卒業研究論文

OpenCL における
処理量自動調整によるデータ並列処理の高速化

指導教員　　津邑 公暁 准教授
松尾 啓志 教授

名古屋工業大学 工学部 情報工学科
平成 19 年度入学　19115066 番

澤田 晃平

平成 24 年 2 月 8 日

OpenCL における処理量自動調整によるデータ並列処理の高速化

澤田 晃平

内容梗概

ゲート遅延に対する配線遅延の相対的な増大から，動作周波数の向上だけではマイクロプロセッサの速度向上を見込めなくなってきた．また，集積回路の微細化に伴う消費電力や発熱量の増大から動作周波数自体の向上も困難になってきている．こうした中で，SIMD などのデータ並列性に基づく高速化手法や多数のコアを搭載したメニーコアプロセッサなどが注目されてきた．

これを受け，画像処理専用プロセッサ GPU や Cell/B.E. など，SIMD 命令が利用可能なコアを多数搭載した特殊な構成を持つアーキテクチャが登場した．このようなアーキテクチャを搭載した実行環境は，1 つの実行環境の中に複数の計算ユニットと，それらを管理，制御するホストと呼ばれるコアの異なる種類のコアを搭載しているため，こうしたアーキテクチャ構成はヘテロジニアスと表現される．

こうした実行環境は汎用 CPU のみに処理を割り当てる場合に比べ，ピーク処理性能が高く，コスト対性能比や電力対性能比に優れているため，様々な分野で活躍している．このようなヘテロジニアスな実行環境を用いて，並列処理をさせる需要の高まりに応えるため，これまで様々な開発環境が提供されてきた．しかしそのほとんどが特定の製品にしか対応していないため，移植性が低く，プログラマは実行環境毎にプログラムを再実装しなければならないという問題がある．

そこで本論文では，ヘテロジニアスな実行環境を扱うことが可能な並列化フレームワークの一つである，OpenCL に注目した．OpenCL はハードウェアを抽象化することで，ソースコードの互換性を実現しているため，実行環境毎に再実装する必要がないという特徴がある．その一方で抽象化によるオーバヘッドにより，特定の製品に特化した開発環境に比べ，実行速度が劣るという問題点が挙げられる．そこでこのオーバヘッドを解消するために，データ並列処理において計算ユニットが 1 度に処理する量を自動調整する事で，高速化する手法を提案する．

さらに提案手法を実現するために，プログラムを変換するプリプロセッサを実装した．しかし，この変換を適用することにより却って実効速度が低下してしまうプログラムが存在した．そこでこの速度低下を回避するために，提案手法を適用すべきでない場合を判定する機能をプリプロセッサに追加した．また本研究の妥当性を検証するために，評価として従来のプログラムとの実行時間を比較した．評価の結果，最大で

5.2 倍の速度向上が得られた．またプリプロセッサは，各ベンチマークに対して提案手法が有効であるかを正しく判定し．速度向上が得られないプログラムには提案手法の適用しないよう制御できることを確認した．

OpenCL における処理量自動調整によるデータ並列処理の高速化

目次

1 はじめに

ゲート遅延に対する配線遅延の相対的な増大から，動作周波数の向上だけではマイクロプロセッサの速度向上を見込めなくなってきた．また，集積回路の微細化に伴う消費電力や発熱量の増大から動作周波数自体の向上も困難になってきている．こうした中で，SIMD などのデータ並列性に基づく高速化手法や多数のコアを搭載したメニーコアプロセッサなどが注目されてきた．

これを受け，画像処理専用プロセッサ **GPU (Graphics Processing Unit)** や **Cell/B.E. (Cell Broadband Engine)**[1] など，SIMD 命令が利用可能なコアを多数搭載した特殊な構成を持つアーキテクチャが登場した．このようなアーキテクチャを搭載した実行環境は，1 つの実行環境の中に複数の計算ユニットと，それらを管理，制御するホストと呼ばれるコアの異なる種類のコアを搭載しているため，こうしたアーキテクチャ構成はヘテロジニアスと表現される．こうした実行環境は汎用 CPU のみに処理を割り当てる場合に比べ，ピーク処理性能が高く，コスト対性能比や電力対性能比に優れているため，HPC (High Performance Computing) 分野のみならず家庭用コンピュータでも活躍している．

このようなヘテロジニアスな実行環境を用いて，並列処理をさせる需要の高まりに応えるため，これまで様々な開発環境が提供されてきた．しかし，これらの開発環境のほとんどが特定の製品にしか対応していないため，移植性が低く，プログラマは実行環境毎に再実装しなければならない．

そこで，実行環境に依存しないソフトウェア開発環境として **OpenCL (Open Computer Language)**[2] が提案されている．OpenCL はハードウェアを抽象化することにより，ソースレベルの互換性を実現しているため，実行環境に依存することのないプログラムが記述可能になる．これにより，プログラムの再利用性と開発効率を改善する事が可能になり，プログラマにかかる再実装の負担を軽減できる．しかし OpenCL はハードウェア抽象化の弊害として，特定の製品向け開発環境に比べて，実行の際に生じるオーバヘッドが大きくなってしまうという問題がある．

そこで本論文では，OpenCL の実行方式の中でも特にデータ並列処理について着目し，このようなオーバヘッドを削減するために，計算ユニットが 1 度に処理する量を自動調整する手法を提案する．以下，2 章では本研究で扱う OpenCL の基本構成について概説し，3 章ではヘテロジニアスな実行環境についての関連研究について述べる．4 章では OpenCL の実行方式の問題点とその解決方法について述べ，5 章では提案手法

を実現するための実装方法について述べる．次に 6 章では，評価として従来の OpenCL と提案手法の実行時間を比較し，最後に 7 章で本論文全体をまとめる．

2　研究背景

本章では，本研究で取り扱う並列化フレームワーク OpenCL について，その基本構成を概説する．

2.1　OpenCL のプラットフォームモデル

OpenCL は，GPU やマルチコア CPU などの計算ユニットを用いて並列処理を行うための，標準化されたフレームワークである．OpenCL ライブラリを使用して実装したプログラムは，デバイスや OS などのプラットフォームが変化した場合でも，ソースコードレベルでの互換性が保持されるという特徴がある．

ソースコードの互換性を実現するために，OpenCL は OpenCL プラットフォームモデルと呼ばれる抽象化したハードウェアを規定している．この OpenCL プラットフォームは，1 つのホストおよびそれに接続された 1 つ以上の OpenCL デバイスから構成される．OpenCL デバイスはヘテロジニアスなアーキテクチャ構成における計算ユニットに相当し，ホストは計算ユニットを管理，制御するためのコアに相当する．OpenCL のプログラムを作成する場合，プログラマは OpenCL デバイス上で実行される**カーネル**と呼ばれるプログラムと，ホスト上で実行される**ホストプログラム**を記述する．ただし，ホストおよび OpenCL デバイスという分類方法は論理的なものであるため，実際には，ホストに相当するハードウェアと OpenCL デバイスに相当するハードウェアが同じである可能性がある．

さらに OpenCL は，OpenCL C と呼ばれるプログラミング言語と OpenCL ランタイム API 関数，OpenCL コンパイラで構成される．OpenCL C は C99 のサブセットを拡張したもので，ベクタ型や画像処理，メモリ同期などをサポートしており，カーネルは OpenCL C で記述する．ホストは API 関数をホストプログラム内で呼び出すことにより，OpenCL プラットフォームの構築や管理，コンテキストの作成や OpenCL デバイスへの処理の割り当てが可能になる．ここでコンテキストとは，OpenCL デバイスに対するカーネルの実行環境を定義したものである．OpenCL デバイスを扱う場合，ホストは OpenCL デバイス毎にコンテキストを定義し，コンテキストに対してカーネルオブジェクト，メモリオブジェクト，プログラムオブジェクトなど，カーネルの実行に必要なオブジェクトを作成する．一方で OpenCL デバイスは，ホストプログラムを

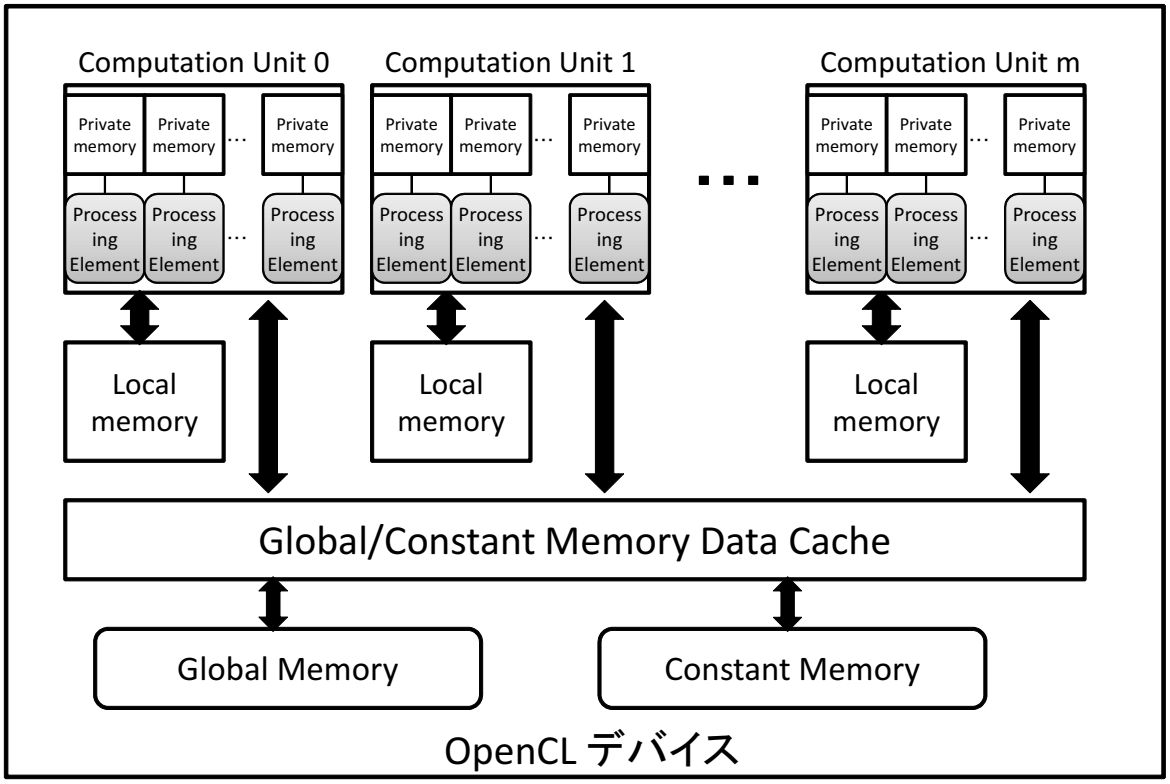

図 1: OpenCL デバイスの構成

逐次的に処理するホストと異なり，豊富な計算ユニットを用いて並列にカーネルを実行可能なため，OpenCL において計算の大部分は OpenCL デバイスが行う．OpenCL デバイスについては次の 2.2 節で詳細に説明する．

2.2 OpenCL デバイス

本節では，OpenCL デバイスの基本構成について概説する．

2.2.1 構成

図 1 に OpenCL デバイスの構成を示す．図のように，OpenCL デバイスは 1 つ以上の計算ユニット **Computation Unit (CU)** から構成されており，CU は 1 つ以上の処理エレメント **Processing Element (PE)** から構成されている．PE は仮想的なスカラプロセッサであり，各 PE は並列に動作することが可能である．また PE は SIMD ユニット，もしくは SPMD (Single Process Multiple Data) ユニットとして機能することが可能である．

PE 上で実行されるカーネルは，以下の 4 種類のメモリにアクセス可能である．

Global Memory

表 1: メモリ領域に対する割り当て方式とメモリアクセス

		Global	Constant	Local	Private
ホスト	割り当て方式	動的	動的	動的	不可
	メモリアクセス	読み書き可	読み書き可	アクセス不可	アクセス不可
カーネル	割り当て方式	不可	静的	静的	静的
	メモリアクセス	読み書き可	読み出し専用	読み書き可	読み書き可

すべての PE が読み書きできるメモリ領域である．

Constant Memory

カーネルの実行中に定数を保持するための領域である．ホストプログラムはメモリオブジェクトをアロケート，初期化した後，Constant Memory に配置する．

Local Memory

それぞれの CU に対するローカルなメモリ領域であり，CU 内の PE で共有される．ローカルメモリは計算ユニット上の実メモリを使って実装されるが，場合によっては Global Memory の一部が Local Memory として使用される場合もある．

Private Memory

PE に対してプライベートなメモリ領域であり，他の PE からは参照できない．

図 1 の Global/Constant Memory Data Cache は，PE が Global Memory と Constant Memory にアクセスする際，キャッシュとして働く．各メモリ領域に対するホストもしくはカーネルからのデータの割り当て方式およびメモリアクセスを**表 1** で表す．表より，ホストは Local Memory や Private Memory などの OpenCL デバイス内の深い階層のメモリにはアクセスできないことがわかる．こうしたメモリ領域にデータを配置するためには，プログラマは`__global`，`__constant`，`__local`，`__private` などの識別子をカーネル中に明示的に挿入する必要がある．なお OpenCL デバイスは特に GPU を意識した構成であるが， OpenCL はより汎用的なハードウェアアーキテクチャを想定しているため，GPU 特有のテクスチャメモリなどは規定されていない．

2.2.2 インデクス空間

カーネルが実行される際，ホストプログラムによって n 次元 ($n = 1 \sim 3$) の抽象的なインデクス空間が定義される．インデクス空間の各座標には次元を表すための n 組の整数値と，**ワークアイテム**と呼ばれるカーネルの実行インスタンスが割り当てられている．さらに，ワークアイテムをまとめたものを**ワークグループ**と呼び，ワークグ

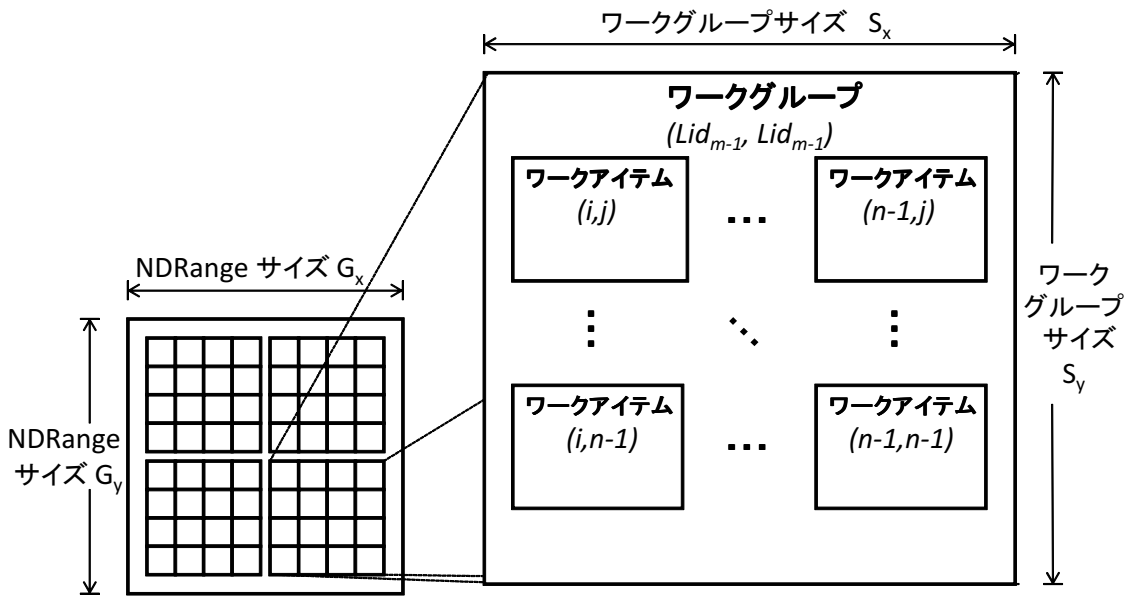

図 2: 2 次元のインデクス空間

ループをまとめたインデクス空間全体のことを NDRange と呼ぶ．ここで，インデクス空間の例を**図 2** に示す．図 2 は 2 次元のインデクス空間を表しているため，ワークグループとワークアイテムには 2 組の整数値が割り当てられている．図においてワークアイテムに割り当てられている整数値をグローバル ID と呼び，ワークグループに割り当てられる整数値をローカル ID と呼ぶ．図 2 における S_x, S_y はワークグループ内のワークアイテム数である**ワークグループサイズ**を表しており，G_x, G_y は NDRange 内のワークグループ数である NDRange サイズを表している．

このようなインデクス空間を用いたプログラミングは，一般的に逐次処理におけるループを使った処理を並列化する際に利用される．このことを**図 3** を用いて説明する．図の 3 行目の関数は，座標 (i,j) を引数として処理対象を変更している．この例では，変数 i と j の値を変更することで，n × m 通りの座標を計算しながら 3 行目の関数の処理対象を変更しているが，OpenCL では予め実行前に n × m の座標を持ち，それぞれの座標に i，j に相当する値が与えられているインデクス空間を用意しておく．これにより OpenCL では，この例のように対象の座標を実行時に計算する必要がなく，実行の際にはそれぞれの座標に対応したワークアイテムを非同期に呼び出すことで，並列処理を実現している．

```
1 for( i = 0; i< n; i++){
2    for(j = 0; j< m; j++ ){
3       workitem(i , j );  //ワークアイテムの実行
4    }
5 }
```

図 3: 2 次元の場合，n × m のワークアイテムを実行する例

2.3 実行モデル

OpenCL はデータ並列処理とタスク並列処理をサポートしており，2 つのモデルを組み合わせて利用することができる．データ並列処理とは，計算ユニットが受け持つデータの要素に対して，同一のカーネルを適用する実行方式の事である．OpenCL では 2.2.2 項で述べたインデクス空間を用いて，データの各要素とワークアイテムを 1 対 1 対応させることでデータ並列処理を実現している．これにより，PE はワークアイテム毎に処理対象を変更させて，カーネルを実行することが可能になる．また OpenCL は 2.2.2 項で述べたように，NDRange，ワークグループ，ワークアイテムによる 3 階層のインデクス空間を定義している．そのためホストは，NDRange を OpenCL デバイスへ，ワークグループを CU へ，ワークアイテムを PE へ割り当てることで，3 階層のデータ並列処理を可能にしている．こうした階層的なデータ並列処理を行うためには，プログラマはインデクス空間の次元数，ワークアイテムの総数，ワークグループサイズを指定することでインデクス空間を定義しなければならない．

このようなデータ並列処理を実行する場合，1 つのワークグループ内で，ワークアイテムの同期をとるには，ワークグループバリアと呼ばれる機能を使用する．これはカーネルのソースコードにおいて，OpenCL C の組込み関数であるバリア関数を呼び出すことで実現できる．ワークグループバリアにより，あるワークアイテムにおいてバリア関数が呼び出されると，同じワークグループに属するすべてのワークアイテムがバリア関数を呼び出すまで，ワークアイテムは同期待ち状態になる．そのため，ワークグループバリアを使用することで Global Memory と Local Memory に対する操作の順序を保証することができる．

一方タスク並列処理とは，カーネルのインスタンスをインデクス空間と無関係に動作させる処理方式である．そのためデータ並列処理と異なり，プログラマはインデクス空間を意識する必要が無く，ワークアイテムの総数などを指定する必要はない．

本研究では，算術科学演算に向いているデータ並列処理に着目する．そこで OpenCL

```
1  __kernel void
2  sample(__global const float *in1,
3         __global const float *in2,
4         __global float *out)
5  {
6     int index = get_global_id (0);
7     out[index] = in1[index] + in2[index];
8  }
```

図 4: OpenCL におけるデータ並列処理の例

でデータ並列処理をする場合の例を，**図 4** に示すカーネルを用いて説明する．図 4 は，1 次元のインデクス空間を用いてデータ並列処理をする場合のカーネルの例である．まず最初の`__kernel` という修飾子は，この関数がカーネル関数であることを表す (1 行目)．カーネル関数はワークアイテム内で動作する関数であり，C や C++などにおける main 文に相当する．カーネル関数を実行する際には，すべてのワークアイテムにグローバル ID が割り当てられる．例えば，ワークアイテム数を 100 と指定した場合，カーネル関数を実行する際に 100 個のワークアイテムが作成される．この時 PE は，ワークアイテム毎に割り当てられたグローバル ID を取得することで，配列の 0 から 99 番目の要素に対する演算を，並列に実行することができる．図 4 の例の場合，i 番目のワークアイテムを実行する PE は配列の i 番目の要素にアクセスし，計算する．カーネル内でワークアイテムに割り当てられているグローバル ID を取得する場合，OpenCL C の組込み関数である`get_global_id` 関数を使用する (6 行目)．`get_global_id` 関数の引数にはどの次元のインデクスを取得するかを指定する．図 4 では，1 次元のインデクス空間を扱っているので，引数には 0 が与えられている．また引数の型である`__global` は，プログラマがこの引数を Global Memory 空間に配置するようにコンパイラに指示するための修飾子である (2 ～ 4 行目)．Global Memory 空間に配置されたデータは，すべてのワークアイテムから参照することが可能である．

3 関連研究

本章では，ヘテロジニアスな実行環境に関する研究について述べる．まず，ヘテロジニアスマルチコアプロセッサの先駆けとして登場した Cell/B.E. は，SONY，東芝，IBM の 3 社により共同開発された，高い処理性能を目指したマルチコア SIMD プロセッサである．Cell/B.E. は，1 つの汎用コア **PPE (PowerPC Processor Element)** と

8 つの演算コア **SPE (Synergistic Processor Element)** を 1 チップ上に集約している．SPE はメインメモリに直接アクセスできないため，SPE がメインメモリ内のデータを必要とする場合は **DMA (Direct Memory Access) 転送**と呼ばれるデータ転送を行い，メインメモリ内のデータを SPE が持つ Local Store に転送しなければならない．DMA 転送とは，CPU のレジスタなどを介すること無く，メモリ間で直接データを転送する方式のことであり，OpenCL においても DMA 転送はサポートされている．しかしこのような DMA 転送に関する明示的な記述をしたり，PPE と SPE への処理の振り分けなどを行うには，プログラマは Cell/B.E. に対する十分な知識が必要となる．そこで，Cell/B.E. を用いたプログラミングを容易にするための開発環境に関する研究が行われており，既存のコンピュータビジョン向けライブラリである OpenCV の API を Cell/B.E. 向けに最適化した CVCell[3] や，DMA 転送などの複雑な記述を抽象化した Cell Tool Kit[4] などが存在する．

またホストに CPU，計算ユニットに GPU を想定した開発環境も提供されている．中でも NVIDIA 社が提供している **CUDA (Compute Unified Device Architecture)** [5] は，GPU プログラミングの複雑さを緩和し，GPU を汎用計算に利用する GPGPU (global purpose GPU) 技術の発展に大いに貢献した．現在，GPGPU の研究には開発環境として CUDA が最も用いられている．OpenCL はハードウェアの抽象化や実行モデルなど，CUDA から多くのモデルを参考にしている．他にも GPU を扱う開発環境として，AMD 社の製品に特化した開発環境である ATI CTM[6] などが存在する．

以上に述べたように，CUDA の登場により GPU プログラミングは容易になった．しかし，プログラマはスレッド階層やデータを配置するメモリ領域などを指定する必要があるため，汎用 CPU を扱う場合と比較すると，依然としてプログラミングが複雑である．そこで，hiCUDA[7] や PGI コンパイラ [8] など，ソースコードにコンパイラへの指示文である pragma を挿入する事で，逐次プログラムを GPU 向けプログラムに自動変換する研究が行われている．しかし，プログラマは新たに hiCDA，もしくは PGI コンパイラ向けの pragma を理解しなければならない．さらに，プログラマは pragma を用いてスレッド階層やメモリ領域などを指定する必要があるため，これらの開発環境を利用するには依然として GPU の知識が必要になる．そこで，汎用 CPU に対して自動並列化を行う OpenMP の pragma を GPU に対応させる OpenMPC[9][10] などの研究が進められている．この研究では，プログラマは OpenMP と同じ構文で並列化箇所を指定できるようになるため，GPU に対する十分な知識が無くとも並列処理をさせることができる．

また，OpenCL に関する先行研究もいくつか存在する．荒井ら [11] は，CUDA と OpenCL の実行時間を比較しており，さらに OpenCL の問題点と行うべき最適化についても考察している．一方で Lee ら [12] は，OpenCL ランタイムに対してさまざまな改良をし，コマンドスケジューラの実装およびデータの先読み等をさせることで OpenCL の高速化を実現している．しかし，OpenCL は提供され始めてから日が浅いため，これに関する研究はまだ少ない．

4　処理量自動調整

本章では，OpenCL におけるデータ並列処理の問題点を指摘し，解決方法を提案する．

4.1　OpenCL の問題点

ハードウェアの抽象化によるオーバヘッドが原因で，データ並列処理を行う際には，以下の 3 つの問題が存在する．まず 1 つ目の問題点は，ワークアイテムの切り替えオーバヘッドである．PE が，あるワークアイテムの演算を終了して，ワークアイテムを切り替える際，コンテキストスイッチが発生する．さらにその後，OpenCL デバイスのメモリ領域からホストのメインメモリに，ワークアイテムの実行結果を DMA 転送を用いて書き戻す．これらに加えて，ワークグループバリアによる同期が必要な場合がある．同期待ちの状態では，ワークアイテムは何も実行することが出来ないため，同期待ちもオーバヘッドとなる．こうしたオーバヘッドがワークアイテムの切り替え時に発生してしまう。

2 つ目の問題点は，負荷分散を細粒度化するためにワークアイテムを多く作成した場合，全体の処理に対してオーバヘッドが占める割合が増加してしまうことである．一般的に OpenCL では，PE はワークアイテムごとに割り当てられたデータの要素しか処理しない．そのため大きなデータサイズに対して，処理内容が単純なプログラムを実行する場合，ワークアイテム 1 つの処理は短時間で終了し，なおかつワークアイテムの切り替えを膨大に行う必要がある．その結果，ワークアイテムの切り替えによって発生するオーバヘッドが全体の処理の中で大きな割合を占めてしまう．また，OpenCL デバイスは特に GPU を意識したアーキテクチャ構成のため，NDRange，ワークグループ，ワークアイテムを用いた 3 階層のデータ並列処理をしているが，この方式は非常に多くの計算ユニットを持つ GPU に適している反面，マルチコア CPU や Cell/B.E. など計算ユニットが少ないプロセッサには向かない．そのため，マルチコア CPU や Cell/B.E. などを OpenCL デバイスとして扱う場合，ワークアイテムを多く作成する

```
1  int globalIdx = get_global_id(0);
2  int globalIdy = get_global_id(1);
3
4  int tmp = size * size;
5
6  for(int i =0; i<size ;i++){
7    array[globalIdy * size + globalIdx] = tmp;
8  }
```

図 5: プログラム例

ことが，却ってオーバヘッドを増加させてしまい，実行速度を悪化させる要因となる可能性がある．

3 つ目の問題点は，ワークアイテムの処理はそれぞれ独立しているため，ワークアイテムに対してローカルな変数の宣言，グローバル ID の取得，制御文における条件判定などは，ワークアイテムごとに行わなければならないことである．変数宣言や条件判定などはオーバヘッドとなるため，これらの処理が全体の処理の中で大きな割合を占めてしまうと，実行速度の低下をもたらしてしまう．この問題を，**図 5** に示すプログラム例を用いて説明する．このプログラムをワークアイテムの総数を 100 と指定して実行した際，ワークアイテム全体で合計すると，プログラム内の変数宣言 (4 行目) は 100 回実行され，for 文における条件判定 (6 行目) は 100 × size 回行われる．そのため，逐次処理では 1 つ生成されるだけで良い変数も，OpenCL ではワークアイテムの総数だけ生成する必要がある．

以上に述べたように，OpenCL のデータ並列処理ではワークアイテムを多く作成した場合，それに伴い様々なオーバヘッドが発生し，実行速度を低下させるという問題がある．そのため，PE と CU が処理する量を調整する仕組みが必要である．

4.2 データ並列処理における処理量自動調整

本節では，OpenCL の問題点を解決するための手法を提案する．本提案ではワークアイテムの統合とワークグループサイズの変更の 2 つの手法により，データ並列処理における PE と CU の処理量を自動的に調整することで OpenCL の高速化を図る．

4.2.1 ワークアイテムの統合

従来の OpenCL では一般的に，ワークアイテムごとに割り当てられたデータの要素しか処理されないため，OpenCL デバイスの処理全体で見ると，ワークアイテムの総

数だけワークアイテムの切り替えが発生してしまう．そこでワークアイテムを統合することで，1つのワークアイテム内で複数の要素を処理可能にする手法を提案する．これによりワークアイテムの総数が削減されるため，ワークアイテムの切り替え回数が減り，その際に生じるオーバヘッドを削減することができる．

さらにこの手法を適用することで，ワークアイテム内の演算の処理効率の向上が図れる．例えばワークアイテム統合後のプログラムにおいて，要素間でデータ依存が存在しない場合，各要素に対する演算を，順序を変更して実行することが可能になる．そのためある要素に対する演算の実行後に，処理対象を変更させた，別の要素に対する演算を実行することができる．このように実行順序を変更することで，グローバルIDの取得，変数の宣言，for文などにおける条件判定など，冗長な処理を削減することが可能になり，処理効率を向上させることができる．例えば図5の例に対しワークアイテムを統合した場合，4行目で宣言されているローカル変数tmpは，そのワークアイテム内で1度生成されれば良い変数であるため，他の要素に対する演算にも利用することができる．本論文では，今後このようなワークアイテム内で1度生成されれば良いローカル変数のことを，**生成数が削減され得る変数**と呼ぶ．さらに6行目以降に存在する制御文であるが，条件判定に使われている変数は，すべてのワークアイテムで一定の値を持つため，各要素毎に条件判定する必要はない．そのため演算順序を変更することで，そのワークアイテムが受け持つ全ての要素に対するループ内の処理を，1つのループを用いて1度に実行することが可能になる．

しかしワークアイテム内で複数の要素を扱うために，1度取得したインデクスの値を書き換える必要がある．そのため，インデクスの計算にかかるコストが発生してしまうという問題がある．さらに従来手法に比べ，1つのワークアイテムの実行時間が長くなるため，1度の同期待ちにかかる時間が長くなる可能性があるといった問題もある．

4.2.2　ワークグループサイズの変更

ワークアイテムの統合により，PEあたりの処理量が調整されるが，その上位階層であるワークグループとCUにおいても処理量を調整する．ワークグループの総数はワークグループサイズを指定することにより決定されるため，ワークグループサイズを変更することで，CUの処理量を調整する．

ここで，ワークグループサイズに与える値は，ワークグループバリアによる同期が必要か否かで値を変更する．まず同期が必要ないプログラムの場合，ワークアイテムの実行は，他のワークアイテムと干渉せず，完全に並列に実行することが可能である．

また，ワークグループを切り替える際にもオーバヘッドが発生するため，ワークグループの総数は少ない方が良い．これらのことからワークグループサイズには大きな値を与えた方が良いが，過度に大きな値を与えた場合，処理が割り当てられない計算ユニットを生む可能性があるため，適度な値を設定する必要がある．

一方で同期が必要なプログラムの場合，ワークグループサイズを大きく設定すると，ワークアイテムの同期待ち時間が増加してしまう可能性がある．そこで，同期が必要なプログラムに対してはワークグループサイズを小さく設定することで，同期待ち時間を低減する．

5　プリプロセッサの実装

本章では，提案手法を用いてデータ並列処理の処理量を調整するための，実装方法について説明する．

5.1　コンパイルフロー

通常 OpenCL では特定の製品向けに提供されているホスト用コンパイラとカーネル用コンパイラを使用してコンパイルする．ホストプログラムをコンパイルする際には，API 関数を利用するために OpenCL のランタイムライブラリをリンクする．一方で，カーネル用コンパイラは OpenCL C で記述されたカーネルをコンパイルするが，C 言語に変換したプログラムを C コンパイラによりコンパイルする方式がある一方で，OpenCL C のままコンパイルする方式も存在するなど，カーネルのコンパイル方式は様々である．

提案手法は，コンパイル前に，処理量を調整するように変換するプリプロセッサを実装することで実現した．なお，プリプロセッサは flex/bison を用いて実装した．これによるコンパイルフローの全体像を**図 6** に示す．

このプリプロセッサは (1) カーネルのフロー解析，(2) ホストプログラムの解析と変換，(3) カーネルの変換の 3 つの工程で構成される．まず (1) カーネルのフロー解析では，処理量調整により高速化が得られるかどうかを判定する．次に (2) ホストプログラムの解析と変換では，データ並列処理を行う際のワークアイテム数，ワークグループサイズを変更することで処理量を調整する．最後に (3) カーネルの変換では，単一ワークアイテム内で複数の要素を処理できるようにカーネルを変換する．以下，これらの各工程について，詳細に説明する．

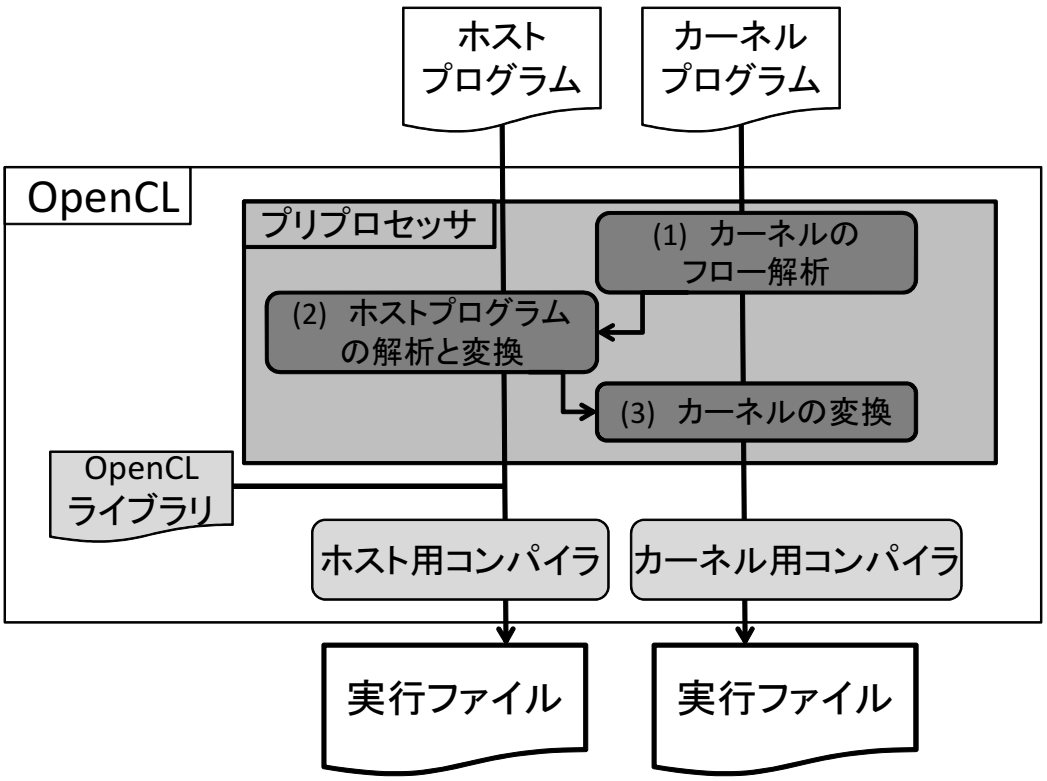

図 6: コンパイルフロー

5.2 カーネルのフロー解析

ワークアイテムの統合により，処理結果の正当性が保証されなくなってしまうプログラムや，却って速度低下を招いてしまうプログラムも存在するため，カーネル内の処理内容を解析することで，カーネルを変換すべきかどうか判定する．処理結果の正当性が保証されないプログラムの事例として，データ依存が存在する場合が挙げられ，速度低下を招いてしまうプログラムの事例として，削減可能な処理が存在しない場合が挙げられる．

データ依存が存在するプログラムに対して，提案手法を適用した場合の問題点を**図 7** を用いて示す．この例の場合，配列 A，B はどちらもカーネルから 2 度以上のアクセスがあり (3，5 行目)，どちらも 1 度の書き込みが存在する (A は 3 行目，B は 5 行目) ため，データ依存が存在する．この例に対し，提案手法によって変換したプログラムを**図 8** に示す．図 8 の 4 行目と 7 行目は，順序を変更させた，他の要素に対する演算である．これらの例を size を 2 として実行した場合の演算順序と，演算ごとの実行結果を，それぞれ**図 9** と**図 10** で示す．図の左側は演算の順序を表しており，右側は左側の演算を実行した後の，配列の値を表している．例えば，図 9 の 2 行目では，A[1] に

```
1 int index = get_global_id(0);
2 for(i = 0 ; i < size ; i++){
3   A[i * size + index] = B[i * size + index];
4   for(j = 0 ; j < size ; j++){
5     B[index * size + j] = 2 * A[index * size + j];
6   }
7 }
```

図 7: データ依存が存在するプログラム

```
1 int index = get_global_id(0);
2 for(i = 0 ; i < size ; i++){
3   A[i * size + index] = B[i * size + index];
4   A[i * size + (index + 1)] = B[i * size + (index + 1)];
5   for(j = 0 ; j < size ; j++){
6     B[index * size + j] = 2 * A[index * size + j];
7     B[(index + 1) * size + j] = 2 * A[(index + 1) * size + j];
8   }
9 }
```

図 8: 図 8 の変換後のプログラム

```
1 A[0] = B[0];     :A[0]=1,B[0]=1
2 A[1] = B[1];     :A[1]=1,B[1]=1
3 B[0] = 2*A[0]; :A[0]=1,B[0]=2
4 B[2] = 2*A[2]; :A[2]=1,B[2]=2
5 B[1] = 2*A[1]; :A[1]=1,B[1]=2
6 B[3] = 2*A[3]; :A[3]=0,B[3]=0
```

図 9: 図 7 の処理内容 (前半部のみ抜粋)

```
1 A[0] = B[0];     :A[0]=1,B[0]=1
2 B[0] = 2*A[0]; :A[0]=1,B[0]=2
3 B[1] = 2*A[1]; :A[1]=1,B[1]=2
4 A[2] = B[2];     :A[2]=1,B[2]=1
5 B[0] = 2*A[0]; :A[0]=1,B[0]=2
6 B[1] = 2*A[1]; :A[1]=1,B[1]=2
```

図 10: 図 8 の処理内容 (前半部のみ抜粋)

対する最初の処理は書き込みであるが，図 10 では，A[1] に対する最初の処理は 3 行目の読み出しである．さらに，図 9 の 4 行目では，A[2] に対する最初の処理として読み出しを行っているが，図 10 の 4 行目では，A[2] に対して書き込みを行っている．このように，データ依存が存在するプログラムに対して提案手法を適用した場合，本来は値が書き込まれた後に読み出しを行うべき演算に対して，値が書き込まれる前に読み出ししてしまう可能性があり，処理結果の正当性を保証できない．

```
1 int i = get_global_id (0);
2 int j = get_global_id (1);
3
4 int tmp = i * j;
5 array[i * size + j] = tmp;
```

図 11: 変換前

```
1 int i = get_global_id (0);
2 int j = get_global_id (1);
3 j = j << 1;
4
5 int tmp = i * j;
6 int __tmp = i * (j + 1);
7 array[i * size + j] = tmp;
8 array[i * size +(j+1)] = __tmp;
```

図 12: 変換後 (一部省略)

また，提案手法により削減可能な処理が存在しない場合，演算に要する時間に対して，インデクスの計算コストの比率が大きくなるため，速度向上が得られない可能性がある．このことを**図 11** のプログラム例と，それの変換例である**図 12** を用いて説明する．図 12 の 3，6，8 行目は，複数の要素を扱うために，加算によってグローバル ID を変更しており，また 6 行目では変数を複製しているが，これらの動作については 5.4 節で詳しく説明する．この例の場合では，提案手法によって削減できる，ローカル変数の生成や条件判定がプログラム中に存在しないため，図 12 の 5，6 行目と 9，10 行目では，変換前の処理 (図 11，4，5 行目) を順序を変更して実行している．このような削減可能な処理が存在しないプログラムの場合，ワークアイテムを統合したとしても，従来のプログラムにおける複数のワークアイテムの処理を，単に 1 つのワークアイテム内で処理しているだけである．そのためワークアイテム全体で見ると，従来のプログラムと統合後のプログラムでは処理内容は変化していない．それだけでなくインデクスを計算する必要があるため，却って処理量が増加してしまう可能性がある．

これらの問題を回避するため，カーネルの解析部では，まずカーネル内の配列に対してデータ依存の有無を解析し，提案手法を適用しても処理結果の正当性が保証されるかを判定する．データ依存が存在しないならば，次に削減可能な処理が存在するか否かを判定する．データ依存が存在する場合，またはデータ依存は存在しないが，削減可能な処理が存在しない場合は，処理量を調整しない．以下，データ依存や削減可能な処理の解析方法について，詳細に説明する．

5.2.1 データ依存の解析

データ依存の解析では，まず `get_global_id` 関数で取得したインデクスを添字として扱う配列を解析対象として選択する．その後，解析対象に対する読み出し回数，書き込み回数を記憶していく．この時，解析対象が計算の右辺に現れる場合は読み出し，

```
1 A[i] += 10;
2 A[i] = A[i] + 10;
```

図 13: データ依存が存在しない処理

計算の左辺に現れる場合は書き込みと判定する．ファイルの終端まで解析した結果，解析対象に対する読み出し回数と書き込み回数の合計が 2 以上かつ，書き込みが 1 度でも行われている場合，解析対象の配列には依存関係が存在すると判定する．ただしこの判定方式では，配列の添字を意識していないため，本来ならデータ依存が存在しないデータに対しても依存関係が存在すると判定してしまう可能性がある．

そこでデータ依存が存在すると判定されても，以下の条件を満たすデータは並列処理が可能であるため，例外として扱う．

1. 復号代入文である
2. 同一のメモリに対する処理である

この例外の例を**図 13** に示す．図 13 の 1，2 行目は，どちらも配列に対して，書き込みおよび読み出しをしている演算であるため，プリプロセッサはデータ依存が存在すると判定してしまう．しかしどちらの演算も同一メモリに対する処理であるため，並列化が可能である．そこで，カーネルの演算内容が上記の演算のみならば，データ依存が存在しないと判定する．

5.2.2 削減可能な処理を解析

この解析では，カーネル内で 2 つの削減可能な処理を探索する．1 つ目の削減可能な処理は，4.2.1 項で定義した生成数が削減され得る変数である．2 つ目の削減可能な処理は，条件式に生成数が削減され得る変数が含まれる制御文である．このような制御文は，ワークアイテム統合後でも条件判定回数は変化しないため，削減可能な処理として扱える．これらを検出した結果，削減可能な処理が 1 つでも存在した場合，高速化が可能と判定する．

高速化が可能と判定した後，ワークグループバリアによる同期をとるかどうかを判定する．同期の有無はプリプロセッサが用意する sync_flag というフラグを用いて記憶する．プログラム内にバリア関数が存在する場合，同期をとると判定し，sync_flag を立て，同期をとらない場合はフラグを下ろす．なお，sync_flag は 5.4 節で述べるワークグループサイズの変更の際に用いる．

ただし本節で述べた解析方式は，単純かつ曖昧なものであるため，厳密に速度向上

```
1 cl_uint  work_dim = 2;                                  //インデクス空間の次元数
2 size_t  global_work_size [] = { Width , Height }; //ワークアイテム数
3 size_t  local_work_size [] = { 8 , 8 };                 //ワークグループサイズ
4 status = clEnqueueNDRangeKernel(queue , kernel,
5                                 work_dim,NULL,
6                                 global_work_size ,
7                                 local_work_size ,
8                                 0,0,
9                                 NULL,NULL);
```

図 14: clEnqueueNDRangeKernel の使用例

が得られるかどうかを判定できない．今後プロファイラ等を用いて精密に解析する予定である．

5.3 ホストプログラムの解析と変換

ホストプログラムを解析および変換することで，ワークアイテムの総数とワークグループサイズを変更し，PE と CU の処理量を調整する．この工程は (A) 次元数，ワークアイテム数，ワークグループサイズの取得，(B) 統合数の算出とワークアイテム数の書き換え，(C) ワークグループサイズの書き換えの 3 つの工程で構成される．

まず，(A) の工程について述べる．通常インデクス空間を用いて OpenCL デバイスに算術演算をさせる場合，ホストプログラムにおいて，clEnqueueNDRangeKernel という API 関数を使用する．clEnqueueNDRangeKernel の使用例を**図 14** に示す．プログラマは clEnqueueNDRangeKernel の引数にワークアイテムの総数，ワークグループサイズ，扱うインデクス空間の次元数などを指定することで，インデクス空間およびワークアイテムを作成する．図 14 では，第 1 引数にはコマンドキュー，第 2 引数にはカーネルのオブジェクトが与えられている (4 行目)．さらに第 3 引数では，先に宣言しておいた次元数 (5 行目)，第 5 引数にはワークアイテム数 (6 行目)，第 6 引数にはワークグループサイズが与えられる (7 行目)．そこで，まずホストプログラムを解析し，この API 関数の引数から次元数，ワークアイテム数，グローバルサイズを取得する．通常直接入力値が与えられることは少なく，配列や変数等に格納されているため，具体的な入力値が見つかるまで繰り返しホストプログラムを解析する．また，取得したインデクス空間の次元数はカーネルを変換する際にも用いる．

次に，取得したワークアイテム数を基に (B) の工程を行う．なお，統合数とは 1 つの

ワークアイテム内で処理する要素数を指す．カーネルの処理内容から最適な統合数を静的に判定するのは困難なため，統合数は 16 を基本値とする．これは予備評価から，統合数が 16 までは速度向上が得られると判断したためである．ただしワークアイテム数を統合数で割り切れない場合，全ての要素が参照されないために，正常に処理できない可能性がある．そのような場合には，ワークアイテム数が割り切れるまで，統合数を減らしていく．なお，統合数もカーネルを変換する際に用いる．その後，インデクス空間の中で最も高次元を表しているワークアイテム数を先ほど取得した次元数を基に書き換える．例えば 3 次元のインデクス空間を扱うプログラムの場合は，ワークアイテム数が格納されている配列の中で 3 番目の値を，2 次元の場合なら 2 番目の値を書き換える．図 14 では 2 次元のインデクス空間を扱っているため，2 行目の Height が書き換えの対象になる．これらの動作の終了後，対象のワークアイテム数をワークアイテム数 ÷ 統合数の値で書き換える．

最後に (C) の工程であるが，この時与えるワークグループサイズは同期の有無で決定する．まず，5.2 節で用意した sync_flag が下りている場合は同期が必要ないため，大きな値を設定する．予備評価の結果，ワークグループサイズの値には 32 が適切であると判断したため，基本値を 32 とし，1 次元の場合は 32，2 次元の場合は 32×32，3 次元の場合は $32 \times 32 \times 32$ とする．反対に sync_flag が立っている場合は同期が必要であるため，同期による待ち時間の増加を回避するためにワークグループサイズは大きくできない．したがって，この場合ワークグループサイズの基本値は 4 とする．ただし，ワークグループサイズは以下の条件式を満たす必要がある．

$$\text{ワークアイテム数} = \text{統合数} \times \text{ワークグループサイズ} \times n$$

ワークグループサイズがこの制約に違反する場合，値を小さくすることで調整する．ワークグループサイズを算出後，その値になるようワークグループサイズを書き換える．

5.4 カーネルの変換

最後に，1 つのワークアイテム内で複数の要素を扱えるようにカーネルを変換する．変換にはいくつかの変換規則を用いる．この工程では，カーネル関数内の処理を，`get_global_id` 関数などで自身のグローバル ID を取得するインデクスの取得部と，実際に演算する演算本体部に分割して異なる変換規則を適用する．

まず，カーネルを変換する際の変換規則について説明する．インデクスの取得部で

は，5.3 節で取得した次元数を基に書き換え対象のインデクスを検出する．例えば 2 次元のインデクス空間を扱う場合では，get_global_id(1) で取得したインデクス，3 次元の場合は，get_global_id(2) で取得したインデクスが書き換え対象である．この時，書き換え対象のインデクスはこれからの変換に必要であるため，プリプロセッサはこのインデクスの名前を記憶しておく．次に 1 つのワークアイテムが受け持つデータの要素数を変化させるため，対象のインデクスを統合数で乗算することにより，そのワークアイテムが受け持つ要素の区間の起点を変更する．さらに 1 度取得したグローバル ID を，順に計算して値を変化させることで，グローバル ID の取得回数を削減できる．例えば統合数が 16 の場合，取得したインデクスを 16 倍することで，そのインデクスが配列の 16 の倍数の要素を指すように設定する．設定後，そのインデクスに 0 ～ 15 までの値を加算することで，そのインデクスはワークアイテム内で 16 の要素を参照することができる．

次に演算本体部を解析し，以下に示す変換規則を用いて変換する．

(規則 1) 書き換え対象のインデクスもしくは複製元の変数が右辺に存在しており，なおかつ宣言されたばかりの変数に書き込みを行う場合，その変数を複製する．

(規則 2) 右辺と左辺の両方に，書き込え対象のインデクスもしくは複製元の変数が存在している場合，グローバル ID の値と複製された変数の変数名が，それぞれ対応するようにコードを生成する．

(規則 3) 初期化処理される変数は，複製する．

(規則 4) それ以外の文は変更せず，コードを生成する．

グローバル ID の書き換えによって値が変化する変数は，削減してはならない変数であるため，そのような変数が存在する場合には規則 1 を用いて変数を複製する．さらに初期化処理される変数が存在する場合，この変数に代入される値は，グローバル ID に伴って変化する可能性があるため，予め変数を複製しておく．ただし，従来のプログラムで生成される数以上の変数が生成されることはない．

上記の変換規則による変換動作を**図 15** のプログラム例と，図 15 の変換後のプログラムである**図 16** を用いて示す．ここで，5.3 節のホストプログラムの解析過程から統合数は 4，次元数は 2 という情報を持っていると仮定する．

まずインデクスの取得部 (図 15，1，2 行目) であるが，次元数は 2 のため，書き換え対象のインデクスは globalIdy となる．globalIdy は規則 1，規則 2 の変換で必要なため，プリプロセッサが記憶しておく．この動作の終了後，globalIdy を統合数で乗算し，値を変更する．この例では統合数は 4 なので，globalIdy の値を左に 2 ビットシフトす

```
1  int globalIdx = get_global_id(0);
2  int globalIdy = get_global_id(1);
3  float sum = 0;
4
5    for(int i =0; i<widthA;i++)
6     {
7       float tempA = mA[globalIdy * widthA + i];
8       float tempB = mB[i * widthB + globalIdx];
9       sum+= tempA * tempB;
10    }
11 mC[globalIdy * widthA + globalIdx] = sum;
```

図 15: プログラム例

る (図 16，3 行目).

次に，本体処理部 (図 15，3 行目以降) の変換に移る．まず変数 sum が宣言されているが (図 15，4 行目)，これは初期化処理が施されているため，規則 3 の変換規則を適用し，3 つの変数を複製する (図 16，5 ～ 8 行目)．ここで，変数 sum を複製元の変数として記憶しておく．次に for 文の条件判定部だが (図 15，5 行目)，これは規則 4 を適用し，そのまま出力する (図 15，6 行目)．次に変数 tempA が宣言されているが (図 15，7 行目)，この変数の値は globalIdy の値の変化に伴って変化するため，規則 1 を適用し，この変数を 3 つ複製する (図 16，11 ～ 14 行目)．その後，この tempA も複製元の変数として記憶しておく．次に変数 tempB が宣言されているが (図 15，8 行目)，この変数は生成数が削減され得る変数であるため，規則 4 を適用する．次の演算 (図 15，9 行目) では，左辺と右辺に複製元の変数である sum および tempA が存在するため，規則 2 を適用し，複製された変数と処理対象の要素がそれぞれ対応するように，コードを生成する (図 16，18 ～ 21 行目)．最後に値の代入だが (図 15，11 行目)，この演算も左辺値に globalIdy，右辺値に複製した変数である sum が存在しているため，規則 2 を適用してコードを生成する (図 16，22 ～ 25 行目).

この例では図 15 の 7，9，11 行目の演算をそのワークアイテムが受け持つ複数の要素に対して実行可能になった．さらに変換前に比べて，比較回数，分岐回数と tempB の生成回数が 1/4 回に削減できており，処理速度の向上が期待できる．

```
1  int globalIdx = get_global_id(0);
2  int globalIdy = get_global_id(1);
3  globalIdy = globalIdy << 2;
4
5  float sum = 0;
6  float __sum1 = 0;
7  float __sum2 = 0;
8  float __sum3 = 0;
9
10 for(int i = 0; i < widthA ; i++){
11   float tempA = mA[globalIdy * widthA + i];
12   float __tempA1 = mA[(globalIdy+1) * widthA + i];
13   float __tempA2 = mA[(globalIdy+2) * widthA + i];
14   float __tempA3 = mA[(globalIdy+3) * widthA + i];
15
16   float tempB = mB[i * widthB + globalIdx];
17
18   sum += tempA * tempB;
19   __sum1 += __tempA1 * tempB;
20   __sum2 += __tempA2 * tempB;
21   __sum3 += __tempA3 * tempB;
22 }
23 mC[globalIdy * widthA + globalIdx] = sum;
24 mC[(globalIdy+1) * widthA + globalIdx] = __sum1;
25 mC[(globalIdy+2) * widthA + globalIdx] = __sum2;
26 mC[(globalIdy+3) * widthA + globalIdx] = __sum3;
```

図 16: 変換後のプログラム

6 評価

本提案手法の有効性を確認するため，5 章で述べたプリプロセッサを実装し，ベンチマークプログラムを用いて評価した．

6.1 評価環境

評価として，従来の OpenCL のプログラムの実行時間と，プリプロセッサが変換したプログラムの実行時間を比較した．評価環境を**表 2** に示す．評価の際に使用したプラットフォームは PLATSTATION3，開発環境には OpenCL 1.1 をサポートする OpenCL

表 2: 評価環境

プラットフォーム	PLAYSTATION3
OS	Fedora 10
計算ユニット	Cell/B.E. の SPE 6 基
コンパイラ	gcc 4.1.1, spe-gcc, ppe-gcc
最適化オプション	-O3
開発環境	OpenCL Development Kit for Linux on Power
OpenCL のバージョン	1.1

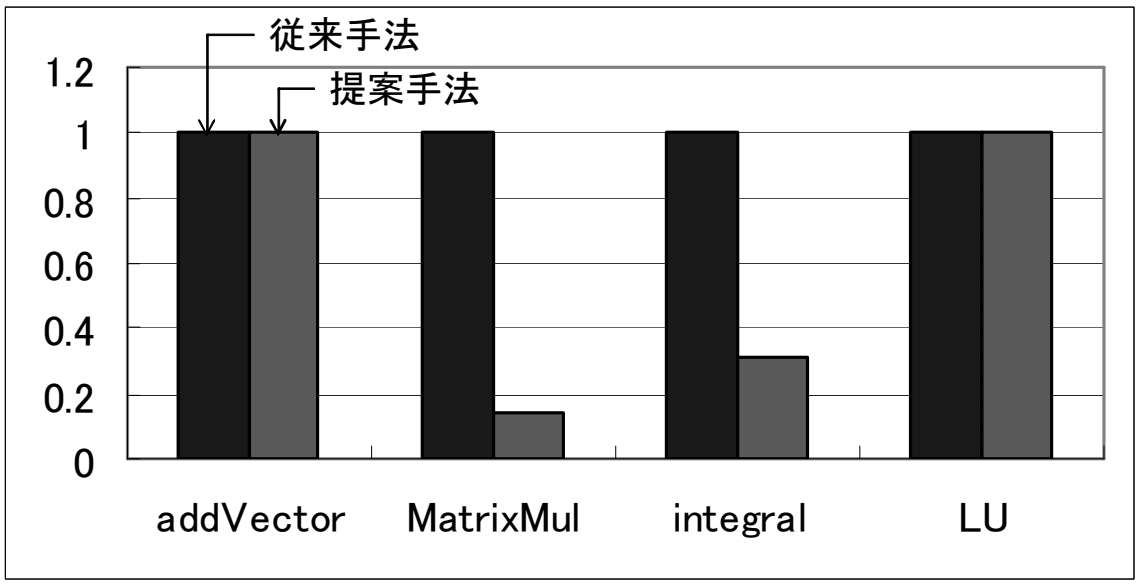

図 17: 評価結果

Development Kit for Linux on Power[13] を用いた．また，OpenCL 対応の計算デバイスとして Cell/B.E. の SPE 6 基を評価に用いた．コンパイルには gcc 4.1.1 を使用し，最適化オプション-O3 を用いてコンパイルした．ベンチマークプログラムには，配列の加算を行う addVector，行列積演算プログラム MatrixMul，台形公式による積分演算プログラム integral，LU 分解を行う LU を用いた．

6.2 評価結果

評価結果を**図 17** に示す．グラフはプログラム毎に左側が従来手法，右側が提案手法の実行時間を表しており，従来の OpenCL の実行時間を 1 として正規化している．な

お，実行時間にはホストがコンテキストを作成するオーバヘッドなども含まれるが，十分な数のデータを用いて実行しているため，それらのオーバヘッドが全体の処理時間に占める割合は非常に小さい．

提案手法の適用により，MatrixMul は統合数を 16，ワークグループサイズを 32 として設定するよう変換され，integral は統合数を 16，ワークグループサイズを 16 として設定するよう変換された．グラフから MatrixMul および integral において，提案手法を用いることで大きく速度向上できていることがわかる．具体的には MatrixMul では 5.2 倍，integral では 3.8 倍の速度向上を実現した．しかし，addVector および LU においては実行時間に変化が見られなかった．

6.3 考察

addVector，LU では速度向上が得られなかったが，これはどちらのベンチマークプログラムもプリプロセッサの判定により，提案手法が適用されなかったからである．まず addVector は，1 度の加算しか行わない非常に軽量なプログラムとなっており，インデクスの計算コストが切り替えオーバヘッドの削減に勝るとプリプロセッサに判定されたことにより，提案手法が適用されなかった．次に LU は，データ間に依存関係が存在するプログラムとなっており，提案手法を適用し操作順序を変更した場合，処理結果の正当性を保証できなくなるとプリプロセッサが判定したため，提案手法が適用されなかった．

一方で MatrixMul と integral では，大きな速度向上が得られた．MatrixMul はデータの要素間で依存関係が無く，完全なデータ並列性を持つため，提案手法による効果が大きいと考えられる．また，プログラム内に生成数を削減可能な変数および条件判定回数を削減可能な制御文が存在したため，提案手法を適用した結果，1 つの変数の宣言と条件判定回数を累計で 16 分の 1 回に削減できたことが，高速化につながったと考えられる．

また integral も同様に完全なデータ並列性を持つため，提案手法による速度向上が得られたと考えられる．さらに integral は，処理対象となるデータが膨大なプログラムとなっているため，OpenCL で実行する際は，非常に多くのワークアイテムが作成される．こうしたプログラムは，ワークアイテムの切り替えオーバヘッドが全体の処理に占める割合が高いため，提案手法を適用することにより，切り替えオーバヘッドが削減できたことも高速化の要因になったと考えられる．

さらに，プログラムが変換されなかった addVector と LU については，プリプロセッ

サの判定の妥当性を検証するために，手動でプログラムを書き換えて実行時間を比較した．プログラムの変換の際，addVector は統合数を 16，ワークグループサイズを 8 とし，LU は統合数を 4，ワークグループサイズを 8×8 とした．ここで，LU はデータ依存により演算順序を変更できないため，単純に統合したプログラムを用いて実行した．その結果，従来手法の実行時間と比較して，addVector では 10%の速度低下，LU では 98%の速度低下となったため，プリプロセッサの判定は妥当であると言える．addVector の速度低下の原因は，速度低下の度合が少ないことから，インデクスの計算コストの増加であると考えられるが，LU において 98%もの大きな速度低下を招いた原因は，インデクスの計算コストの増加以外にも存在すると考えられる．この原因については現在調査中である．

addVector と LU は速度低下し，integral では速度向上したことから，ワークアイテムの切り替えオーバヘッドの削減が実行速度に与える影響は，プログラムの性質に依存すると考えられる．また提案手法の適用により，グローバル ID の取得回数は削減されるが，これにより得られる実行時間の短縮はグローバル ID の計算にかかるコストに相殺されてしまうため，速度向上の要因にならないと考えられる．しかし，削減可能な処理が存在するプログラムの場合は，大きな速度向上を得ていることから，冗長な変数の生成や条件判定が多大なオーバヘッドとなっていることがわかる．今回利用したベンチマークプログラムは比較的軽量であったが，ベンチマークプログラムの処理内容が複雑になるにつれて，変数の生成や条件判定の回数が増える可能性があるため，そのようなベンチマークを実行した場合ではより提案手法の効果が得られると考えられる．

7　終わりに

本論文では，ソフトウェア開発環境である OpenCL において，データ並列処理を行う際に発生するオーバヘッドについて述べ，それらを解消するための手法を提案した．提案手法は 2 つあり，1 つはワークアイテムの切り替えによるオーバヘッドの削減および，演算の順序変更による高速化のために，ワークアイテムを統合する手法である．もう 1 つは，ワークグループサイズを変更して，CU の処理量を調整する手法である．さらに，提案手法に基づくプログラム変換を自動的に行うプリプロセッサを実装し，その有効性を確認するために Cell/B.E. を用いて評価を行った．その結果，従来手法での実行と比較して最大で 5.2 倍の速度向上が得られ，提案手法の有効性を確認できた．また，速度向上が得られないプログラムに対しては，プリプロセッサが適切に，速度

向上が得られないと判定することができた.

本研究の今後の課題としては，まずプロファイラを用いてより精密な解析，判定を行うことにより，今回高速化が得られなかったプログラムを高速化することが挙げられる．また，実行環境毎に特化したプログラムの変換をすることが考えられる．本研究では実際に扱う計算デバイスを意識すること無く処理量を調整しているが，計算デバイスの特徴を上手く活かした変換をする事で，さらなる高速化が得られる可能性がある.

謝辞

本研究の為に，多大な御尽力を頂き，御指導を賜わった名古屋工業大学の松尾啓志教授，津邑公暁准教授，齋藤彰一准教授，松井浩准教授に深く感謝致します．また，本研究へ多くの助言，協力をして頂いた松尾・津邑研究室及び齋藤研究室，松井研究室の方々にも感謝致します.

参考文献

[1] Sony Computer Entertainment: *Cell Broadband Engine Architecture*, 1.01 edition (2006).

[2] Khronos OpenCL Working Group: *The OpenCL Specifiction*, 1.0 edition (2009).

[3] : OpenCV on the Cell, http://cell.fixstars.com/opencv/index.php/OpenCV_on_the_Cell.

[4] : CTK: Cell ToolKit Library, http://cell.fixstars.com/ctk/.

[5] NVIDIA Corp.: *NVIDIA CUDA Programming Guide*, 2.0 edition (2008).

[6] AMD: ATI CTM Guide, http://ati.amd.com/companyinfo/researcher.

[7] Han, T. D. and Abdelrahman, T. S.: hiCUDA: High-Level GPGPU Programming, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 22, pp. 78–90 (2011).

[8] Wolfe, M.: *Implementig the PGI Accelerator model*, The Portland Group.

[9] Lee, S. and Eigenmann, R.: OpenMPC: Extended OpenMP Programming and Tuning for GPUs, *SC'10: Proceedings of the 2010 ACM/IEEE conference on Supercomputing*, IEEE press (2010).

[10] Lee, S., Min, S.-J. and Eigenmann, R.: OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization, *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel pro-*

gramming, ACM, pp. 101–110 (2009).

[11] YUSUKE ARAI, K. S.: *Performing Evaluation of GPU Computing with OpenCL*, Department of Mechanical and Aerospace Engineering,School of Engineering,Tohoku University (2010).

[12] Jaejin Lee, Jungwon Kim, S. S.: *An OpenCL Framework for Heterogeneous Multicores with Local Memory*, Seoul National University,Samsung Electronics Co. (2010).

[13] IBM: *OpenCL Development Kit for Linux on Power*, 0.3 edition (2011).