E-02	2.26E-02	2.83E-02	3.47E-02							
40	8.58E-03	5.97E-03	5.81E-03	8.16E-03	9.61E-03	1.21E-02	2.31E-02	2.79E-02								
50	6.87E-03	6.19E-03	6.46E-03	7.44E-03	9.98E-03	1.19E-02	2.22E-02									
100	6.00E-03	5.42E-03	6.04E-03	7.35E-03	9.78E-03	1.17E-02										
125	4.82E-03	4.96E-03	5.82E-03	7.27E-03	9.59E-03											
200	5.27E-03	5.15E-03	5.85E-03	7.24E-03												
250	4.81E-03	4.95E-03	5.57E-03													
500	4.82E-03	4.89E-03														
1000	4.79E-03															

表 10: 線形補完法カーネルのローカル・ワークサイズと実行時間の関係 (Quadro 2000D)

(単位：秒)

＼	1	2	4	5	8	10	20	25	40	50	100	125	200	250	500	1000
1	1.83E+00	9.36E−01	5.50E−01	4.89E−01	3.62E−01	3.26E−01	2.57E−01	2.49E−01	2.02E−01	1.81E−01	1.76E−01	1.80E−01	1.80E−01	1.78E−01	1.87E−01	2.24E−01
2	9.13E−01	4.68E−01	2.75E−01	2.45E−01	1.81E−01	1.63E−01	1.37E−01	1.41E−01	1.50E−01	1.61E−01	1.58E−01	1.63E−01	1.60E−01	1.63E−01	1.79E−01	
4	4.56E−01	2.34E−01	1.37E−01	1.22E−01	9.06E−02	8.74E−02	9.52E−02	1.16E−01	1.39E−01	1.37E−01	1.38E−01	1.45E−01	1.36E−01	1.42E−01		
5	3.65E−01	1.87E−01	1.10E−01	9.79E−02	7.88E−02	7.07E−02	9.22E−02	1.08E−01	1.32E−01	1.45E−01	1.43E−01	1.36E−01	1.37E−01			
8	2.28E−01	1.17E−01	6.88E−02	6.66E−02	4.90E−02	5.30E−02	8.13E−02	1.01E−01	1.24E−01	1.25E−01	1.26E−01	1.30E−01				
10	1.83E−01	9.37E−02	5.99E−02	5.33E−02	4.62E−02	5.19E−02	7.95E−02	1.01E−01	1.23E−01	1.27E−01	1.26E−01					
20	9.13E−02	5.18E−02	3.53E−02	3.97E−02	3.65E−02	4.62E−02	7.88E−02	9.91E−02	1.25E−01	1.24E−01						
25	7.31E−02	4.15E−02	3.55E−02	3.18E−02	4.08E−02	4.31E−02	7.86E−02	9.87E−02	1.24E−01							
40	5.06E−02	3.17E−02	2.76E−02	3.48E−02	3.65E−02	4.38E−02	7.93E−02	1.00E−01								
50	4.05E−02	3.26E−02	3.09E−02	3.18E−02	3.80E−02	4.30E−02	7.83E−02									
100	3.26E−02	2.84E−02	2.88E−02	3.18E−02	3.98E−02	4.41E−02										
125	2.61E−02	2.60E−02	2.82E−02	3.22E−02	3.83E−02											
200	2.85E−02	2.65E−02	3.06E−02	3.26E−02												
250	2.60E−02	2.60E−02	2.94E−02													
500	2.60E−02	2.73E−02														
1000	2.71E−02															

表 11: 線形補完法カーネルのローカル・ワークサイズと実行時間の関係 (GeForce 8800GTX)

(単位：秒)

＼	1	2	4	5	8	10	20	25	40	50	100	125	200	250	500
1	1.18E+00	6.06E-01	5.99E-01	6.05E-01	7.20E-01	6.59E-01	7.70E-01	7.88E-01	7.68E-01	7.64E-01	7.59E-01	8.22E-01	7.53E-01	7.68E-01	7.43E-01
2	5.91E-01	3.85E-01	3.98E-01	4.03E-01	4.69E-01	4.68E-01	6.57E-01	7.75E-01	7.67E-01	7.68E-01	7.66E-01	8.23E-01	7.62E-01	7.66E-01	
4	3.28E-01	3.27E-01	3.36E-01	3.51E-01	3.42E-01	4.71E-01	6.66E-01	7.64E-01	7.81E-01	7.84E-01	7.83E-01	8.19E-01			
5	3.20E-01	3.24E-01	3.28E-01	3.36E-01	4.35E-01	4.26E-01	6.22E-01	7.65E-01	7.91E-01	7.84E-01	7.82E-01				
8	3.15E-01	3.14E-01	3.16E-01	3.65E-01	3.42E-01	4.14E-01	6.32E-01	7.74E-01	7.95E-01	8.21E-01					
10	3.16E-01	3.16E-01	3.36E-01	3.38E-01	3.64E-01	3.95E-01	6.25E-01	7.81E-01	8.02E-01	8.20E-01					
20	3.22E-01	3.27E-01	3.24E-01	3.39E-01	3.43E-01	3.73E-01	5.96E-01	7.49E-01							
25	3.26E-01	3.28E-01	3.30E-01	3.27E-01	3.50E-01	3.69E-01	6.09E-01								
40	3.32E-01	3.25E-01	3.27E-01	3.26E-01	3.44E-01	3.51E-01									
50	3.32E-01	3.31E-01	3.25E-01	3.24E-01	3.39E-01	3.51E-01									
100	3.37E-01	3.20E-01	3.20E-01	3.20E-01											
125	3.38E-01	3.24E-01	3.20E-01												
200	3.33E-01	3.18E-01													
250	3.36E-01	3.22E-01													
500	3.35E-01														

L1 キャッシュにバーストされたデータを読み込むため、メモリアクセスレイテンシが削減され、カーネルプログラムの実行時間が小さくなったと考えられる。同様に、線形補完法カーネルカーネルでは、ワークグループの y 方向に多くのワークアイテムをマッピングすることで、各ワークアイテムのメモリアクセス範囲の重なりが、x 方向に多くマッピングした場合に比べて多くなったと考えられる。

- 平均で 3 割の組み合わせが、ローカル・ワークサイズが 32 の倍数である。

  NVIDIA 社の GPU ではローカル・ワークサイズを 32 の倍数に設定することが高いパフォーマンスを得られることが多い [11]。これは、NVIDIA 社の GPU では 32 個のスレッドを物理的に一単位として実行させているためである。
  また、AMD 社の GPU では同様の理由で、ローカル・ワークサイズを 64 の倍数に設定することが良いとされている [12]。

これらの理由と 3.2 節で述べた性質から、本研究では、NDRange の設定した次元数が 2 の場合には、ワークグループの各次元へマッピングするワークアイテムの個数は、ワークアイテムの総数が 32 の倍数になり、かつその総数が可能な限り大きくなるように決定する。さらにカーネルプログラムのメモリアクセスパターンに応じて、ワークグループの x 方向または y 方向の一方向に多くのワークアイテムをマッピングして、x 方向と y 方向にマッピングするワークアイテムの個数の組み合わせを決定する。

詳細な決定の手順は下記の通りとする。

NDRange の設定した次元数が 1 の場合

1. グローバル・ワークサイズに設定した値の約数をすべて抽出する。
2. 1 で抽出した数の中から、使用する GPU に搭載されている CU あたりの PE 数に等しい、または初めに超えた値をワークグループの x 方向へマッピングするワークアイテムの個数、すなわちローカル・ワークサイズに設定する。

NDRange の設定した次元数が 2 の場合

1. NDRange の x 方向にマッピングしたワークアイテムの個数の約数と、y 方向にマッピングしたワークアイテムの個数の約数をすべて算出する。
2. 1 で算出した数のすべての組み合わせ（約数$_x$, 約数$_y$）の中から、

   約数$_x$×約数 $_y \leq$ (使用する *GPU* に設定可能なローカル・ワークサイズの最大数)

   を満たす組み合わせを抽出する。
3. 2 で抽出した組み合わせの中から、(約数$_x$×約数 $_y$)$mod(32) = 0$ を満たす組み合わせを抽出する。
4. 3 で抽出した組み合わせの中から、カーネルプログラムのメモリアクセスパターンが x 方向に多くのアクセスをするパターンの場合には、最大の 約数$_x$ を、y 方向に多くのアクセスをするパターンの場合には、最大の 約数$_y$ を要素にもつ組み合わせを抽出する。
5. 4 で抽出した組み合わせの中から、約数$_x$×約数 $_y$ が最大となる組み合わせに決定し、ワークグループの x 方向にマッピングするワークアイテムの個数をこのときの 約数$_x$ とし、y 方向にマッピングするワークアイテムの個数をこのときの 約数$_y$ とする。

ただし、上述の方法によって得られる値が存在しない場合は、カーネルプログラムのメモリアクセスパターンが x 方向に多くのアクセスをするパターンの場合には、ワークグループの x 方向にマッピングするワークアイテムの個数を 1 で算出した 約数$_x$ の最大値とし、y 方向にマッピングするワークアイテムの個数を 1 とする。カーネルプログラムのメモリアクセスパターンが y 方向に多くのアクセスをするパターンの場合には、ワークグループの x 方向にマッピングするワークアイテムの個数を 1 とし、y 方向にマッピングするワークアイテムの個数を 1 で算出した 約数$_y$ の最大値とする。

上述の方法で、各 GPU 内のローカル・ワークサイズを決定すると、表 3 の GPU で表 5 のカーネルプログラム実行させた際には、表 12 に決まる。

表 12 のローカル・ワークサイズで各カーネルプログラム実行させた際の実行時間は、最も実行時間が短くなるローカル・ワークサイズで実行させた際の実行時間に比べて、実数ソートカーネルと区分求積カーネルでは、平均で 1.57 倍であり、行列積カーネルと線形補完法カーネルでは、平均で 1.06 倍となった。

表 12: 本決定方法で決定されるローカル・ワークサイズ

	Tesla K20	Quadro 2000D	GeForce 8800GTS
行列積カーネル	200 × 4	200 × 4	200 × 2
線形補完法カーネル	2 × 200	2 × 200	2 × 200
実数ソートカーネル	200	50	8
区分求積カーネル	200	50	8

5　提案するパラメータ調整機構

本章では、前章で述べたパラメータチューニング手法によって得られた適切なパラメータを、対象とするOpenCLプログラムのソースコードに反映させるパラメータ調整機構について述べる。

5.1　パラメータ調整機構の概要

提案するパラメータ調整機構では、前章で述べたパラメータチューニング手法を用いて、計算に使用するGPUと各GPUのグローバル・ワークサイズ、ローカル・ワークサイズ、NDRangeの各次元方向に割り当てるワークアイテム数を決定し、1台のGPUを使用するOpenCLプログラムのソースコードを、決定した各ワークサイズで実行するOpenCLプログラムに自動的に変換する。

本機構では、1台のGPUを使用するOpenCLプログラムのソースコードと、カーネルプログラムのメモリアクセスパターン、カーネルプログラムの計算量（大、中、小）、使用可能な最大ノード数を入力とし、適切なパラメータで実行するようにしたホストプログラムのソースコードを出力する。

また、本機構は大きく分けて、GPUアーキテクチャの情報取得部と、ホストプログラムのソースコード解析部、適切なパラメータ決定部、ホストプログラムのソースコード変換部の4つの実装から構成される。

GPUアーキテクチャの情報取得部

前章で述べたパラメータチューニング手法では、各ノードに搭載されているGPUのCU数とCUあたりのPE数をもとに適切なパラメータを決定する。また、各GPUに設定可能なローカル・ワークサイズの最大値、および、NDRangeの各次元方向に割り当てるワークアイテム数の最大値も適切なパラメータの決定の際に必要となる。

これらの値は、GPUの種類やアーキテクチャの世代によって異なるため、予めこれらの値を取得しておくことで異なる計算機環境下における移植性の向上が図れる。

GPUアーキテクチャの情報取得部が取得した各値は、適切なパラメータ決定部に受け渡される。

ホストプログラムのソースコード解析部

適切なパラメータで実行するようにホストプログラムのソースコードを変換するために、ホストプログラムのソースコードの構造を解析する必要がある。

具体的には、入力されたホストプログラムのソースコードから OpenCL API の各関数コールが記述された行番号と各関数の実引数の値、および、各関数が含まれているスコープの開始行と終了行、GPU 内のメモリからホスト内のメモリに転送する際に転送先のメモリアドレスとして指定されている変数名を取得する。これらの情報は、ホストプログラムのソースコード変換部に受け渡される。

また、入力されたホストプログラムがカーネルプログラムを実行する際に設定してあったグローバル・ワークサイズ、すなわち実行する全ワークアイテム数、および NDRange の次元数を取得する。この数値は、適切なパラメータ決定部に受け渡される。

適切なパラメータ決定部

適切なパラメータ決定部は、GPU アーキテクチャの情報取得部とホストプログラムのソースコードの解析部から得たれた使用可能な各 GPU の CU 数と CU あたりの PE 数、実行する全ワークアイテム数をもとに、4 章で述べたパラメータチューニング手法を用いて、計算に使用する GPU と各 GPU のグローバル・ワークサイズ、ローカル・ワークサイズと NDRange の各次元方向に割り当てるワークアイテム数の順に適切な値を決定する。

ホストプログラムのソースコード変換部

ホストプログラムのソースコードの解析部によって決定された適切なパラメータで、指定したカーネルプログラムを実行するようにホストプログラムのソースコードを書き換える。

上で述べたこれらの実装と全体の処理とデータの流れを図 11 に示す。

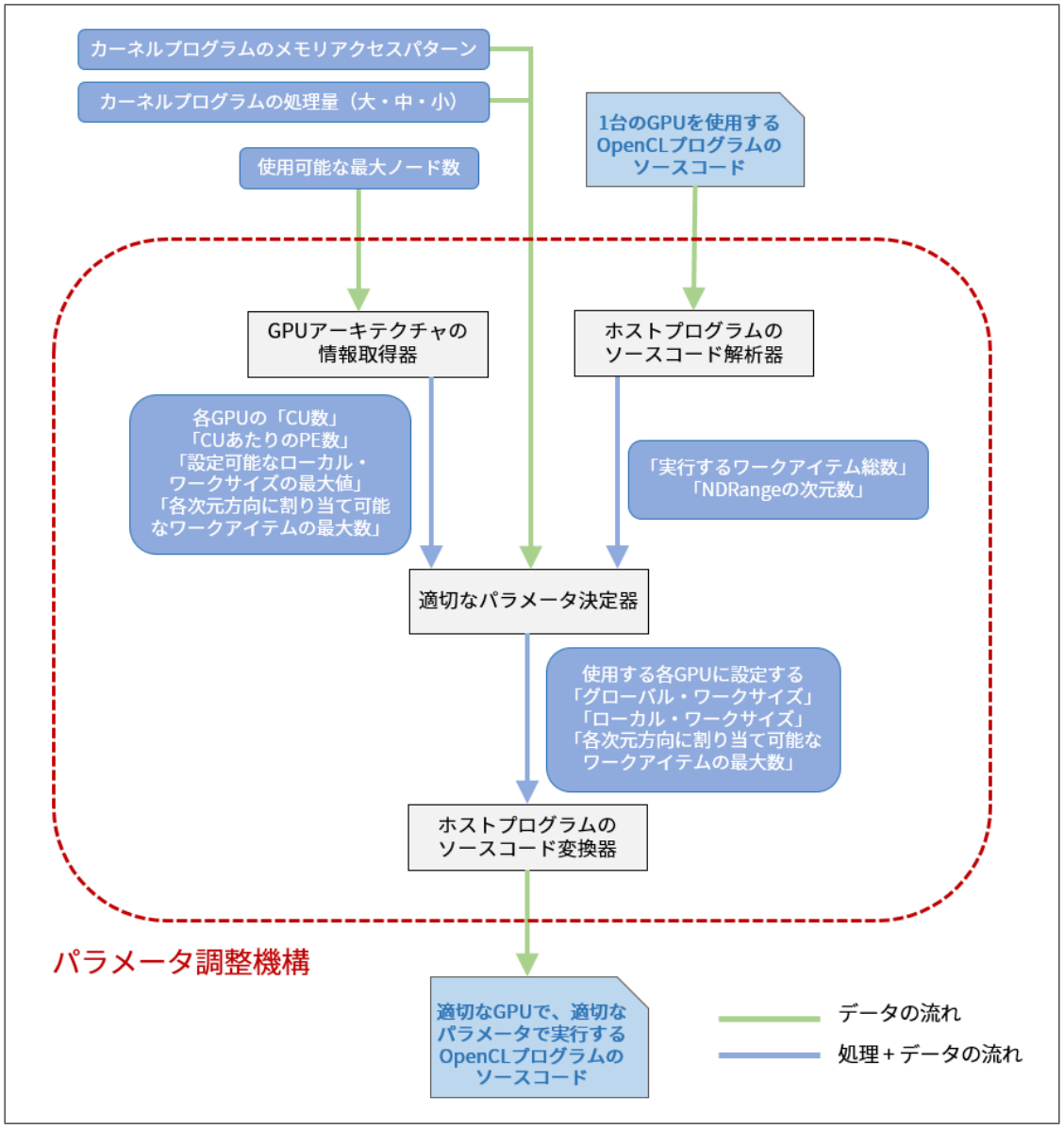

図 11: 提案するパラメータ調整機構の実装と処理の流れ

5.2　パラメータ調整機構の実装

パラメータ調整機構を構成する GPU アーキテクチャの情報取得部と、ホストプログラムのソースコード解析部、適切なパラメータ決定部、ホストプログラムのソースコード変換部の実装について述べる。これらの実装は、全て C++11 で行った。

GPU アーキテクチャの情報取得部の実装

各 GPU に搭載されている CU 数と設定可能なローカル・ワークサイズの最大値、NDRange の各次元方向に割り当てるワークアイテム数の最大値は、OpenCL デバイスの情報を取得する OpenCL の clGetDeviceInfo 関数を使用して取得する。

GPU に搭載されている CU あたりの PE 数は、ハードウェアアーキテクチャの世代によって決まっているため、同関数を使用してハードウェアアーキテクチャの世代である Compute Capability 値を取得する。

例えば、Kepler アーキテクチャの GPU は Compute Capability 値が 3.0 または 3.5 と設定されているため、Compute Capability 値がこれらの値ならば、GPU の CU あたりの PE 数は 192 と求まる。

ホストプログラムのソースコード解析部の実装

入力された 1 台の GPU を使用する OpenCL プログラムのホストプログラムのソースコードから、文字列マッチングによって各種類の OpenCL API の関数コールの行番号と、その行での関数コールの各実引数の文字列をすべて取得する。これらの行番号や文字列は、ホストプログラムのソースコード変換の際に、複数台の GPU の使用や複数のノード上で実行するために OpenCL API の各関数のコール回数や引数を書き換える際に使用される。また、カーネル関数を指定したデバイスで実行させるためのサブミットを行う clEnqueueNDRangeKernel 関数では、実引数として書かれている設定されたグローバル・ワークサイズ、および設定された NDRange の次元数を取得できる。

ただし、本機構のソースコード解析において各実引数に指定されている数値は、数値リテラル、または、プリプロセッサによって記号定数の置換を行う #define 文で定義された記号定数のみが解析可能である。

適切なパラメータ決定部の実装

4.2 節と 4.3 節で提案した使用する GPU と各 GPU ごとのグローバル・ワークサイズ、ローカル・ワークサイズ、NDRange の各次元方向に割り当てるワークアイテム数の決定方法にしたがって、これらのパラメータをホストプログラムのソースコー

ド変換前に決定する。

ホストプログラムのソースコード変換部の実装

1 台の GPU を使用するホストプログラムを、決定された使用する GPU とパラメータで実行するようにホストプログラムのソースコードの書き換えを行う。

複数台の GPU を使用するように変換する場合には、使用する GPU 数の数だけ生成しなければななならいオブジェクトであるバッファオブジェクトやカーネルオブジェクトなどを、ソースコード 2 の 3 から 4 行目を、8 から 16 行目のように、使用する GPU 数の数だけ生成するように変換する。使用する GPU 数の数だけ実行する、clEnqueueNDRangeKernel 関数や、GPU 内のメモリへのデータ転送などの関数コールも同様に、使用する GPU 数の数だけ実行するように変換する。

ソースコード 2: ホストプログラムのソースコードの変換例 1 : 使用する GPU 数の数だけ生成しなければななならいオブジェクトの生成

```
1  // 変換前 :バッファオブジェクトとカーネルオブジェクトの生成
2  // 前後は省略
3  buffer = clCreateBuffer( 実引数は省略する );
4  kernel = clCreateKernel( 実引数は省略する );
5
6  // 変換後 :バッファオブジェクトとカーネルオブジェクトをそれぞれ 3個生成
7  // 前後は省略
8  cl_mem translated_buffer[3];
9  for(int i=0; i<3; ++i) {
10         translated_buffer[i] = clCreateBuffer( 実引数は省略する );
11 }
12
13 cl_kernel translated_kernel[3];
14 for(int i=0; i<3; ++i) {
15         translated_kernel[i] = clCreateKernel( 実引数は省略する );
16 }
```

また、clEnqueueNDRangeKernel 関数での NDRange に対するワークアイテムの ID と、GPU 内のメモリからのデータ転送を行う関数での転送先のメモリアドレス

位置に対しては、オフセット値を指定するようにする。ソースコード 3 のように、パラメータの決定時にオフセット値は決定しておき、ソースコードの変換時にその値をソースコードに書き加えることで、ホストプログラム内でオフセット値を計算する時間を削減する。

ソースコード 3: ホストプログラムのソースコードの変換例 2 : 複数 GPU を使用する際の、ワークアイテム ID のオフセット値の調整

```
1  // clEnqueueNDRangeKernel( ... , offset, global_work_size, ... )
2  //
3  // 変換前 :カーネルプログラムを 1台のGPUで実行するためのサブミット処理
4  // 前後と、実引数の一部は省略。
5  clEnqueueNDRangeKernel( ... , 0, 1024, ... );
6
7  // 変換後 :カーネルプログラムを 4台の
       GPUで処理を均等に実行するためのサブミット処理
8  // 前後は省略
9  const int NUM_GPUS = 4;
10 const int TRANSLATED_GLOBAL_WORK_SIZE[NUM_GPUS] = {256,
       256, 256, 256};
11 const int TRANSLATED_WORK_ITEM_ID_OFFSET[NUM_GPUS] = {0,
       256, 512, 768};
12 for(int i=0; i<NUM_GPUS; ++i) {
13         clEnqueueNDRangeKernel( ... ,
               TRANSLATED_WORK_ITEM_ID_OFFSET[i],
               TRANSLATED_GLOBAL_WORK_SIZE[i], ... );
14 }
```

さらに、起点ノードとは異なるノードに搭載された GPU を 1 台以上使用するように変換する場合には、OpenCL API の関数が呼ばれるスコープの最上部と最下部に、MPI を使用するための変数の宣言と初期化、終了処理を行う MPI の記述を書き加える。OpenCL API の関数が呼ばれる部分は、使用する GPU が搭載されているノード上で実行するプロセスランクでのみ実行されるようにする。

加えて、OpenCL API の呼び出し処理の直前行と直後行で、使用する GPU が搭載されているノードと起点ノードとの間でのデータ転送処理を書き加える。このノー

ド間の計算結果のデータ転送時にも、転送先のメモリアドレス位置に対して事前に計算したオフセット値を指定する。

6　おわりに

本章では、本研究の総括と今後の課題について述べる。

6.1　まとめ

本研究では、OpenCL プログラミングにおける性能パラメータの中から、プログラムの実行時間への影響が大きいとされているグローバル・ワークサイズとローカル・ワークサイズ、ワークグループの各次元方向へマッピングするワークアイテムの個数、これら 3 種の性能パラメータの値が実行時間に与える影響の傾向を調べることで、これらの性能パラメータの決定方法を策定した。

また、高速な GPU プログラムの開発コスト低減を目的とし、1 台の GPU を使用する OpenCL プログラムのホストプログラムのソースコードを、策定した性能パラメータのチューニングを行い、同一、または異なるノード上に分散して搭載されている複数台の GPU を使用するソースコードに変換する機構を実装した。

6.2　今後の課題

各 GPU に割り当てるワークアイテムの割合を求めるためのモデル化の精度向上

4.2 節では、GPU に搭載されている総 PE 数が 2 : 3 である 2 台の GPU を使用して、GPU に搭載されている総 PE 数の割合とカーネルプログラムの計算量による各 GPU に割り当てるワークアイテムの割合のモデル化を行った。この 2 台の GPU とは総 PE 数の異なる GPU を用いて、複数のモデル化を行うことで、モデル化の精度向上が期待できる。また、その GPU に搭載されている総 PE 数のみを GPU の性能指標としていたが、GPU のクロック周波数や起点ノードからのデータ転送速度なども加味した性能指標とすることが、性能パラメータチューニングの精度向上につながる可能性がある。

ホストプログラムのソースコード変換機構のユーザー入力項目数の削減

ホストプログラムのソースコード変換機構には、カーネルプログラムのメモリアクセスパターン、およびカーネルプログラムの計算量（大、中、小）を入力する設計にしているが、これらをカーネルプログラムのソースコードなどから自動的に取得することで、ユーザー入力項目数を減らすことが可能になると考えられる。

そのために、カーネルプログラムの構文、構造解析器によるソースコード解析や、使用するレジスタ数の取得、各 GPU メーカーが公表しているプロファイラツールの利用などで実現できる可能性がある。

性能パラメータのチューニングのためのホストプログラムのソースコード解析器の作成

ホストプログラムのソースコード変換のためのソースコード解析では、構文解析などを行わずに関数名との文字列マッチングによって、変換箇所の特定や変数名の取得を行っている。そのため、一般的な変数からはその値が取得できない問題などが残されている。

構文解析などのソースコード解析手法の適用によって、入力可能なホストプログラムの制限が緩和されると予想される。

謝辞

本研究を進めていくにあたって、多くのご指導と研究の指針を示してくださり、加えて実験手法やその結果の考察に関する的確なご指摘を頂きました本多弘樹教授に深く御礼申し上げます。

また、現東京大学の近藤正章准教授、及び、現早稲田大学の和田康孝助教には、研究内容とその方向性について的確なご指摘をしていただいたことに深く感謝いたします。

また、研究から日常の悩みまで、多方面に渡って多くの支援や励みをいただいた高性能コンピューティング学講座の皆様に感謝いたします。

参考文献

[1] NVIDIA Corporation. CUDA, 2015. http://www.nvidia.co.jp/object/cuda-jp.html.

[2] Cyberlink Corporation. PowerDirector 13 Ultra, 2015. http://jp.cyberlink.com/products/powerdirector-ultra/features_ja_JP.html.

[3] Adobe Corporation. Photoshop Help, 2015. http://helpx.adobe.com/jp/photoshop/kb/234358.html.

[4] M. Potmesil and E. M. Hoffert, “The Pixel Machine: A Parallel Image Computer,” Proceedings of SIGGRAPH89, pp.69-78, 1989.

[5] AMD. AMD Brook+, 2015. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/AMD-Brookplus.pdf.

[6] 池田 成樹. OpenCL 並列プログラミング. 株式会社カットシステム, 2010.

[7] Microsoft. DirectCompute, 2015. http://blogs.msdn.com/b/chuckw/archive/2010/07/14/directcompute.aspx.

[8] NVIDIA Corporation. OpenCL Programming Guide for the CUDA Architecture Version 4.2, 2015. http://hpc.oit.uci.edu/nvidia-doc/sdk-cuda-doc/OpenCL/doc/OpenCL_Programming_Guide.pdf.

[9] The Khronos Group. OpenCL - the open standard for parallel programming of heterogeneous systems -, Dec 2014. http://www.khronos.org/opencl/.

[10] 松井 南実, 富永 浩文, 中村 あすか , 篠塚 研太, 前川 仁孝. GPU のキャッシュヒット率向上による DEM の高速化. 全国大会講演論文集 2012(1), pp. 215-217, 2012.

[11] NVIDIA Corporation. NVIDIA OpenCL Best Practices Guide Version 1.0, 2015. http://www.nvidia.com/content/cudazone/cudabrowser/downloads/papers/nvidia_opencl_bestpracticesguide.pdf.

[12] AMD. OpenCL Optimization Guide, 2015. http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/opencl-optimization-guide/.

[13] 本間 咲来, 須田 礼仁. 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. 情報処理学会研究報告, 2011-HPC-129, pp. 1-7, 2013.

[14] Hyeran Jeon，Yinglong Xia，Viktor K. Prasanna.Parallel Exact Interface on a CPU-GPGPU Heterogenous System.2010 39th International Conference on Parallel Processing，pp． 61-70，2010.

[15] Snaider Carrillo, Jakob Siegel, Xiaoming Li. A control-structure splitting optimization for GPGPU. CF '09 Proceedings of the 6th ACM conference on Computing frontiers, pp. 147-150, 2009.

[16] 伊藤 信悟，伊野 文彦，萩原 兼一．GPGPU アプリケーションの開発を支援するための性能モデル．情報処理学会論文誌コンピューティングシステム，第 48 巻，pp．235-246，2007.

[17] 島田 大地，遠藤 敏夫，丸山 直也，松岡 聡．OpenCL を用いた異種 GPU における性能特性に応じた最適化．情報処理学会研究報告. 計算機アーキテクチャ研究会報告，第 23 巻，pp．1-7，2010.

[18] Yaohung M. Tsai, Weichung Wang， Ray-Bing Chen.Tuning Block Size for QR Factorization on CPU-GPU Hybrid Systems．2012 IEEE 6th International Symposium on Embedded Multicore Socs，pp．205-211，2012.

[19] 富田 翔．GPU プログラムのスレッド数最適化に関する研究. Master's thesis，電気通信大学大学院情報システム学研究科情報システム基盤学専攻，2011

[20] 岩下 光弘．GPU 向け並列化プログラミング環境におけるパラメータチューニングに関する研究. Master's thesis，電気通信大学大学院情報システム学研究科情報システム基盤学専攻，2013.