# マルチプラットフォーム対応<br>並列プログラミングフレームワーク

# Multi-Platform Framework For Parallel Programming

平成26年度 3月 学位論文(修士)

琉球大学大学院　理工学研究科
情報工学専攻

渡真利 勇飛

# 要 旨

Cerium は並列プログラミングフレームワークである。Cerium は Cell Broad band Engine 用のフレームワークであったが、マルチコア CPU や GPU といったマルチプラットフォームに対応させ、より汎用的なフレームワークへ改良を行った。

マルチコア CPU は SynchronizedQueue 、 GPU は OpenCL と CUDA により実装した。データ並列実行をサポートすることで GPGPU を行う際に高い並列度を維持することを可能にした。また、Cerium ではファイル読み込みを mmap 及び専用スレッドを用いる方法で十分な性能が出ることを示した。Sort、WordCount 、FFT の例題を元に測定評価を行った。

# Abstract

Cerium is a parallel programming framework, which originally designed to work on Cell Broad band engine. We add Multi-platform supports which includes Many core, GPGPU. GPGPU part uses OpenCL or CUDA and it support multi dimensional computation i.e. data parallel computation on GPGPU. We also improve I/O for parallel processing using a separate I/O thread. We evaluate Cerium by Sort, Word count, and FFT.

# 目 次

# 図 目 次

# 表 目 次

# 第1章　マルチプラットフォームなフレームワークにおける並列プログラミング

プログラムが PC に要求する処理性能は上がってきているが、消費電力・発熱・クロックの限界から、 CPU の性能を上げることによる処理性能の向上は難しい。プロセッサメーカーはマルチコア CPU や GPU を含むヘテロジニアス構成の路線を打ち出している。クロックの性能を上げるのではなく、コア数を増やすことでパフォーマンスを向上させている。

マルチコア CPU や GPU といったマルチプラットフォームなアーキテクチャ上でリソースを有効活用するには、それぞれのプラットフォームに対して最適な形でプログラムを並列に動作させる必要がある。ここでいう最適な形とは、実行の順番やどのリソース上で Task を実行するかといった Scheduling を含めたチューニングの事である。

しかしこれらのチューニングは複雑で、コーディング時に毎回行うと煩雑さや拡張性の点で問題がある。そういった問題を解決するため、本研究では並列プログラミングフレームワーク Cerium の開発を行った。Cerium によりプラットフォーム間で統一的にコードを記述できる。一般的なマルチプラットフォームフレームワークとして OpenCL や CUDA が挙げられるが、これらと同等の並列度を維持したい。

Cerium では Task という単位で処理を記述し、それらは並列に実行される。プログラマが Task 間の依存関係を設定するとパイプラインが構成され、高い並列度で実行される。プログラマが生成した Task は TaskManager に送信され、設定した依存関係の解決を行った後、 Scheduler に送信される。Scheduler がパイプラインの機構を持っており、Task はそのパイプラインに沿って実行される。Scheduler が受信した Task は既に TaskManager が依存関係を解決しているため、実行順序は任意で良い。

計算機において並列処理を行うための設計様式として、1 つの命令を同時に複数のデータに適用し、並列に処理する方式がある。これを SIMD と呼ぶ。GPU や Cell といったヘテロジニアスなアーキテクチャでは、 SIMD 型が主流である。Cerium は SIMD 型において高い並列度を維持するため、データ並列実行の機構を備えている。

Cerium においてプログラマは Task を記述し、Input データを用意した後はデータ並列用の API で Task を spawn するだけでデータ並列実行することができる。TaskManager はプログラマが記述した単一の Task を複数生成し、受け取ったデータ (Input/Output) に対しその Task を割り当てる。生成した復数の Task を並列実行する事でデータ並列実

行を実現した。

Sort 、Word Count 、FFT の例題を元に、Cerium の持つ並列実行機構がマルチプラットフォームにおける並列プログラミングで有効に作用する事を示す。

# 第2章　既存のマルチプラットフォームフレームワーク

マルチプラットフォームでプログラムを動作させる場合、そのアーキテクチャを意識する必要がある。マルチプラットフォームにはマルチコア CPU 、 GPU や Cell といったヘテロジニアスマルチコアのような様々な構成がある。

## 2.1　GPU/Cell の Architecture

本研究では、 CPU の他に GPU 上でのプログラミング (GPGPU) にも対応する。

GPU(Graphics Processing Unit) は PC の画像処理を担当するユニットで、レンダリングに特化したプロセッサが多く集まった構造を持つ。一つ一つのプロセッサの構造は単純で、その機能は CPU に比べて限定的ではあるが大量のデータを複数のプロセッサで並列処理することに長けている。つまり、データ並列による実行を行った際、特に GPU の性能を充分に発揮できる。

GPGPU (General Purpose computing on Graphics Processing Units) とは、GPU の高い演算能力を画像処理ではなく汎用計算に使用することである。

計算機にはメモリ空間が別の計算機と、共有メモリ (Shared Memory) な計算機がある。GPU のメモリ空間 (図:2.1) はマルチコア CPU (図:2.2) と違い、共有メモリ (shared mermoy) でないので Host と Device 間で Data の共有ができない。そのためマルチプラットフォーム環境に対応したフレームワークには、Device と Host 間でデータの転送を行う API 備わっている。しかし、異なる Device 間でデータの転送を行うとネックになる。そのためデータの入出力を行う回数を減らす、入出力の処理をパイプライン処理にするなどの工夫が必要になる。

Shared Memory でないプロセッサとして、Cell が挙げられる。

Cell は 1 基の制御系プロセッサコア PPE ( PowerPC Processer Element ) と 8 基の演算系プロセッサコア SPE ( Synergistic Processer Element ) で構成される。各プロセッサコアは EIB (Element Interconnect Bus ) と呼ばれる高速なバスで接続されている。また、EIB はメインメモリや外部入出力デバイスと接続されており、各プロセッサコアは EIB を経由してデータアクセスを行う。PPE、SPE、メインメモリ、EIB の構成図を図:2.3 に示す。

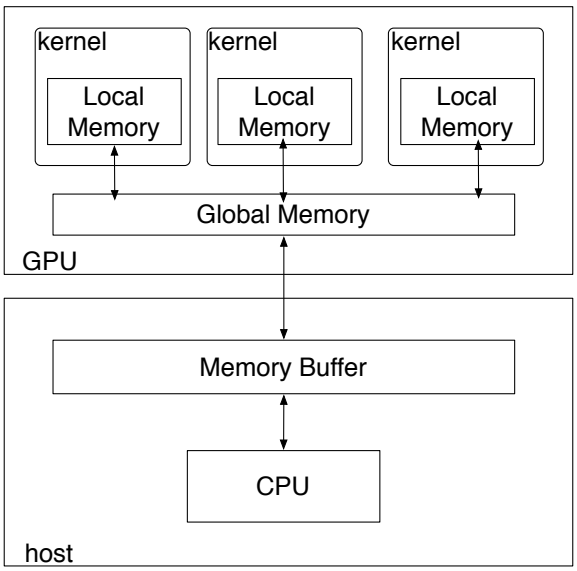


図 2.1: GPU Architecture

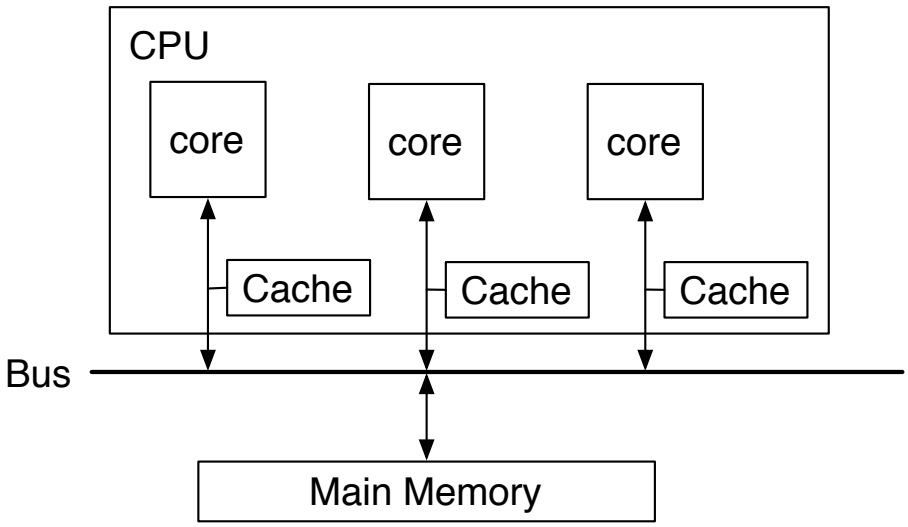


図 2.2: CPU Architecture

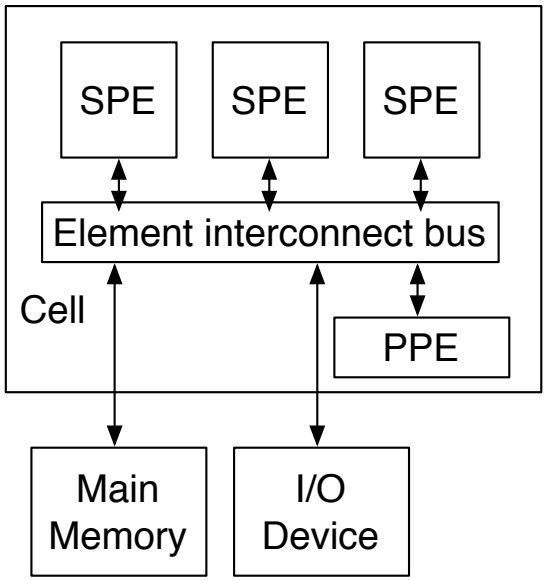


図 2.3: Cell Architecture

## 2.2 OpenCL

OpenCL とは、 Khronos Group の提供するマルチコア CPU と GPU といった、ヘテロジニアス環境を利用した並列計算を支援するフレームワークである。

OpenCL では演算用プロセッサ側を Device 、制御用デバイス側を Host として定義する。また、 Device 上で動作するプログラムの事を kernel と呼ぶ。

OpenCL では、デバイスの操作に Command Queue を使用する。Command Queue は Device に命令を送るための仕組みである。Command Queue は clCreateCommandQueue という OpenCL API で作成され、Command Queue が所属するコンテキストや実行対象となる Device を指定する。

kernel の実行、input data への書き込み、 output data の読み込みといったメモリ操作はこの Command Queue を通して行われる。

OpenCL には主に 2 つの仕様がある。

- OpenCL C 言語
- OpenCL Runtime API

OpenCL C は演算用プロセッサ上で動作する、 C 言語を拡張したプログラミング言語である。一方で OpenCL Runtime API は OpenCL C で記述した kernel を Queuing するために Host が利用する API である。

Host では主に Data を input/output するメモリ資源の確保を行う。OpenCL は host 側で memory buffer を作成してメモリのコピーを行う。これらの処理や Task は Command Queue に enqueue することで実行される。

## 2.3 CUDA

CUDA とは、半導体メーカー NVIDIA 社が提供する GPU コンピューティング向けの総合開発環境である。

CUDA も OpenCL と同様、演算用プロセッサ (GPU) を Device 、制御用デバイス側を Host として定義する。また、 Device 上で動作するプログラムの事も kernel と呼ぶ。

OpenCL における Command と CommandQueue に対応するものとして、 CUDA には Operation と Stream がある。Stream は Host 側で発行された Operation を一連の動作として Device で実行する。Operation は発行された順序で実行されることが保証されている。更に、異なる Stream に発行された Operation も依存関係が存在しない場合、Operation は並列に実行される。更に依存関係が存在しない、異なる Stream に発行された Operation は並列に実行される。

CUDA には主に 3 つの仕様がある。

- CUDA C
- CUDA Runtime API
- CUDA Driver API

CUDA C は GPU 上で動作する、C 言語を拡張したプログラミング言語である。CUDA Runtime API も CUDA Driver API も CUDA C で記述した Kernel を Queueing するために Host が利用する API である。Driver API は Runtime API に比べ、プログラマが管理しなければならないリソースが多くなる代わり、より柔軟な処理を行う事ができる。

Stream は cuStreamCreate という Driver API で生成される。引数に Stream を指定しない API は全て host 側をブロックする同期的な処理となる。複数の Stream を同時に走らせ、 Operation を並列に実行するためには非同期的な処理を行う API を利用する必要がある。

## 2.4 StarPU

StarPU はフランス国立情報学自動制御研究所 (INRIA) の StarPU 開発チームの提供する、ヘテロジニアス環境向けのフレームワークである。GPU の制御に OpenCL と CUDA を用いており、どちらかを選択することで GPU 上で実行することができる。

OpenCL と CUDA における実行の単位は kernel だったが、 StarPU では実行の単位を Task と定義している。

StarPU では Task を制御するために codelet と呼ばれる構造体を使う。codelet を Task 生成時にポインタ渡しすることで、演算を行うリソースや実行する関数等を指定することができる。CPU と GPU で並列に実行する例を 2.1 に示す。

ソースコード 2.1: codelet の例

```
1 starpu_codelet codelet = {
2       .where = STARPU_CPU|STARPU_CUDA,
3       .cpu_func = cpu_function,
4       .cuda_func = cuda_function,
5 };
```

## 2.5 Cell Broadband Engine

Cell Broadband Engine は、ソニー・コンピュータエンタテイメント、ソニー、IBM 、東芝によって開発されたプロセッサである。Cell は PPE と SPE によって構成されており、これらは OpenCL や CUDA で言うところの Device にあたる。PPE は Cell Broadband Engine のメインプロセッサで、複数の SPE をコアプロセッサとして使用できる汎用プロセッサである。メインメモリや外部デバイスへの入出力、SPE を制御する役割を担っている。

SPE は PPE によって制御される演算系のプロセッサである。2.1 節でも述べた通り、SPE からメインメモリへ直接アクセスすることはできず、DMA (Direct Memory Access) 転送によってアクセスを行う。DMA 転送とは CPU を介さずに周辺装置とメモリとの間でデータ転送を行う事で、SPE が持っているコントローラが DMA Controller と通信することで行われる。手順としては以下のようになる。

1. SPE で起動しているプログラムが、コントローラに対して DMA 転送命令を発行
2. SPE の持つコントローラが DMA Controller を介して DMA 転送を開始。この間 SPE で起動しているプログラムは停止しない
3. 転送が開始したら、SPE プログラムが転送の完了を待つ

Cell の PPE には主に 2 つの仕様がある。
OpenCL には主に 2 つの仕様がある。

- SPU 拡張 C/C++
- libSPE2

SPE 上で動作する、拡張された C/C++ の言語を用いる。通常の C/C++ 言語との違いは DMA 転送、SIMD 演算 (加算、減算、乗算) に対応していることが挙げられる。一方で libSPE2 は PPE が SPE を制御するためのライブラリ群である。

# 第3章　並列プログラミング フレームワーク Cerium

Cerium は、当初 Cell 用の Fine-Grain TaskManager として当研究室で開発された。本章では Cerium の実装について説明する。

## 3.1　Cerium の概要

Cerium は当初 Cell 用であったが、現在では Linux、 MacOS X 上で動作する。GPGPU の Data Parallel を含めて同じ形式で記述できる。

Cerium は TaskManager、 SceneGraph、Rendering Engine の 3 つの要素から構成される。本研究では Cerium の TaskManager を汎用計算フレームワークとして改良を行った。

## 3.2　Cerium TaskManager

Cerium TaskManager では、処理の単位を Task としてプログラムを記述していく。関数やサブルーチンを Task として扱い、 Task 間の依存関係を考慮しながら実行される。

ソースコード:3.1 に Host 側で Task を生成する例題を示す。input data を 2 つ用意し、input data の各要素同士を乗算し、output に格納する multiply という例題である。

ソースコード 3.1: Task の生成

```
1  void
2  multiply_init(TaskManager *manager, float *i_data1, float *i_data2, float *o_data) {
3
4      // create task
5      HTask* multiply = manager->create_task(MULTIPLY_TASK);
6      multiply->set_cpu(spe_cpu);
7
8      // set indata
9      multiply->set_inData(0, i_data1, sizeof(float) * length);
10     multiply->set_inData(1, i_data2, sizeof(float) * length);
11
12     // set outdata
13     multiply->set_outData(0, o_data, sizeof(float) * length);
14
15     // set parameter
16     multiply ->set_param(0,(long)length);
17
18     // set device
19     multiply->set_cpu(SPE_ANY);
```

```
20
21      // spawn task
22      multiply ->spawn();
23  }
```

表:3.1 は Task 生成時に用いる API の一覧である。create された Task は Input Data や 依存関係を設定し、spawn することで TaskManager に登録される。

| | |
|---|---|
| create_task | Task を生成する |
| set_inData | Task への入力データのアドレスを追加 |
| set_outData | Task からの出力データのアドレスを追加 |
| set_param | Task へ値を一つ渡す。ここでは length を渡している |
| set_cpu | Task を実行する Device の設定 |
| spawn | 生成した Task を ActiveTaskList に登録する |

表 3.1: Task 生成おける API

次に、ソースコード:3.2 に Device 側で実行される Task (OpenCL、CUDA でいう kernel) の記述を示す。

ソースコード 3.2: Task

```
1  static int
2  run(SchedTask *s) {
3      // get input
4      float *i_data1 = (float*)s->get_input(0);
5      float *i_data2 = (float*)s->get_input(1);
6      // get output
7      float *o_data = (float*)s->get_output(0);
8      // get parameter
9      long length = (long)s->get_param(0);
10
11     // calculate
12     for (int i=0; i<length; i++) {
13         o_data[i] = i_data1[i] * i_data2[i];
14     }
15     return 0;
16 }
```

表:3.2 は Task 側で使用する API である。Host 側で設定した Input Data やパラメタを取得することができる。

| | |
|---|---|
| get_input | 入力データのアドレスを取得 |
| set_output | 出力先データのアドレスを取得 |
| set_param | パラメータを取得 |

表 3.2: Task 側で使用する API

Task を生成する際に設定できる要素は以下の通りとなる。

- Input Data

- Output Data
- Parameter
- CpuType
- Dependency

Input/Output Data, Parameter は関数で言うところの引数に相当する。CpuType は Task が動作する Device を示し、 Dependency は他の Task との依存関係を表す。

## 3.3 Cerium における Task

図:3.1 は Cerium が Task を生成/実行する場合のクラスの構成図である。TaskManager で依存関係が解消され、実行可能になった Task は ActiveTaskList に移される。Active-TaskList に移された Task は依存関係が存在しないのでどのような順番で実行されても良い。Task は転送を行いやすい TaskList に変換され、CpuType に対応した Scheduler に転送される。なお、転送は Synchronozed Queue である mail を通して行われる。

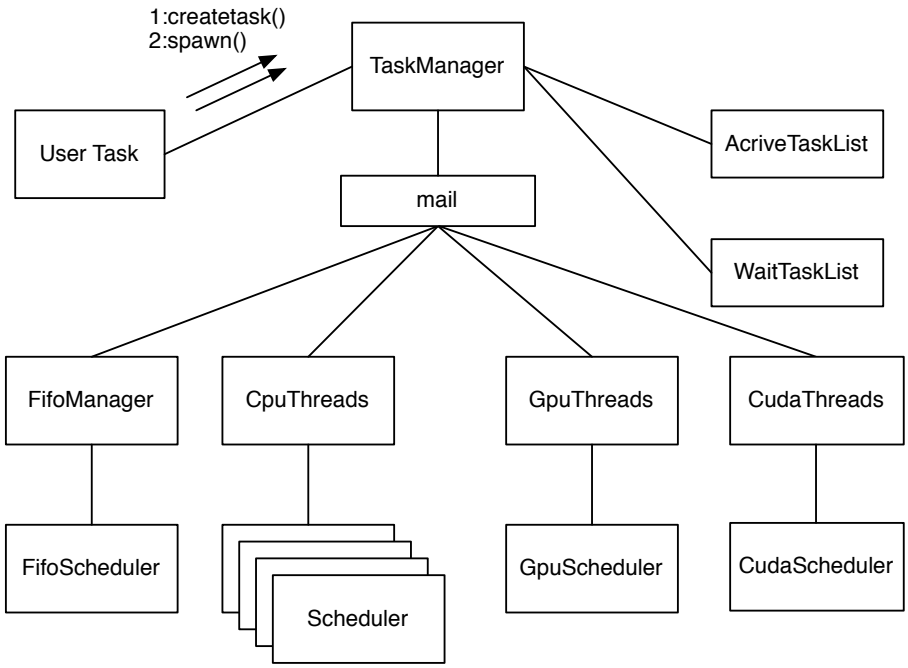


図 3.1: Task Manager

## 3.4 Task の Scheduling

GPU や Cell のような Shared Memory でない環境でのプログラミングを行う場合、Task の入出力となるデータを転送し、転送が終わってから Task を起動しなければならない。転送処理がボトルネックとなり、並列度が低下してしまう。そのため、Cerium はパイプライン実行をサポートしている。

Scheduler に転送された Task はパイプラインで処理される (図:3.2)。Task が全て終了すると Scheduler から TaskManager に mail を通して通知される。通知に従い依存関係を解決した Task が再び TaskManager から Scheduler に転送される。

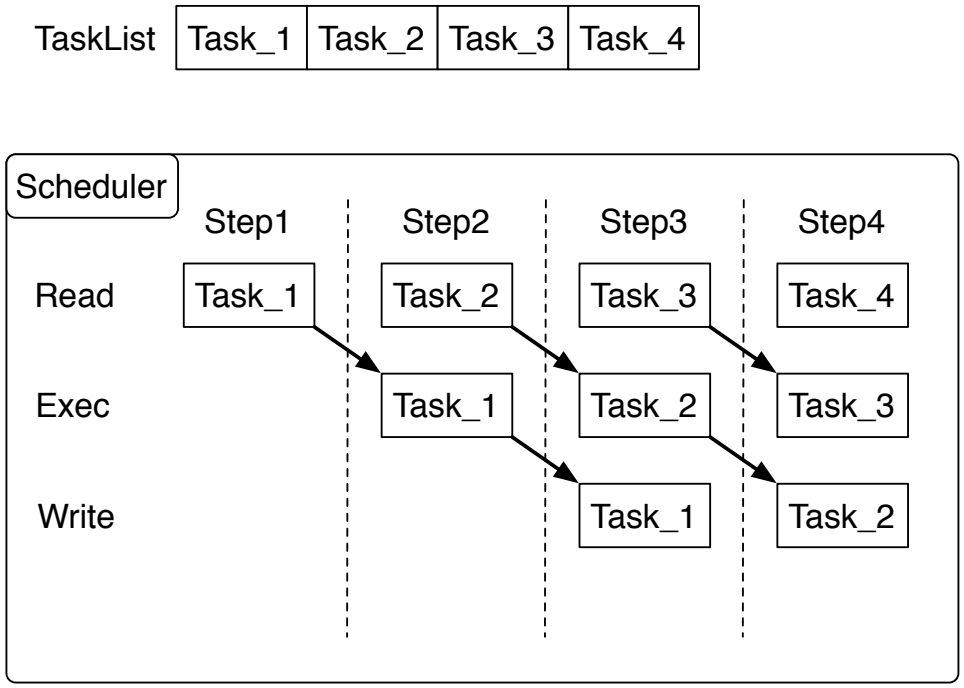


図 3.2: Scheduler

Cerium の Task は SchedTask と呼ばれるデータ構造で表現されている。SchedTask は input/output data の length や合計 size を持っており、これらのパラメタから自分の data が格納されているアドレスを算出し、read/write を実行する。SchedTask を利用することで容易にパイプラインを構築できる。Task をパイプライニングにより Scheduling している部分をソースコード:3.3 に示す。

ソースコード 3.3: Task

```
1 void
2 Scheduler::run(SchedTaskBase* task1)
3 {
4     // Pipeline Stage
5     SchedTaskBase* task2 = new SchedNop();
6     SchedTaskBase* task3 = new SchedNop();
```

```
7
8       // main loop
9       do {
10
11          task1->read();
12          task2->exec();
13          task3->write();
14
15          delete task3;
16
17          task3 = task2;
18          task2 = task1;
19          task1 = task1->next(this, 0);
20
21      } while (task1);
22
23      delete task3;
24      delete task2;
25  }
```

引数として受け取っている task1 は Task のリストである。このリストがなくなるまでパイプライン実行のループを回す。task1 が read、task2 が exec、task3 が write を担当している。つまり task3 には read と exec が終わった Task が来るため、write が終わったら delete して良い。各 Task はそれぞれの処理を行い、task2 は task3 に、task1 は task2 に自分の Task を渡していく。

このメインループを回すことで Cerium の Scheduler はパイプラインによる実行を可能にしている。

# 第4章　Cerium を用いた例題

Cerium は様々な例題を含んでいる。本論文では Bitonic Sort、 Word Count、 FFT の 3 つの例題を扱う。

Bitonic Sort は、ベンチマークをとる際の一般的な例題として選択した。

Word Count は、計算自体は条件に合う word をカウントアップしていくシンプルな内容である。シンプルな計算でも並列化する事で大きな性能向上を狙える事を示す。

FFT:Fast Fourier Transform(高速フーリエ変換) は、信号処理や画像処理から大規模シミュレーションに至るまで幅広い分野で活用されている計算である。バタフライ演算などの計算の性質上、データ並列と相性がよく、 GPGPU で高い並列度を維持できる事が知られている。

以上 3 つの例題を用いてベンチマークを行っていく。本論文で使用する各種例題について紹介する。

## 4.1　Bitonic Sort

Cerium Task Manager を使った Sort である。Bitonic Sort は配列の分割を行い、分割した部分に対して sort を行う。分割後の Sort には QuickSort を使用している。Task の構成は以下のようになる。

- SortSimpleTask
- QuickSortTask

指定された数の乱数を生成し、sort する例題である。SortSimpleTask は Task の割り当てを行う Task である。QuickSortTask は割り当てられた範囲を QuickSort により Sort する Task である。図:4.3 に Bitonic Sort の例を示す。SimpleSortTask は乱数列を分割し、QuickSortTask に割り当てる。QuickSortTask は割り当てられた部分を Sort する。分割した部分を QuickSortTask に割り当て、繰り返し起動していく事で Sort を行う。

1. SimpleSortTask が乱数列を分割し、 QuickSortTask に割り当てる
2. QuickSortTask が割り当てられた部分を Sort する
3. SimpleSortTask が最初に割り当てた範囲の中間から次の範囲の中間までを QuickSortTask に割り当てる

4. QuickSortTask が割り当てられた部分を Sort する

このような Task の分割 → Sort を分割数分繰り返し実行することで全体を Sort する。
本論文では Bitonic Sort による測定を行う場合、10 万入力を Input とするベンチマークを行う。

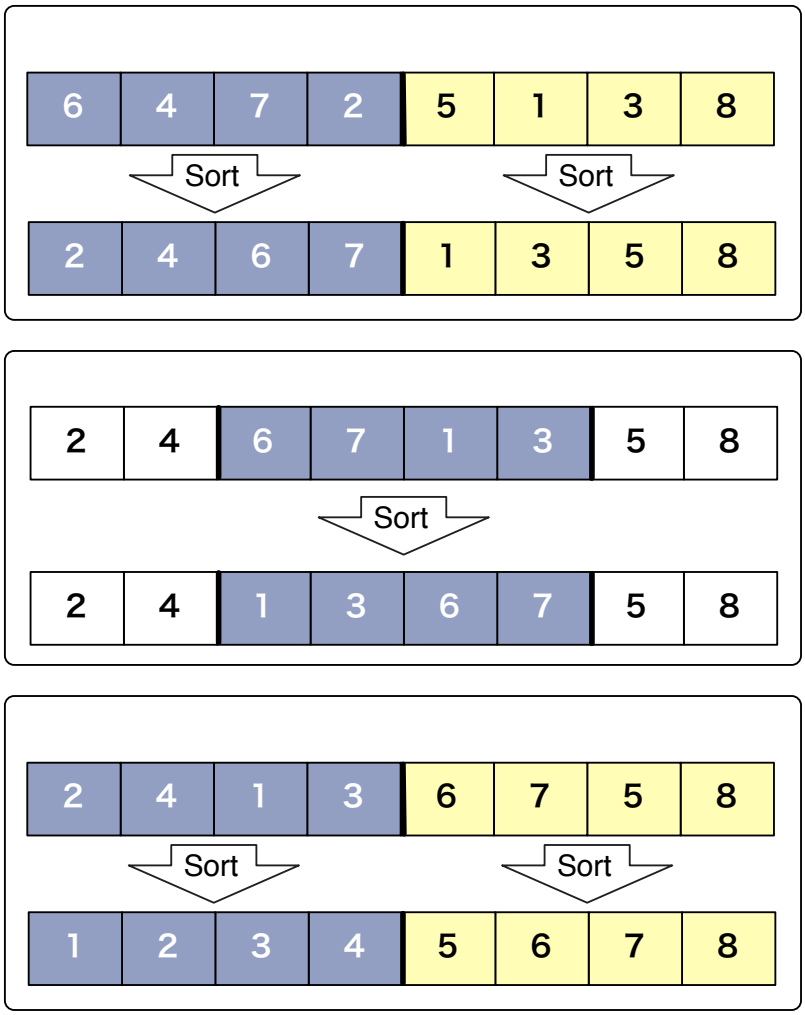


図 4.1: Bitonic Sort の例

## 4.2 Word Count

WordCount は Input としてファイルを受け取り、ファイルの単語数と行数を集計し、表示する。空白で区切られたものを単語として扱う。

Word Count の Task の構成は以下の通りである。

- WordCountTask
- PrintTask

WordCountTask は Input された data を Word Count し、単語数と行数を Output として指定された Data 領域に書き込む Task である。

Task には分割されたテキストが送られてくるため、送られてきたテキストの前後の状態によっては振る舞いを変える必要がある。分割により Word の途中で切れてしまう場合があり、その場合だと word 数-1 する処理が必要になる。そのため、 WordCountTask は自分が割り当てられた範囲である data の先頭と末尾のパラメータを返す。

PrintTask は WordCountTask によって書き出された単語数と行数を集計し、出力する Task である。集計時は全 WordCount の結果を照らし合わせ、分割されたテキストを正しく整合する。PrintTask は WordCountTask を wait する設定で、全ての WordCountTask が終了したあとに動作する。

WordCount の対象として入力されたファイルは、 mmap を用いてメモリに展開する。その後データを 16KByte の大きさに分割しながら WordCountTask に割り当てていく。

各 Task 間の data の流れを図:4.2 に示す。

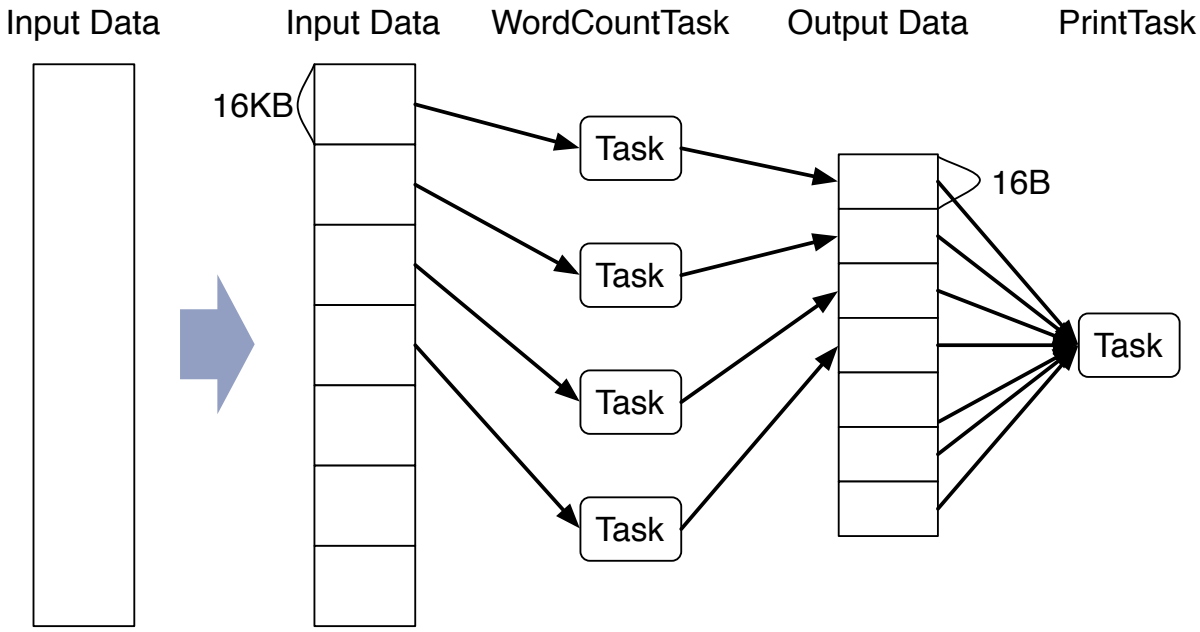


図 4.2: WordCount のフロー

## 4.3 FFT

FFT:Fast Fourier Transform(高速フーリエ変換) は、フーリエ変換と周波数フィルタによる画像処理を行う例題である。今回は入力として受け取った画像に対してハイパスフィルターを行う例題である。

FFT の Task の構成は以下のとおりである。

- BitReverse
- Butteerfly
- HighPassFilter
- SpinFact
- Transpose

FFT は画像 Data に対して様々な Task を割り当てる。入力は比較的大きなサイズを想定している。Input と Output を繰り返し行うと、特に GPU だとボトルネックになってしまう。このベンチマークで並列度を維持するにはデータ並列実行に対応し、データ依存で並列化を可能にする必要がある。

## 4.4 Task の生成

Cerium において並列処理を行う場合、Task を大量に生成する場合がある。WordCount や BitonicSort がそれにあたり、データの分割数分 Task の生成を行う必要がある。しかし、そういった場合において一気に Task を生成して実行を行うと著しく性能が低下する。起動していなくても Task そのものがメモリを圧迫してしまい、処理速度が低下する。Task の生成と実行を並行して行う必要があり、Task は徐々に一定数ごとに生成されるべきである。

Sort の例題を元に Task の生成について考える。Sort の手順は 4.1 節でも述べたが、「乱数列を分割して Sort 」と「割り当てた範囲の中間から次の範囲の中間までを Sort 」2 つの Sort により構成される。「乱数列を分割して Sort」を fsort 、「割り当てた範囲の中間から次の範囲の中間までを Sort 」を bsort とする。2 つの Sort の構成を図:**??**に示す。

bsort は fsort の結果に対して Sort を行う。つまりこの 2 つの Sort 間には依存関係が存在する。更にステージが次へ進むと fsort は前のステージの bsort の結果に対して Sort を行うため、更に依存関係が存在する。例題に関する依存関係を記述しながら、Task を徐々に生成する依存関係も記述するため、Task 生成部分は複雑になるという問題がある。Sort の例題における依存関係の記述を行う Task を SortSimple とし、ソースコード:**??**に示す。

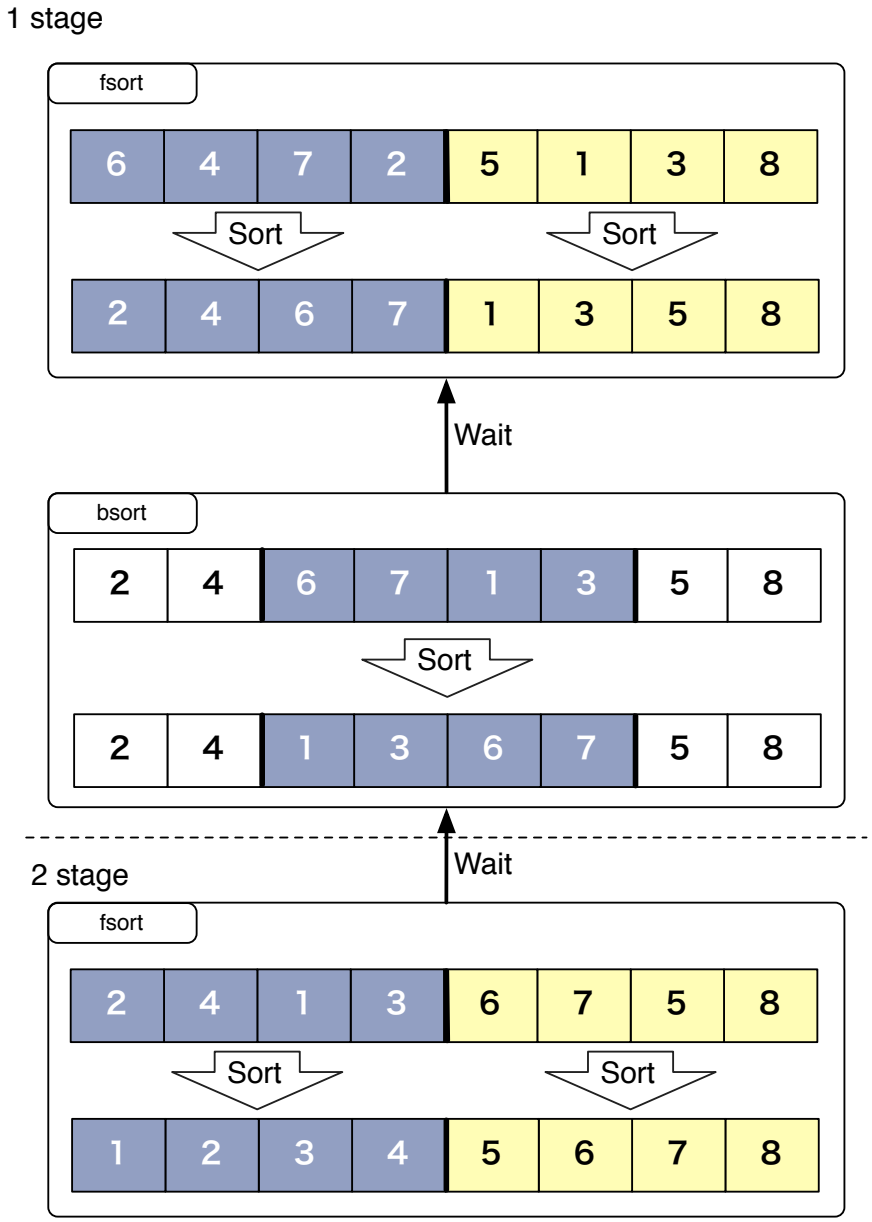


図 4.3: fsort と bsort

ソースコード 4.1: Sort の例題における依存関係の記述

```
1  static int
2  sort_start(SchedTask *manager, void *d, void *e)
3  {
4      Sort *s = (Sort*)manager->get_param(0);
5      long half_num = s->split_num-1;
6
7      for (int i = 0; i < s->split_num-1; i++) {
8          s->fsort[i] = manager->create_task(QUICK_SORT,
9                                             (memaddr)&s->data[i*block_num], sizeof(Data)*
                                                   block_num,
10                                            (memaddr)&s->data[i*block_num], sizeof(Data)*
                                                   block_num);
11
12         if (i>0 && s->bsort[i-1]) {
13             s->fsort[i]->wait_for(s->bsort[i-1]);
14         }
15         if (i<s->split_num-2 && s->bsort[i]) {
16             s->fsort[i]->wait_for(s->bsort[i]);
17         }
18     }
19
20     HTaskPtr restart = manager->create_task(SortSimple,0,0,0,0);
21     restart->set_param(0,(memaddr)s);
22     if (!all) restart->wait_for(s->fsort[0]);
23     for (int i = 0; i < s->split_num; i++) {
24         s->fsort[i]->spawn();
25     }
26     if (sort_count == 1) {
27         // last loop wait for all task
28          for (int i = 0; i < half_num; i++) {
29             restart->wait_for(s->bsort[i]);
30             s->bsort[i]->auto_free();
31         }
32     }
33     restart->spawn();
34
35     return 0;
36 }
```

- 13 行目 : fsort が前のステージの fsort の結果を wait する
- 16 行目 : bsort が fsort の結果に対して wait する
- 20 行目 : SortSimple 内で SortSimple Task を生成する事で再起を行う
- 23 行目 : このループで分割ブロックの分だけ Task を生成している
- 28 行目 : ループの最後は全ての Task が終了するのを待つ

このように例題ごとの依存関係と、Task をブロックで区切って徐々に生成していくための依存関係の両方を記述しなければならない。Task 間で wait_for することで依存関係を設定するのではなく、 Data Dependency により依存関係を記述できる事が望ましい。

# 第5章　マルチコアへの対応

Cerium は Cell 上で並列に動作するフレームワークであったが、 Mac OS X 、 Linux 上でも並列に実行させることを可能にした。

## 5.1　マルチコア上での実行の機構

Cell には MailBox という機能がある。Cell は Shared Memory でないので、 Memory に直接アクセスできない。そこで MailBox を用いて双方向のデータの受け渡しを可能にしている。MailBox は FIFO キュー構造になっており、Device と Host でこの MailBox に対応させる形で Synchronized Queue を用いて MacOSX 、Linux 用の TaskManager へ MailBox を移植した。

Syncronized Queue はバイナリセマフォで管理されており、Queue を操作しているスレッドは常に一つになる。各スレッドは Input 用と Output 用として Synchronized Queue を 2 つ持っており、管理スレッドから Task を受けて並列に実行する。

## 5.2　DMA

Cell ではデータの受け渡しとして MailBox 以外に DMA 転送を使用する方法がある。CPU を介さずに周辺装置とメモリ間でデータ転送を行う方式である。

Cerium も DMA 転送を用いている箇所がある。しかしマルチコア CPU 上で実行する場合は各 CPU で同じメモリ空間を利用できる。よって DMA 転送を用いていた部分をポインタ渡しを行うように修正し、メモリに直接アクセスさせることで速度の向上が見込める。

更に、DMA には prefetch 機能がある。DMA の転送効率を向上させるため、 DMA で転送する送信データを予め取り込むことができる。Cerium では DMA による prefetch、DMA を用いないポインタ渡し、更に明示的なコピーによるデータ転送をサポートする。

## 5.3　データ並列

並列プログラミングを行う際、並列化の方式としてタスク並列とデータ並列の 2 つがある。Cerium における並列処理は、タスク並列により実現されている。

タスク並列は 1 つのデータに対して異なる処理方法を適用し、それぞれ独立して実行させるものである。一方で 1 つの Task に対して多くのデータを与え、データごとに独立した処理を行わせる事をデータ並列という。プログラムを並列化する際はどちらの並列化手法が適しているか考える必要がある。独立した Task が充分にある場合はタスク並列が有効となる。処理対象となるデータが充分な数のサブデータへ分割することが可能で、サブデータへの処理を行うことが元のデータへの処理と同じになる場合、データ並列が有効となる。

本研究で Cerium でデータ並列による実行を可能にした。Cerium でデータ並列実行を行う場合、Task を spwan API でなく iterate API で生成すればよい。iterate API は複数の length を引数とし、length の値がデータ分割後に各 Task が担当するサイズ、length の個数がデータの次元数となる。これを元に Scheduler が各 Task が担当する index を計算し、Task に set_param する。

index の割り当ての例を表:5.1 に示す。データ数 10 個の入力を持つ Task に対して CPU 数 4、一次元における分割でデータ並列実行した場合の index の割り当ては表:5.1 になる。

この例だと各 CPU に対する index の割り当ては CPU0 は index 0、4、8、 CPU1 は index 1、5、9、CPU2 は index 2、6、CPU3 は index 3、7 となる。

| stage | CPU0 | CPU1 | CPU2 | CPU3 |
|---|---|---|---|---|
| 1 | 0 | 1 | 2 | 3 |
| 2 | 4 | 5 | 6 | 7 |
| 3 | 8 | 9 | | |

表 5.1: データ並列実行時の index の割り当て

並列プログラミングだと、並列化部分が全て同一の Task であるということは少なくない。その際、 Task 生成部分をループで回すことなく、iterate API により簡単な Syntax で記述することができる。

マルチコア CPU 上でデータ並列実行する場合、ソースコード:5.1 のように Task を記述する。なお、2 つの input データの積を output データに格納する例題、 multiply を用いた。

ソースコード 5.1: Multiply(CPU)

```
1  static int
2  run(SchedTask *s, void *rbuf, void *wbuf) {
3      float *indata1, *indata2, *outdata;
4
5      indata1 = (float*)s->get_input(rbuf, 0);
6      indata2 = (float*)s->get_input(rbuf, 0);
7      outdata = (float*)s->get_output(wbuf, 0);
8
9      long id = (long)s->get_param(0);
10     outdata[id] = indata1[id] * indata2[id];
11     return 0;
12 }
```

Task 間で共有する Input/Output データと自分が計算を行う index は Scheduler によ

り送られてきている。get_input 、get_output API を用いて Input/Output データを取得し、get_param API で担当する index を取得する。後は各自担当範囲に対して計算を行うだけでよい。

データ並列で実行する場合、1 つの Input と Output を各 Task 間で共有し、各 Task は自分が担当する index に対してのみ計算を行うため、少ないコピーに抑えられる。

# 第6章　GPGPU への対応

Cerium の新たな演算資源として GPU の使用を可能にした。現在、GPU のような異なる種類のアーキテクチャを搭載した CPU 、つまりヘテロジニアスな CPU が増えている。特定の計算に特化した Task の生成やスケジューリングを行い、 GPGPU により高い並列度を出す研究は様々な分野で行われている。本研究では Cerium を特定の計算に限らず、 GPU を用いて汎用計算を行えるフレームワークに改良する。

## 6.1　OpenCL および CUDA による実装

OpenCL 、CUDA による GPGPU 対応を行った。Scheduler と CpuThreads に対応させる形で OpenCL を用いた GpuScheduler と GpuThreads、CUDA を用いた CudaScheduler と CudaThreads を実装した。それぞれの Scheduler 内で各フレームワークの API を用いて GPU の制御を行っている。

TaskManager から受け取った TaskList をもとに Device 上のメモリバッファを作成する。その後 CommandQueue、 Stream といったそれぞれの Queue に Device 制御用の Command を Queueing していく。

Command は Queueing した順に実行されるので、以下のように Command を Queueing する。

1. Host から Device へのデータ転送
2. kernel の実行
3. Device から Host へのデータ転送

データの転送や kernel の実行は非同期 API を用いることで並列に行うことができる。

通常、フレームワークが依存関係を解決して実行するが、非同期 API を用いる場合はユーザが依存関係を考慮する必要がある。しかし Task の依存関係は TaskManager が既に解決した状態で送ってくるので、Scheduler は依存関係を考慮せずに実行して問題ない。

GPGPU 用の Scheduler は CommandQueue を 2 つ持っており、Task をパイプライン的に実行する。GpuScheduler のパイプライン処理部分をソースコード:6.1 に示す。

ソースコード 6.1: GpuScheduler におけるパイプライン処理

```
1  void
2  GpuScheduler::run() {
3    for (;;) {
4      memaddr params_addr = connector->task_list_mail_read();
5      // read task list mail from DmaManager
6
7      while (params_addr) {
8        // since we are on the same memory space, we don't has to use dma_load here
9        tasklist = (TaskListPtr)connector->dma_load(this, params_addr,sizeof(TaskList),
10                                                   DMA_READ_TASKLIST);
11
12     for (TaskPtr nextTask = tasklist->tasks; nextTask < tasklist->last();
13          nextTask = nextTask->next()) {
14
15           kernel[cur] = clCreateKernel(program, function, &ret);
16
17           int param = 1;
18           for(int i=0;i<nextTask->inData_count;i++) {
19                ListElement *input_buf = nextTask->inData(i);
20                if (input_buf->size==0) break;
21                createBuffer(&memin[cur], param, context, mem_flag, input_buf->size, &ret);
22                ret = clEnqueueWriteBuffer(command_queue[cur], memin[cur].buf[param],
23                                           CL_FALSE, 0, input_buf->size,
24                                           input_buf->addr, 0, NULL, NULL);
25                ret = clSetKernelArg(kernel[cur], param, sizeof(memaddr),
26                                     (void *)&memin[cur].buf[param]);
27                param++;
28           }
29           memin[cur].size = param; // +1 means param
30
31           for(int i = 0; i<nextTask->outData_count;i++) { // set output data
32                ListElement *output_buf = nextTask->outData(i);
33                if (output_buf->size==0) break;
34                createBuffer(&memout[cur], i, context, CL_MEM_WRITE_ONLY, output_buf->size,
                       &ret);
35                ret = clSetKernelArg(kernel[cur], param,
36                                     sizeof(memaddr), (void *)&memout[cur].buf[i]);
37                param++;
38           }
39           memout[cur].size = param - memin[cur].size;
40
41           ret = clEnqueueTask(command_queue[cur], kernel[cur], 0, NULL, NULL);
42
43           for(int i=0;i<nextTask->outData_count;i++) { // read output data
44                ListElement *output_buf = nextTask->outData(i);
45                if (output_buf->size==0) break;
46                GpuBufferPtr mem = memout ;
47                ret = clEnqueueReadBuffer(command_queue[cur], mem[cur].buf[i0], CL_FALSE, 0,
48                                          output_buf->size, output_buf->addr, 0,
49                                          NULL,&memout[cur].event[i]);
50                }
51           cur++;
52           if (STAGE <= cur) cur = 0;
53           wait_for_event(kernel_event, memout, tasklist, cur);
54         }
55         reply = (memaddr)tasklist->waiter;
56         params_addr = (memaddr)tasklist->next;
57       }
58
59       wait_for_event(kernel_event, memout, tasklist, cur);
60
61       unsigned long long wait = 0;
```

```
62          (*connector->end_dmawait_profile)(&wait, &(connector->start_time), &(connector->
                stop_time));
63          connector->mail_write((memaddr)MY_SPE_STATUS_READY);
64      }
65      /* NOT REACHED */
66  }
```

- 4 行目 : DMAManager から tasklist mail を取得
- 9 行目 : 取得した mail から TaskList を read をする。TaskList に格納されている Task 全てを実行し終わるまでループする
- 15 行目 : 二段のパイプラインを形成するため、kernel を 2 つ持作る
- 18 行目 : パイプラインの read 部分。input データを全て kernel の引数として MemoryBuffer に書き込み、kernel に Buffer を set する
- 31 行目 : Output データを書き込む MemoryBuffer を用意し、kernel に set している。
- 40 行目 : パイプラインの exec 部分。kernel を実行する
- 42 行目 : パイプラインの write 部分。kernel は実行した結果を MemoryBuffer に書き込む。MemoryBuffer に書き込まれた値をここで読み出している。
- 52 行目 : パイプラインのステージの切り替えを行っている。wait_for_event 内で依存関係を解決したら実行の終わったステージの MemoryBuffer を delete し、次のステージへ移行する
- 55 行目 : 次の task を読み出し、このループを終了する

DMAManager から転送されてきた Task を読み込み、Input/Output データを取り出す。データは OpenCL の API を介して GPU の MemoryBuffer に送信され、kernel が実行される。実行終了後は MemoryBuffer から戻り値を読み取り、Cerium に Output データとして返している。一連の処理は CommandQueue を介して GPU で実行される。GpuScheduler は CommandQueue を 2 つ持っており、二段のパイプラインが形成される。

全ての Task が終了すると、TaskManager 間の通信を担当する同期キューである mail を通して TaskManager に Task の終了を通知する。終了が通知されると TaskManager でその TaskList に関する依存関係が解消される。

Scheduler 内で Platform や Device ID の取得、 Context の生成、 Kernel の Build と Load 等も行っており、並列処理したい計算のみに集中できる。

## 6.2 データ並列

並列プログラミングにおいて、明示的な並列化部分はループ部分である。GPU は数百個のコアを有しており、ループ部分に対してデータ並列で処理を行うことで CPU より高速に演算を行う事ができる。プログラムの大部分がループであれば、データ並列による実行だけでプログラムの性能は向上する。

OpenCL 、 CUDA ともにデータ並列をサポートしている。OpenCL と CUDA は Task を実行する際にデータをどう分割するか指定し、kernel にデータ並列用の処理を加えることで可能となる。5.3 節で Cerium でマルチコア CPU におけるデータ並列を可能にした。GPGPU においてもデータ並列実行をサポートする。GPU 上でのデータ並列実行もマルチコア CPU と変わらず、iterate API によりデータ並列用の Task を生成することができる。iterate で Task を生成することで Scheduler が OpenCL 及び CUDA の API に適切なパラメタを渡している。Task の生成部分は マルチコア CPU と GPU で完全に同じ形式で記述できる。

データ並列実行の際、Task は以下のように記述する。なお、例題は multiply を用いている。

ソースコード 6.2: Multiply(OpenCL)

```
1  __kernel void
2  multiply(__global const long *params,
3           __global const float *input1,
4           __global const float *input2,
5           __global const float *output) {
6
7      long id = get_global_id(0);
8
9      output[id] = input1[id] * input2[id];
10 }
```

ソースコード 6.3: Multiply(CUDA)

```
1  __global__ void
2  multiply(__global const long *params,
3           __global const float *input1,
4           __global const float *input2,
5           __global const float *output) {
6
7      int id = blockIdx.x * blockDim.x + threadIdx.x;
8
9      output[id] = input1[id] * input2[id];
10 }
```

このような Task を分割数分生成する。分割数は Task それぞれのフレームワークが用意している API を用いて指定する。

- 自分の計算する範囲を取得 (ソースコード 6.2、ソースコード 6.3 の 7 行目)
- 取得した範囲を計算 (ソースコード 6.2、ソースコード 6.3 の 9 行目)

いずれの Task も上記の手順で処理を行っている。計算する範囲について、 OpenCL では取得用の API を用い、 CUDA では kernel の持つ組み込み変数から算出する。マルチ

コア CPU では引数としてデータを直接渡していたが、OpenCL 、CUDA では上記の方法でメモリバッファから Load し、計算を行う。値渡しや修飾子等若干の違いはあるが、OpenCL 、CUDA ともにマルチコア CPU(ソースコード:**??**) とほぼ同じ形式で kernel を記述することができる。CPU、 OpenCL、 CUDA いずれか 1 つの記述から残りのコードも生成できるようにする事が望ましい。

データ並列で実行する場合、 Input と Output を各 Task 間で共有するため、少ないコピーに抑えられる。CPU ではメモリ領域を節約する事はできるが、 Task と Manager でメモリ領域が同じ (2.1 節) なため、コピーによるオーバーヘッドは少ない。

しかし GPU は SharedMemory ではなく、データの転送がオーバーヘッドとなるため、コピーを減らす事で並列度の向上が見込める。

# 第7章　並列処理向けI/O

ファイル読み込みなどの I/O を含むプログラムは、読み込み時間が Task の処理時間と比較してオーバーヘッドになることが多い。プログラムの並列化を行ったとしても、I/O がボトルネックになってしまうと処理は高速にならない。本項では Cerium に並列処理用の I/O の実装を行う。これにより I/O 部分の高速化を図る。

## 7.1　mmap

Cerium ではファイルの読み込みを mmap により実装していた。しかし、mmap や read によりファイルを読み込んでから処理を実行させると、読み込んでいる間は他の CPU が動作せず、並列度が落ちてしまう。そこで、I/O 部分も並列に動作するよう実装した。

Read を並列に行うには、File Open ではなく mmap を使う方法がある。Cerium でも mmap を使用していた。mmap はすぐにファイルを読みに行くのではなく、まず仮想メモリ空間にファイルの中身を対応させる。メモリ空間にアクセスが行われると、OS が対応したファイルを読み込む。

mmap で読み込んだファイルに Task1、Task2 がアクセスし、それぞれの処理を行う際の例を図:7.1 に示す。

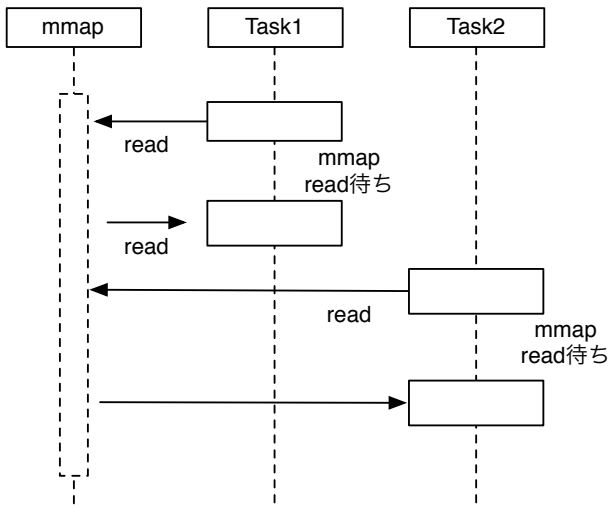


図 7.1: mmap の Model

Task1 が実行される時、仮想メモリ上 Open されたファイルの読み込みを行い、Task1 の処理を行う。その後、同様に Task2 も読み込みを行ってから処理を行う。この Task1 と Task2 の間に待ちが入る。

ファイルの読み込みが起こると、アクセスした Thread/Process には wait がかかってしまう。しかし mmap による Read は Task と並列に実行されるべきである mmap は逐次アクセスを仮定しているので、OS 内部で自動的にファイルの先読みを行う事も期待できる。しかしそれは OS の実装に依存してしまう。読み込みが並列に実行されない場合、Task が読み込み待ちを起こしてしまう。読み込みが OS 依存となるため、環境によって左右されやすく、汎用性を損なってしまう。

そこで、mmap を使わず、read を独立したスレッドで行い、ファイルを一度に全て読み込むのではなくある程度の大きさ (Block) 分読み込み、読み込まれた部分に倒して並列に Task を起動する。これを Blocked Read と呼ぶ。Blocked Read によるプログラミングは複雑になるが、高速化が期待できる。

## 7.2 Blocked Read による I/O の並列化

Blocked Read を実装するに辺り、WordCount を例に考える。Blocked Read はファイル読み込み用の Task(以下、ReadTask) と読み込んだファイルに対して処理を行う Task(今回は WordCount) を別々に生成する。ReadTask はファイル全体を一度に全て読み込むのではなく、ある程度の大きさで分割してから読み込みを行う。分割後の読み込みが終わると、読み込んだ範囲に対して WordCount を行う。

BlockedRead による WordCount を行う場合、図:7.2 のようになる。

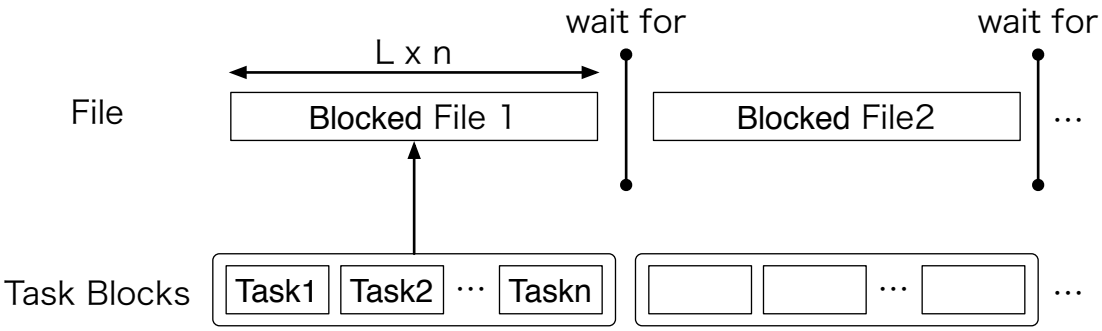


図 7.2: BlockedRead による WordCount

Task を一定数まとめた単位で生成し、起動を行っている。この単位を Task Block と定義する。TaskBlock が BlockedRead を追い越して実行してしまうと、まだ読み込まれてない領域に対して処理を行ってしまう。その問題を解決するため、依存関係を設定する。BlockedRead による読み込みが終わってから TaskBlock が起動されるよう、Cerium の API である wait_for により依存関係を設定する。しかし、Task が BlockedRead を追

い越すことによるロックはオーバーヘッドとなるため、起こさないようにしたい。つまり、 BlockedRead は連続で走っている必要がある。

以上を踏まえ、BlockedRead の実装を行った。BlockedRead Task の生成はソースコード:7.1 のように行う。

ソースコード 7.1: BlockedRead を行う Task の生成

```
1 HTaskPtr readTask = manager->create_task(READ_TASK)
2 readTask->set_cpu(DEVICE_TYPE);
3 readTask->set_outData(0, file_map + task_num * division_size, task_blocks * division_size);
4 readTask->set_param(0, fd);
5 readTask->set_param(1, task_num * division_size);
6 runTask();
7 readTask->set_param(2, task_num * division_size)
8 readTask->spawn();
```

- 3 行目、set_outData(0): ファイルを読み込んだ際の格納場所を設定
- 4 行目、set_param(0): 読み込むファイルディスクリプタを設定
- 5 行目、set_param(1): BlockedRead Task で読み込むファイルの範囲の先頭のポジションを設定
- 7 行目、set_param(2): BlockedRead Task で読み込むファイルの範囲の末尾のポジションを設定

Cerium において、 WordCount で必要な Task を全て生成してしまうと、その Task のデータ構造自体がメモリを消費してしまう。そこである程度の量の Task を起動し、それが終了してから (正確には終了する前に) 次の Task を生成するようになっている。それらの機能を持った関数が 6 行目にあたる runTask である。runTask に wait_for による ReadTask との待ち合わせの処理を入れれば良い。

BlockedRead の Task をいかに示す。

ソースコード 7.2: BlockedRead Task

```
1 static int
2 read_task(SchedTask *s, void *rbuf, void *wbuf) {
3     long fd = (long)s->get_param(0);
4     long start = (long)s->get_param(1);
5     long end = (long)s->get_param(2);
6     char txt = (char*)s->get_output(wbuf, 0);
7     long size = end - start;
8
9     pread(fd, txt, size, start);
10    return 0;
11 }
```

Cerium の API により、生成部分で設定したパラメタをそれぞれ受け取る。ファイル読み込みの先頭・末尾のポジションが渡されているので、ファイルから読み込むサイズは求められる。受け取ったパラメタをそれぞれ pread 関数に渡すことで Blocked Read を実現している。

## 7.3 I/O 専用 Thread の実装

Cerium Task Manager では、各種 Task にデバイスを設定することができる。SPE_ANY 設定を使用すると、 Task Manager で CPU の割り振りを自動的に行う。BlockedRead は連続で読み込まれなければならないが、SPE_ANY 設定で実行すると BlockedRead 間に別の Task が割り込んでしまう場合がある。(図:7.3)

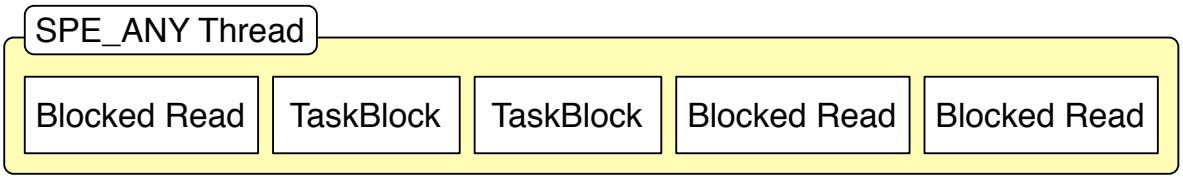


図 7.3: BlockedRead と Task を同じ thread で動かした場合

そこで I/O 専用の Thread である IO_0 の追加を行った。

IO_0 は SPE_ANY とは別 Thread の Scheduler で動くので、SPE_ANY で動いている Task に割り込まれることはない。しかし、読み込みの終了を通知し、次の read を行う時に他の Task がスレッドレベルで割り込んでしまう事がある。pthread_getschedparam() で IO_0 の priority の設定を行う必要がある (図:7.4)。

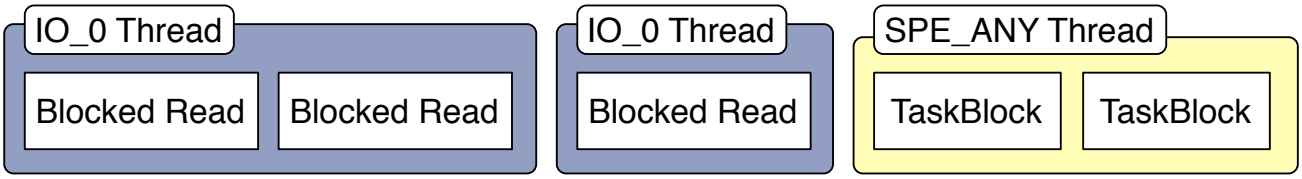


図 7.4: IO Thread による BlockedRead

IO_0 で実行される Task は BlockedRead のみなので、IO_0 の priority を高く設定することで Blocked Read は連続で実行される。

また、以上の事から I/O を含む並列処理において読み込みの Task と計算を行う Task を並列で走らせたい場合、I/O を行う Thread の priority を高くする必要があるという知見を得られた。

# 第8章　ベンチマーク

## 8.1　実験環境

今回使用する実験環境を表:8.1、表:8.2 に示す.

| 名前 | 概要 |
|---|---|
| Model | MacPro Mid 2010 |
| CPU | 6-Core Intel Xeon @2.66GHz |
| Serial-ATA Device | HDD ST4000VN000-1H4168 |
| Memory | 16GB |
| OS | MacOSX 10.10.1 |
| Graphics | NVIDIA Quadro K5000 4096MB |

表 8.1: Cerium を実行する実験環境 1

| 名前 | 概要 |
|---|---|
| Model | MacPro Late 2013 |
| CPU | 6-Core Intel Xeon E5@3.5GHz |
| Serial-ATA Device | Apple SSD SM0256 |
| Memory | 16GB |
| OS | MacOSX 10.10.1 |
| Graphics | AMD FirePro D700 6144MB |

表 8.2: Cerium を実行する実験環境 2

なお、表:8.1 と表:8.2 は CPU クロック数の他にも Strage や GPU の性能にも違いがある。実験環境 1(表:8.1) は実験環境 2(表:8.2) に比べてクロック数が低く、Strage は HDD を使用している。GPU は NVIDIA を使用している。実験環境 2(表:8.2) はクロック数が高く、 Strage に SSD を使用している。GPU は AMD を使用している。

以上の環境で今回新たに実装したマルチコア、GPGPU 、並列 I/O のベンチマークを行う。

## 8.2 マルチコア

マルチコア CPU における並列実行について、Sort(図:8.1) と WordCount(図:8.2) によるベンチマークを行った。

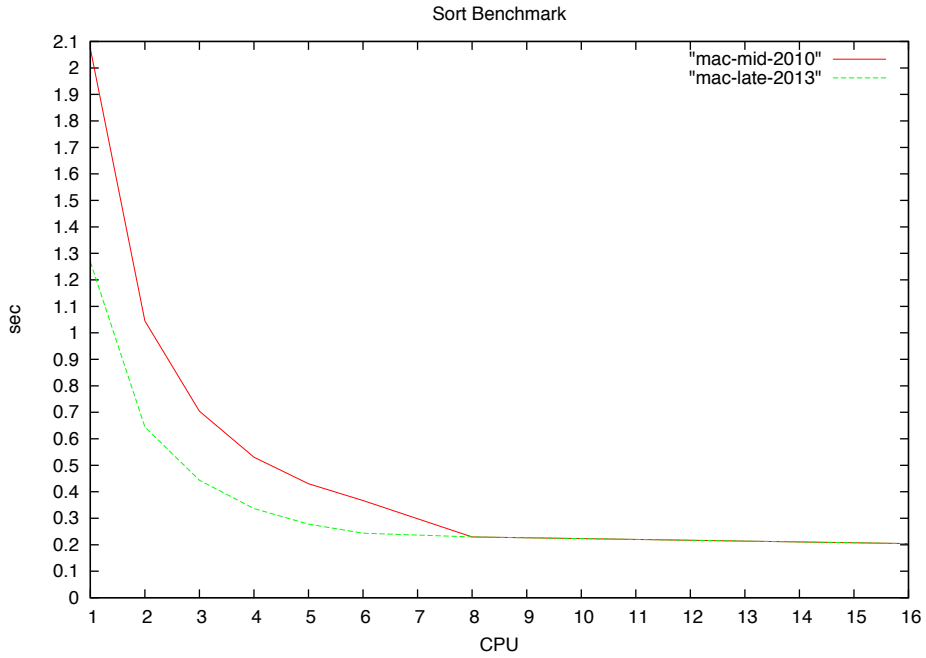


図 8.1: マルチコア CPU における Sort

MacPro 2013 において、6CPU を使用した場合、1CPU を利用した場合と比較して、Sort は 5.2 倍、WordCount は 4.9 倍の速度向上が見られる。MacPro 2010 においても Sort は 5.65 倍、WordCount は 5.0 倍の速度向上が見られた。MacPro のコア数以上の Thread 数になると並列度の低下が見られる。

Cerium は実行時に -pre オプションをつけることで使い分ける事ができる。DMA の prefetch 機能によるデータ転送のベンチマークを行った (図:8.3)。

測定の結果、CPU 数が 1 の場合は prefetch オプションを入れると 1.17%、CPU 数が 6 の場合は 1.63%の性能向上が見られた。

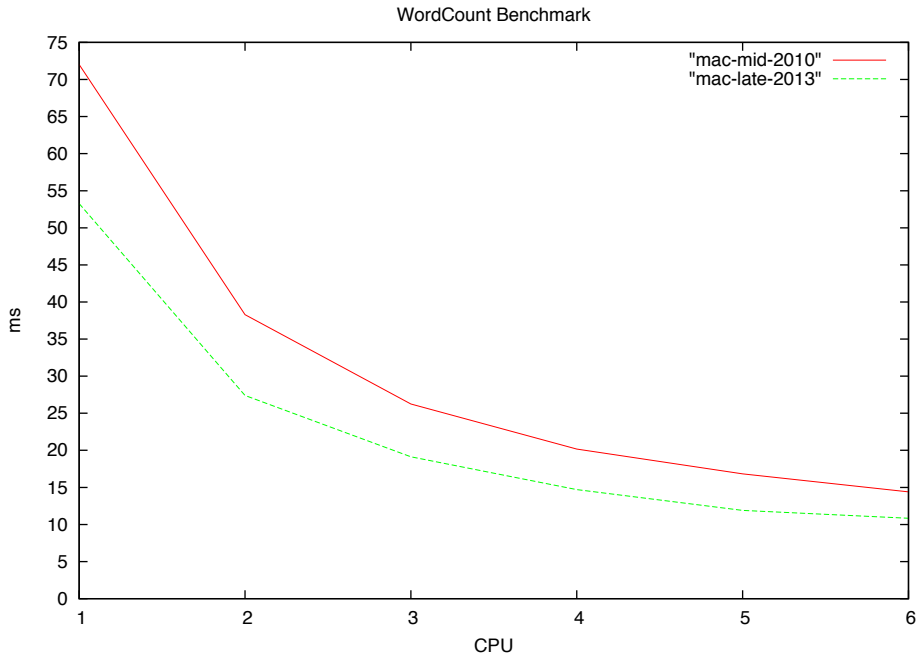


図 8.2: マルチコア CPU における WordCount

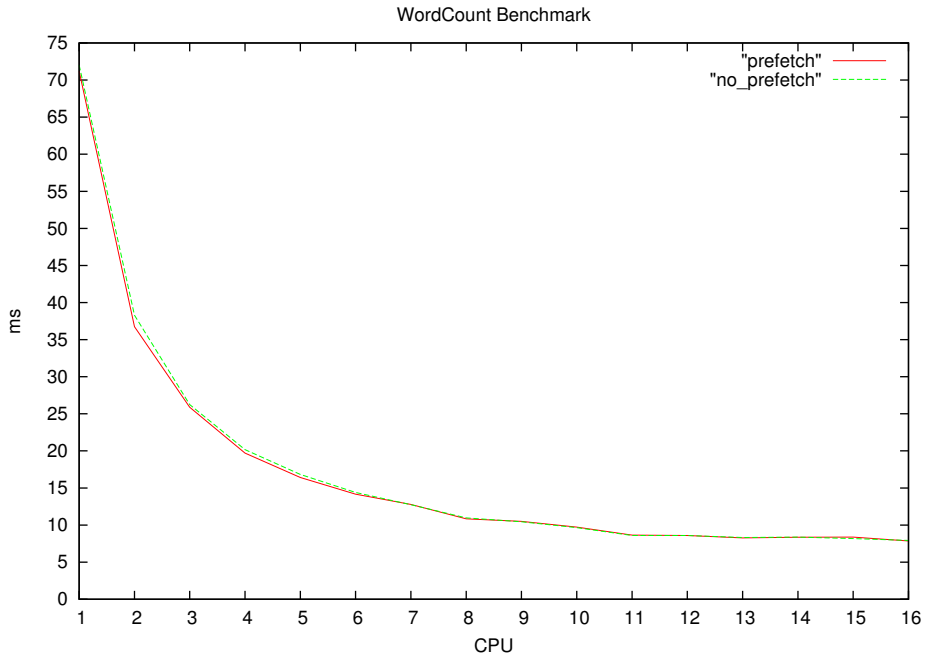


図 8.3: Word Count による prefetch 機能のベンチマーク

## 8.3 GPGPU

GPGPU を行う際はデータ並列による処理を行った時、充分に性能を発揮することができる (2.1 節)。WordCount による OpenCL、CUDA、マルチコア CPU 上におけるデータ並列実行の性能評価を行った (図:8.4)。なお、MacPro 2013 (表:8.2) は GPU が NVIDIA 製でないため、仕様上 CUDA による測定ができない。MacPro 2010 で測定を行った。

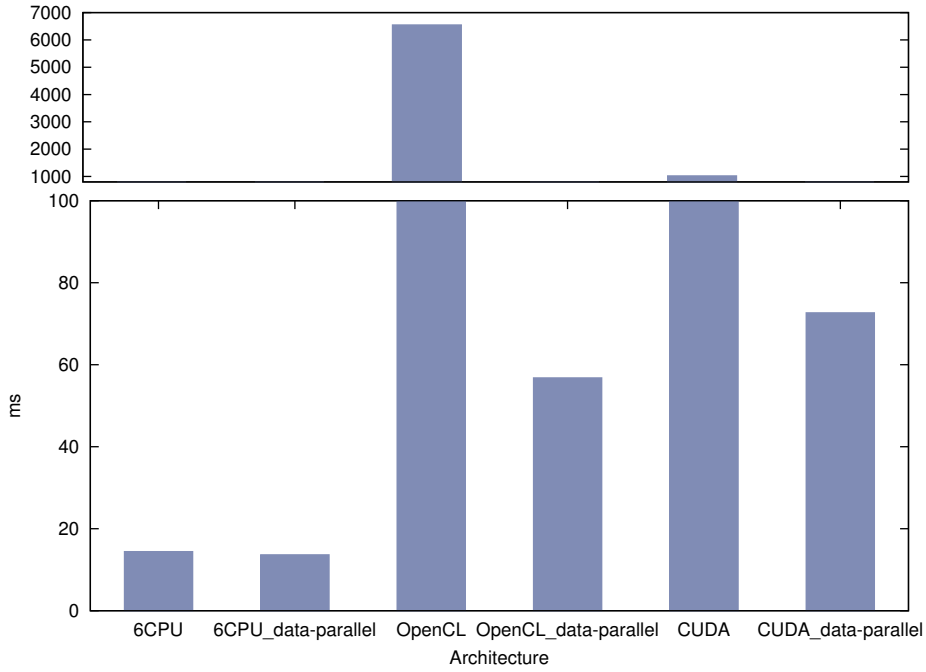


図 8.4: Word Count によるデータ並列実行のベンチマーク

データ並列実行することにより、マルチコア CPU では 1.06 倍の性能向上が見られた。GPU に関しては特に性能向上が大きく、OpenCL においては 115 倍、CUDA においては 14 倍の性能向上が見られた。この結果から、GPGPU を行う際はデータ並列による実行が必須であることがわかる。更に、マルチコア CPU においても性能向上が見られた。データ並列を行わない場合、OpenCL の実行時間が CUDA の約 6 倍かかっている。これは MacPro2010 の GPU が NVIDIA 製であることが要因として考えられる。CUDA は NVIDIA 製の GPU でのみ動作するため、 NVIDIA に特化した最適化が行われていると考えられる。

しかしデータ並列による実行を行った場合、CUDA の実行時間が OpenCL の 1.27 倍かかっており、OpenCL の実行速度が上回っている。これは OpenCL の kernel が cl_float2 という型によりベクター演算をサポートしていることが要因と考えられる。CUDA では cl_float2 を float に変換して演算している。OpenCL でベクター演算の最適化が行われた結果、性能向上が見られたと考えられる。

データ並列によって性能向上は実現できたが、マルチコア CPU と比較すると実行時間が OpenCL では 4.2 倍、CUDA では 5.3 倍かかっている。WordCount の例題は特定の文

字を区切り文字とし、区切り文字による分岐でカウントアップしている。区切り文字による分岐がマルチコア CPU と比較して性能を落としていると考えられる。

GPU は分岐命令を苦手としており、GPU で並列度を維持するには分岐を最小限にする必要がある。

そこで、FFT(図:8.5) による測定を行う。このベンチマークにより、GPU の制約に当てはまる Task であれば並列度を維持できることを示す。OpenCL、CUDA、マルチコア CPU の性能比較を行う。更に、Cerium を用いないで OpenCL を使用した場合 (OpenCL-original) についても測定を行う。

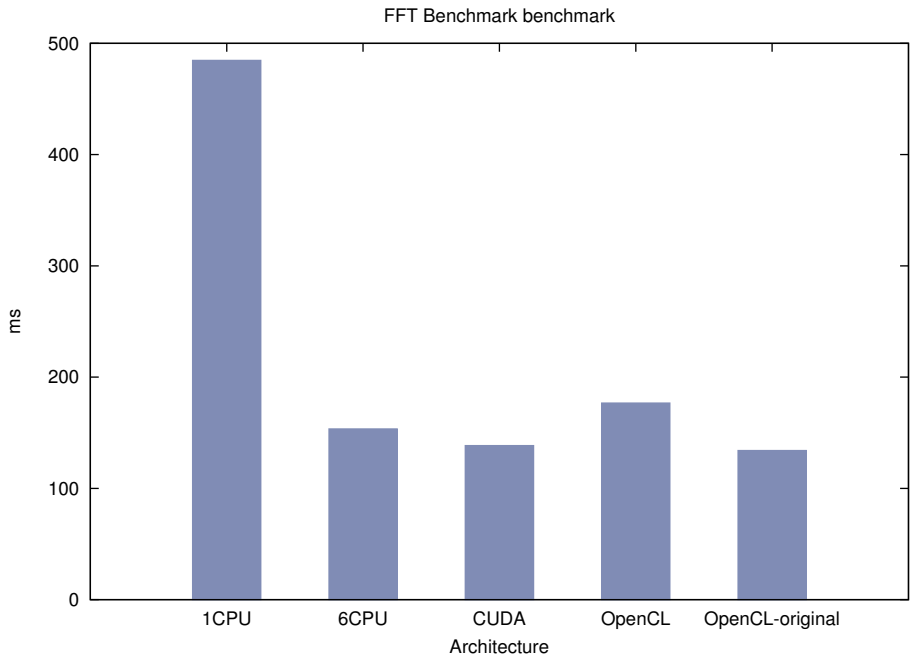


図 8.5: マルチコア CPU、OpenCL、CUDA における FFT

CUDA は 1CPU に比べて 3.5 倍、6CPU に比べて 1.1 倍の性能向上が見られる。OpenCL は 1CPU に比べて 2.75 倍の性能向上が見られたが、6CPU と比べると 0.87 倍、OpenCL-original と比べると 0.76 倍の性能低下が見られた。高性能の GPU を使用することで OpenCL でも並列度が向上が期待できる。また、Cerium では Task のパイプライン実行により、GPU よりも上のレイヤでの並列化を行っている。CPU の性能を上げることで Scheduling 部分が高速になり、OpenCL-original と比較した場合の性能向上も見込める。

そこで、MacPro 2013 にて測定を行った (図:8.6)。

より高性能な GPU を搭載した計算機 (表:8.2) で測定したところ、OpenCL が 1CPU に比べて 6 倍、6CPU に比べて 1.6 倍の性能向上が見られた。マルチコア CPU での実行速度も向上しているため、GPU の性能だけでなく、 CPU のクロック数や Strage に SSD を使用している事も性能向上の要因と考えられる。SSD はランダムアクセスでのデータ読み込み性能が高く、ディスク読み書きに関するオーバーヘッドの改善が見込める。

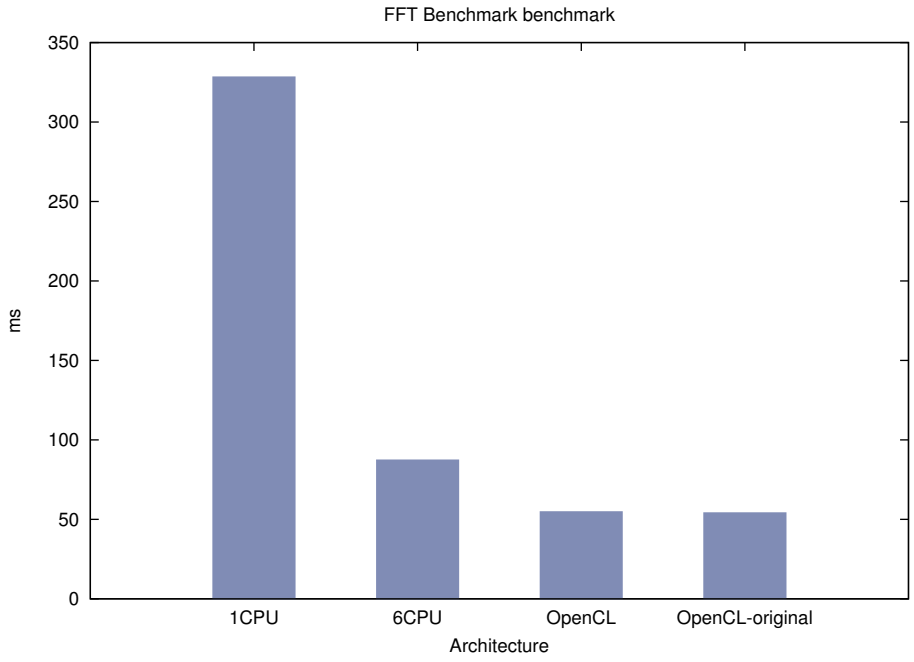


図 8.6: MacPro 2013 における FFT

Cerium による実行は OpenCL-original による実行とほぼ同じ性能で、約 1% OpenCL-original の方が速い。Cerium のパイプライン構造を利用しており、本来 OpenCL-original よりも高い並列度が期待できる、まだチューニングを行う余地がある事がわかる。

## 8.4 並列 I/O

Cerium の従来のファイル読み込みである mmap、一般的な File Open である read、今回実装した Blocked Read を比較した測定を行った。なお、Blocked Read については IO Threads を使用した場合としてない場合 (SPE_ANY) 両方の測定を行う。例題として Word Count を使用した測定を行った。図:8.7 が MacPro 2010 における測定で、図:8.8 が MacPro 2013 における測定となる。

6CPU において BlockedRead_IO を使用した場合、mmap に比べて 1.1 倍、read に比べて 1.58 倍、BlockedRead_SPE_ANY に比べて 1.34 倍の性能向上が見られた。しかし、実験環境のコア数である 6CPU 以上になると並列度は低下していき、8CPU からは BlockedRead による並列実行に比べて mmap によるファイルの先読みが有効に働いている。

また、SPE_ANY による BlockedRead は他の読み込み形式と違い、10CPU の時、CPU 数を増やしたにも関わらず極端に処理が遅くなっている。これは 7.3 節で述べた、ReadTask に対する 実行 Task の割り込みによりロックがかかる問題が起きていると考えられる。IO Thread を用いた BlockedRead では極端な速度低下は起きていない。測定の結果から、IO Thread を用いることでロックの問題が解決できていることがわかる。

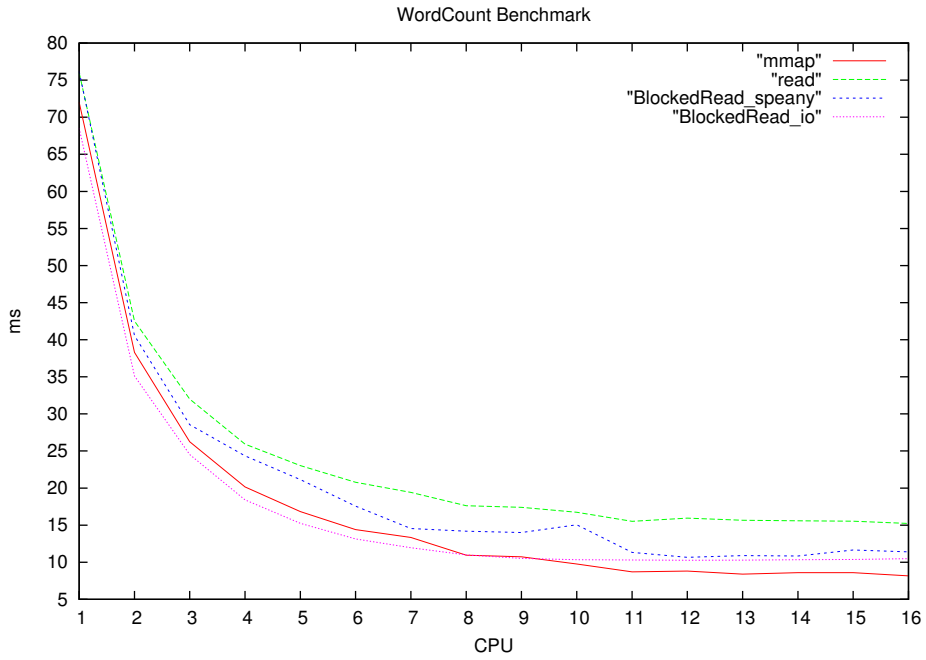


図 8.7: WordCount によるファイル読み込み方式のベンチマーク (MacPro2010)

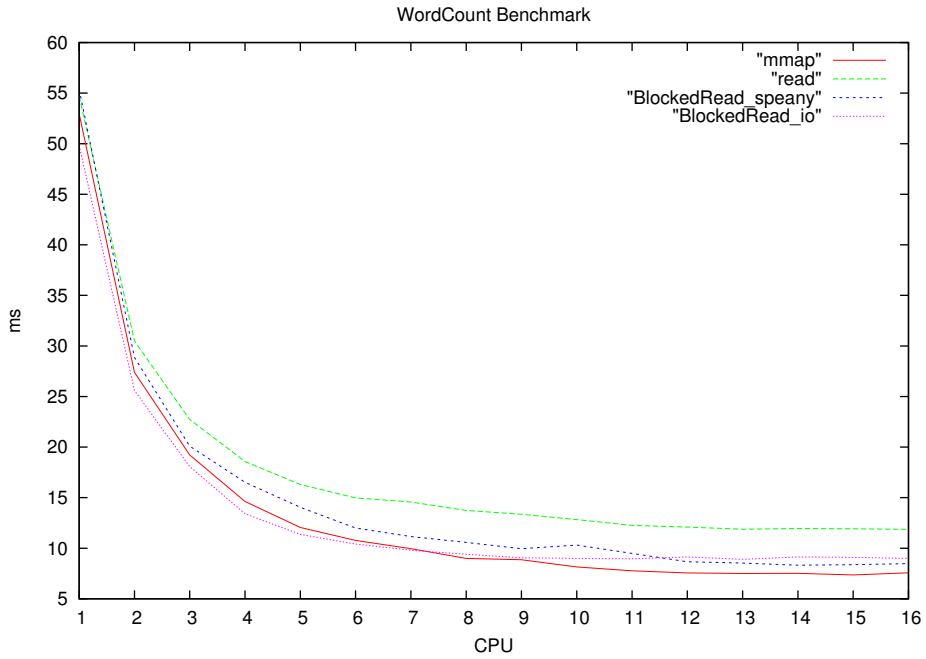


図 8.8: WordCount によるファイル読み込み方式のベンチマーク (MacPro2013)

# 第9章　既存のプログラミングフレームワークとの比較

## 9.1　OpenCL

多次元のデータ構造を扱う計算において高い並列度を保つには、多次元データを分割して並列に実行する機能が必要である。これをデータ並列実行という。 OpenCL はデータ並列実行もサポートしている。OpenCL は次元数に対応する index があり、OpenCL は一つの記述から異なる index を持つ複数の kernel を自動生成する。その添字を global_id と呼ぶ。この時入力されたデータはワークアイテムという処理単位に分割される。

OpenCL はワークアイテムに対してそれぞれを識別する ID ( global_id ) を割り当てる。kernel は get_global_id API によって ID を取得し、取得した ID に対応するデータに対して処理を行い、データ並列を実現する。この ID によって取得してきたワークアイテムをグローバルワークアイテムという。また、ワークアイテムは 3 次元までのデータを渡すことができる。

データ並列による kernel 実行の場合は clEnqueueNDRangeKernel API を使用するが、この関数の引数としてワークアイテムのサイズと次元数を指定することでデータ並列で実行できる。

前節でワークアイテムという処理単位について述べたが、さらに複数個のグローバルワークアイテムを work_group という単位にまとめることができる。work_group 内では同期やローカルメモリの共有が可能となる。

グローバルワークアイテム (ワークアイテム全体) の個数と、ローカルワークアイテム (グループ一つ辺りのアイテム) の個数を指定することでワークアイテムを分割する。なお、このときグローバルワークアイテム数はローカルアイテム数の整数倍でなければ clEnqueueNDRangeKernel API 呼び出しは失敗する。

ローカルアイテム数は 0 を指定することで、コンパイル時に最適化させることができる。したがってローカルアイテムのサイズは 0 を指定するのが一般的である。

なお、 work_group を設定した場合は global_id の他に work_group_id 、local_id がそれぞれの kernel に割り当てられる (図:9.1)。

なお、work_group を設定した場合は global_id の他に work_group_id、local_id がそれぞれの kernel に割り当てられる (図:9.1)。

kernel 側からそれぞれ ID に対応した API を使用して、各 ID を取得する。取得した ID から自分が担当する index を計算して導く。表:9.1 は kernel 側で使用できる、 ID を取得するための API となる。 なお、 local_id 、global_id を取得する API は引数に 0、

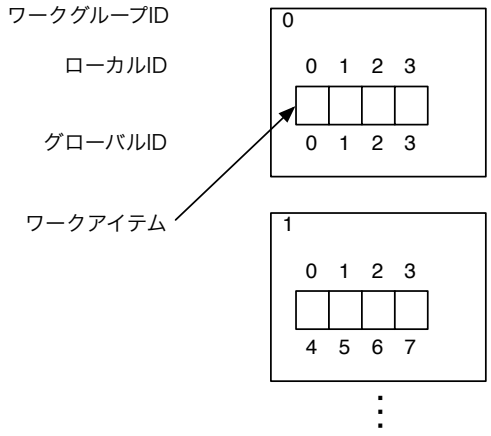


図 9.1: WorkItem ID

| | |
|---|---|
| get_group_id | work_group_id を取得 |
| get_local_id | local_id を取得 |
| get_global_id | global_id を取得 |

表 9.1: kernel で使用する ID 取得の API

1、2 の値を set することができる。id は x, y, z 座標があり、それぞれが 0, 1, 2 に対応している。例えば get_global_id(1) と呼び出した場合は y 座標の、get_global_id(1) と呼び出した場合は z 座標の global_id を取得する。

## 9.2 CUDA

CUDA では OpenCL の WorkItem に相当する単位を thread として定義している。この thread をまとめた単位として block がある。

CUDA でデータ並列による kernel 実行を行う場合、cuLaunchKernelAPI を使用する。この関数は引数として各座標の block 数、各座標の block 1 つ辺りの thread 数を指定することによりデータ並列実行を行う。

cuLaunchKernel で kernel を実行すると各 thread に対して blockID と threadID が割り当てられる。CUDA には OpenCL と異なり、ID を取得する API が存在しない。それに代わり、 kernel に組み込み変数が準備されている。その組み込み変数を参照し、対応するデータに対し処理を行うことでデータ並列実行を実現する。組み込み変数は以下の 3 つである。

- uint3 blockDim
- uint3 blockIdx

- uint3 threadIdx

3 つの組み込み変数はベクター型で、 blockDim.x とすると x 座標の thread 数を参照することができる。同じように blockID 、 threadID の x 座標を参照することができる。blockDim.x * blockIdx.x + threadIdx.x とする事で OpenCL における get_global_id(0) で取得できる ID に相当する値を算出する事ができる。

例としてある kernel で get_global_id(0) の値が 8 の時、CUDA では 図 9.2 のように算出する。

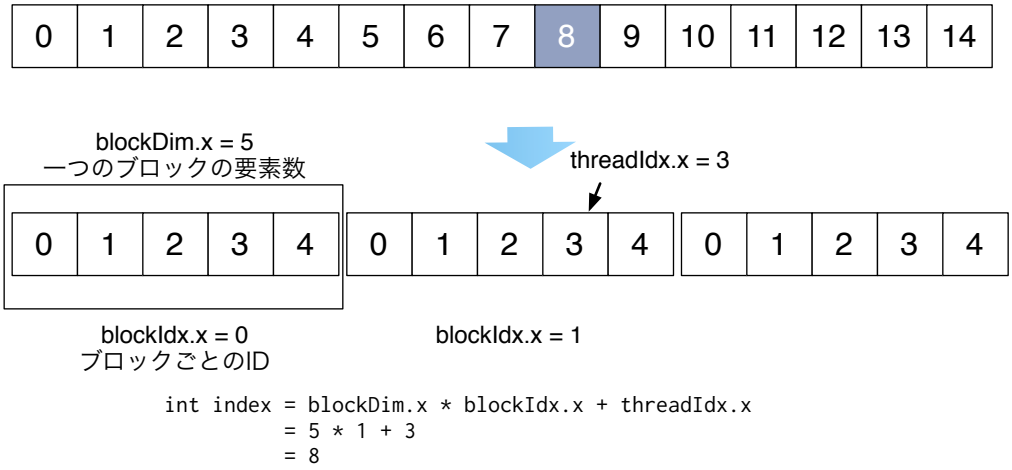


図 9.2: Calculate Index example

## 9.3 StarPU

計算に必要なデータは、StarPU のデータプールに登録されている必要がある。StarPU ではデータを starpu_data_handle という型で登録する。Task はこの handle を参照することで値を参照することができる。

図:9.3 に StarPU におけるデータ並列実行の流れを示す。StarPU では配列の初期化や代入を行った後、starpu_data_register 関数を使って StarPU のデータプールに登録する。

データ並列で実行する場合、更にデータを分割する必要がある。starpu_data_partition 関数を用いる事で分割を行うことができる。分割数を指定することで、データプールに登録したデータを chunk と呼ばれる単位に分割する。starpu_task_submit 関数により chunk を CPU や GPU に割り当てることができる。

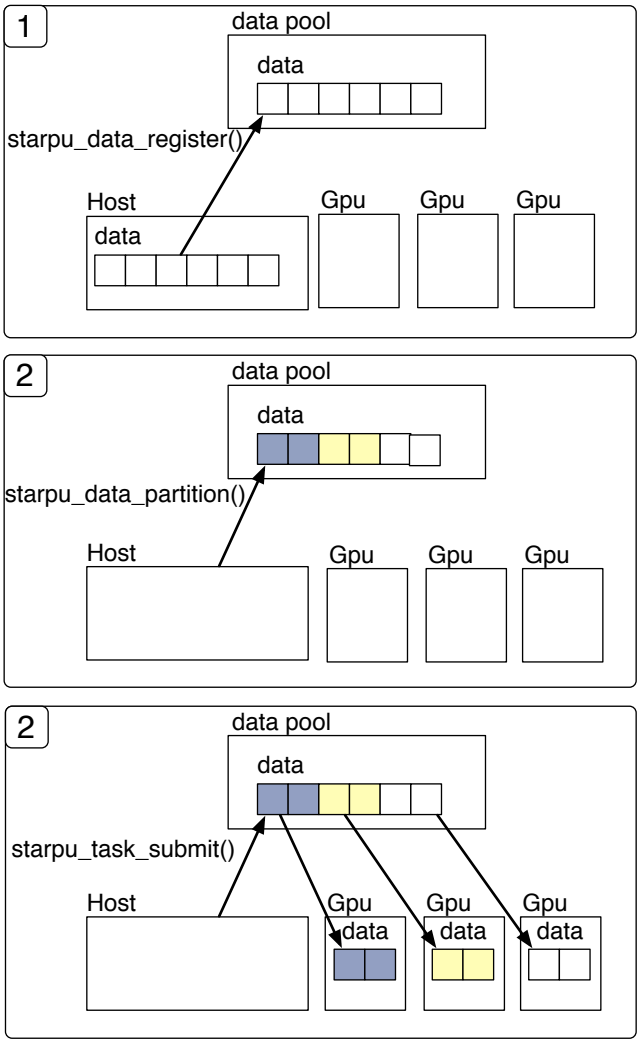


図 9.3: StarPU におけるデータ分割

# 第10章　結論

本研究室で開発している Cerium を用いて、マルチプラットフォーム対応並列プログラミングフレームワークに関する研究を行った。

マルチコア CPU や GPU といったマルチプラットフォームなアーキテクチャ上でリソースを有効活用するには、それぞれのプラットフォームに対して適した形でプログラムが並列に動作できるようチューニングする必要がある。しかしこれらのチューニングは非常に複雑になる。動作させたいプラットフォームに対応した、適切なチューニングを行えるフレームワークが必要である。

Cerium では Scheduler がパイプラインの機構を持っており、Task はパイプラインに沿って実行されている。Scheduler が受信した Task は既に TaskManager が依存関係を解決しているため、実行順序は任意で良い。Cerium はプログラムの様々なレベルでパイプライン処理を行っており、WordCount のようなシンプルな問題でも並列化することで性能向上する事が確認できた。

マルチコア CPU への対応として、 SynchronizedQueue を用いた機構を実装し、並列実行を可能にした。WordCount と Sort による測定の結果、高い並列度を維持出来ていることを確認した。また、より高い性能を持つ計算機で測定したところ、速度の向上を確認できた。

OpenCL と CUDA を用いて GPGPU への対応を行った。WordCount による測定を行い、データ並列実行をサポートすることで OpenCL や CUDA と同等な性能を確認できた。SIMD 型であることやループ命令を苦手とするといった理由から、GPU における演算はデータ並列による実行をサポートしなければ性能が出ない事がわかった。Cerium においてプログラマは Task を記述し、Input データを用意した後はデータ並列用の API で Task を spawn することでデータ並列用の Task を生成する。TaskManager はプログラマが記述した単一の Task を複数生成し、受け取ったデータ (Input) に対しその Task を割り当てる。生成した複数の Task を並列実行する事でデータ並列実行を実現した。また、GPU は SharedMemory でないため、入出力がネックになる場合が多い。それらのオーバヘッドを吸収するために GPU 制御のコマンドはパイプライン形式で実行していくのが望ましい。

更に、 Blocked Read による並列処理向けの I/O の実装を行った。I/O 専用の Thread を追加し、Task が割り込みが起きた際のオーバーヘッドを減らすことに成功した。一般的な file open である mmap と read の測定も行い、BlockedRead を含めたそれぞれの読み込みに対して、どのような状況でより性能を発揮できるのか考察を行った。

## 10.1 今後の課題

Sort、WordCount、FFT と様々なベンチマークを行ったが、それぞれ Task 生成部分の記述が煩雑になるという問題が発生した。例題によって Task を大量に生成する事があるが、生成と実行を一括で行ってしまうとデータがメモリを圧迫してしまい、性能低下に繋がる。そこで Task をある一定の分割数で区切り、徐々に実行していく必要がある。これを Task 間の依存関係で記述するのだが、この記述が原因でコードの量が増えてしまっている。OpenCL では、MemoryBuffer の依存関係により暗黙的に Task 間で依存関係を設定することで Data Dependency を実現している。Cerium でも Data Dependency を導入することにより、プログラマへの負担を減らすことができると考えられる。

Cerium による GPGPU に関して、さらなる性能向上が見込める。今回のベンチマークで、 Cerium による GPGPU と OpenCL 単体による GPGPU で、Cerium の実行時間が OpenCL の 1.01 倍と、ほぼ同じ結果になった。Cerium は Task の入出力と処理の部分をパイプラインにより並列に実行している。本来であれば Cerium がパイプライン処理の分だけ並列度は高くなる。パイプラインが適切に機能していないか、別のオーバーヘッドの存在が考えられる。FFT 以外にも例題を作成し、オーバーヘッド部分の特定を行う必要がある。

マルチコア CPU と GPU において、異なる形式でパイプラインを構築している。マルチコア CPU では SchedTask を用いてパイプラインの構築を行ったが、GPU 側では CommandQueue を 2 つ用意し、それを大きいループで回すことで構築した。可読性の観点から、GPU 側もマルチコア CPU のように SchedTask によるパイプライニングを行うことが望ましい。GPU 用の SchedTask を用意し、read() や write() 部分で OpenCL と CUDA の API を操作する事で実現できる。可読性を向上させることはパイプライン部分のデバッグの行いやすさにも繋がる。

# 謝辞

本研究を行うにあたりご多忙にも関わらず日頃より多くの助言, ご指導をいただきました河野真治准教授に心より感謝いたします.

また、Cerium の先行研究結果がなければ本研究は成り立ち得ませんでした。Cerium の設計や実装に関わった全ての先輩方に感謝いたします。

研究を行うにあたり, 並列計算環境の調整, 意見, 実装, 実験に協力いただいた小久保翔平さん, 古波蔵正隆さん、並びに並列信頼研究室の全てのメンバーに感謝いたします.

最後に, 大学の修士まで支えてくれた家族に深く感謝します.

# 参考文献


[1] Yutaka KINJO. 並列プログラミングフレームワーク cerium の改良. 琉球大学工学部情報工学科平成 24 年度学位論文 (修士), March 2012.

[2] Daichi TOMA and Shinji KONO. Ceriumtask manager におけるマルチコア上での並列実行機構の実装. 第 53 回 プログラミング・シンポジウム, Jan 2012.

[3] Yuhi TOMARI and Shinji KONO. Cerium task manager の gpgpu への対応. 情報処理学会システムソフトウェアとオペレーティング・システム研究会, 2013.

[4] Yuhi TOMARI and Shinji KONO. Cerium task manager における gpu と multicore cpu の同時実行. 第 55 回 プログラミング・シンポジウム, 2014.

[5] Shouhei KOKUBO and Shinji KONO. 並列プログラミングフレームワーク cerium の opencl、cuda への対応. 情報処理学会システムソフトウェアとオペレーティング・システム研究会, 2014.

[6] Masataka KOHAGURA and Shinji KONO. Cerium による並列処理向け i/o の実装. 情報処理学会システムソフトウェアとオペレーティング・システム研究会, 2014.

[7] Yasuhiko OGATA, Toshio ENDO, Naoya MARUYAMA, and Satoshi MATSUOKA. 性能モデルに基づく cpu 及び gpu を併用する効率的な fft ライブラリ. 情報処理学会論文誌コンピューティングシステム, 2008.

[8] Sony Corporation. Cell broadband engine architecture. 2005.

[9] Aaftab Munshi and Khronos OpenCL Working Group. The opencl specification version 1.0. 2007.

[10] Cédric Augonnet, Samuel Thibault, Raymond Namyst, Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, Vol. 23, pp. 187–198, February 2011.

[11] CUDA. https://developer.nvidia.com/category/zone/cuda-zone/.

# 発表履歴

- Cerium Task Manager の GPGPU への対応, 渡真利勇飛、河野真治 (琉球大学) 情報処理学会システムソフトウェアとオペレーティングシステム研究会 (OS), April, 20
- Cerium Task Manager における GPU と Multicore CPU の同時実行 , 渡真利勇飛、河野真治 (琉球大学) 第 55 回プログラミングシンポジウム, January, 2014