Cerium Task Manager における GPU と MultiCore CPU の同時実行

渡真利 勇飛[†1]　小久保　翔平[†2]　河　野　真　治[†2]

Cerium Task Manager は並列プログラミングフレームワークである。今回、MultiCore と GPU 上での同時実行を可能とした。Task を CPU と GPU の両方で動かす場合、どの Task を CPU/GPU に割り当てるかという Scheduling 等の様々な問題がある。FFT を用いて実行速度を測定し、これらの問題の解決方法について考察する

Support GPGPU of Cerium Task Manager

Yuhi TOMARI,[†1] Shohei KOKUBO[†2] and Shinji KONO [†2]

Cerium Task Manager is a parallel programming framework. To achieve good performance in GPGPU using Open CL, various tuning is needed. In particular, it is necessary to implement the dependency of task in Cerium by the function of Open CL. But, to match specialization for OpenCL spoils of flexibility of framework. Balance of flexibility and the performance is considered. We evaluate example FFT.

1. 研究の目的

当研究室では PS3 および Linux 、Mac OS X 上で動く並列プログラミングフレームワーク、Cerium Task Manager[4] の開発・改良を行っている。Cerium は普遍的に演算資源を利用できるフレームワークを目指している。Cell だけでなく、 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 が増えてきた。GPU の普及と高性能化にともない、GPU の演算資源を画像処理以外の目的にも使用する GPGPU (GPU による汎目的計算) が注目されている。Cerium は演算資源として GPU の使用が可能である。

しかし、GPU のみで並列計算を行った場合、Task によっては並列度が出ない場合がある。そこで今回、データ並列による Task の実行と、MultiCore と GPU 上での同時実行を可能とした。Task を CPU と GPU の両方で動かす場合、どの Task を CPU/GPU に割り当てるかという Scheduling 等の問題がある。CPU と GPU では Task の実行速度に差が出ることが多い。Task を CPU と GPU の両方で動かす場合、どの Task を CPU/GPU に割り当てるかという Scheduling 等の問題がある。FFT を用いて実行速度を測定し、これらの問題の解決方法について考察する。

†1 琉球大学理工学研究科情報工学専攻
Interdisciplinary Information Engineering, Graduate School of Engineering and Science, University of the Ryukyus.
†2 琉球大学工学部情報工学科
Information Engineering, University of the Ryukyus.

2. Cerium TaskManager

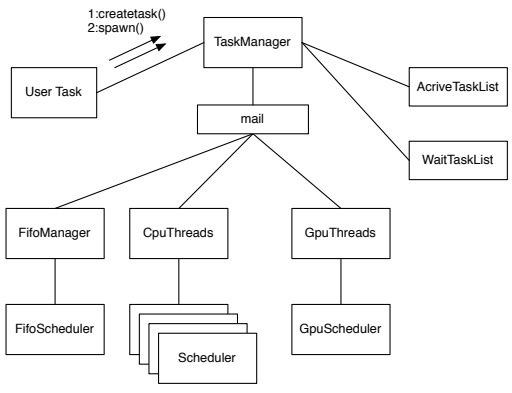

図 1　Task Manager

Cerium Task Manager では、 並列処理を Task 単位で記述する。関数やサブルーチンを Task として扱い、Task には input データ、 output データ及び依存関係を設定する。Cerium Task Manager によってそれらの Task は管理され、実行される。

図 1 は Cerium が Task を作成/実行する場合のクラスの構成となる。user が createtask() を行い、input data や依存関係の設定を行うと TaskManager で Task が生成される。Task 毎に依存関係を表す wait_i と wait_me というリストがあり、依存関係が解消されて実行可能になった Task は ActiveTaskList に移される。さらに、Scheduler に転送しやすい TaskList に変換してから各 Scheduler に転送される。

2.1 Task の生成

以下に Task を生成する例題を示す。input data を二つ用意し、 input 同士を乗算し、 output に格納する multiply という例題となる。

```
void
multiply_init
(HTaskPtr twice, int *i_data, int *o_data) {
    multiply =
        manager->create_task(MULTIPLY_TASK);
    // MULTIPLY_TASK is task id(enum)
    multiply->set_inData(0, i_data1,
                         sizeof(int)*length);
    multiply->set_inData(1, i_data2,
                         sizeof(int)*length);
    multiply->set_outData(0, o_data,
                         sizeof(int)*length);
    multiply->set_param(0, (memaddr)length);
    multiply->set_cpu(SPE_ANY);
    multiply->spawn();
}
```

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 への入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
set_cpu	Task を実行するデバイスの設定
spawn	生成した Task を ActiveTaskList に登録

表 1 Task 生成における API

Task (OpenCL における kernel) の記述は以下のようになる。

```
static int
run(SchedTask *s,void *rbuf, void *wbuf)
{
    float i_data1=(float*)s->get_input(rbuf,0);
    float i_data2=(float*)s->get_input(rbuf,1);
    float o_data=(float*)s->get_output(wbuf,0);
    long length=(long)s->get_param(0);
    for (int i=0;i<length;i++) {
        outdata[i]=indata1[i]*indata2[i];
    }
    return 0;
}
```

3. OpenCL

OpenCL とは、マルチコア CPU と GPU のようなヘテロジニアスな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。このフレームワークを用いて Cerium を GPGPU に対応させる。

get_input	Scheduler から input data を取得
get_output	Scheduler から output data を取得
get_param	set_param した値を取得

表 2 Task 側で使用する API

OpenCL には主に 2 つの仕様がある。

- OpenCL C 言語
- OpenCL ランタイム API

OpenCL C は演算用プロセッサ (本研究では 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。一方で OpenCL ランタイム API は OpenCL C で記述したプログラムを GPU 上で実行させるため、制御用のプロセッサ (本研究では CPU) が利用する API である。

OpenCL では GPU 側を kernel 、制御デバイス側を host として定義する。

3.1 Command Queue

OpenCL では、デバイスの操作に Command Queue を使用する。Command Queue は kernel 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デバイスを指定する。

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。

3.2 メモリアクセス

host 側は主に data を input/output するメモリ資源の確保を行う。GPU のメモリ空間 (図:2) や Cell のメモリ空間 (図:3) はマルチコア CPU (図:4) と違い、共有メモリでないため host と kernel (task) 間で data の共有ができない。アクセスするにはメモリ空間間でコピーしなければならない。

GPGPU では 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。

3.3 データ並列

多次元のデータ構造がある場合に高い並列度を保つには、それを分割して並列に実行する機能が必要である。これを OpenCL ではデータ並列と読んでいる。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

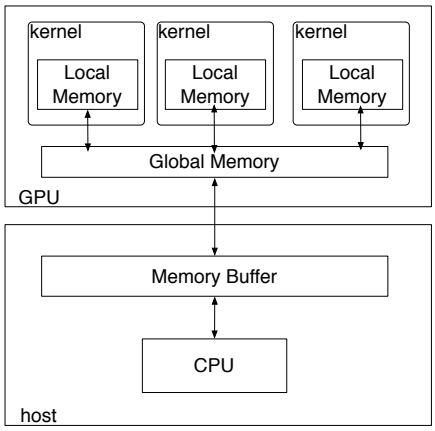

図 2 Gpu Architecture

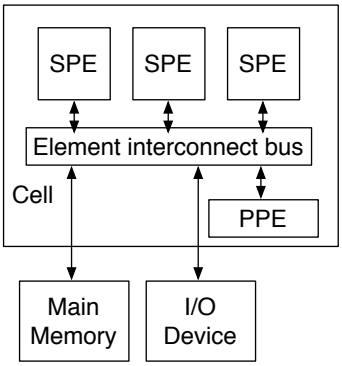

図 3 Cell Architecture

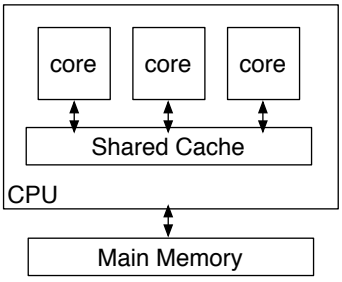

図 4 Cpu Architecture

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。

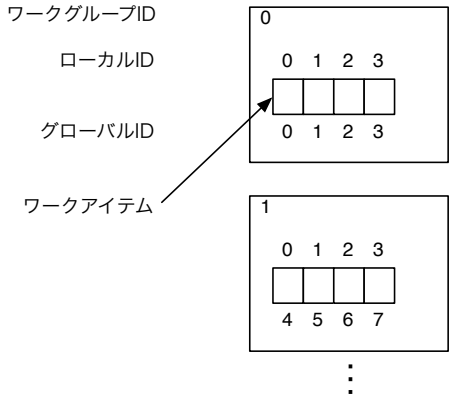

図 5 WorkItem ID

3.4 ワークグループ

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:5)。

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:5)。

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:3.4 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、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

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 3 kernel で使用する ID 取得の API

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。

4. GPU 上での実行の機構

Scheduler と CpuThreads に対応させる形で、GpuScheduler 、GpuThreads を実装した。TaskList からメモリバッファを作成し、EnqueueWriteBuffer, EnqueueTask, EnqueueReadBuffer の順に CommandQueue に enqueue する。Task の投入は CommandQueue を 2 つ用意しパイプライン的に実行を行う。Task の終了は、clWaitForEvent によって検出し、TaskManger 間の通信を担当する同期キューである mail を使って通知する (図:1)。

GpuScheduler 内で platform や device の ID の取得、context の生成、kernel の build と load 、等も行っているため並列計算のみに集中できる。

現在は kernel の記述は、CPU 上で実行する場合と GPU 上で実行する場合のファイルは異なるものを用いる。両者はほとんど同じであるが、若干形式が異なる。これらは将来的には自動変換などを行うのが望ましいと考えられる。

5. 新たに実装した Cerium におけるデータ並列

OpenCL で充分な並列度を得るには、データ並列による実行をサポートした方が良い。Cerium で OpenCL のデータ並列を使うために、iterate という API を用意した。

OpenCL でデータ並列を行う際は、NDRange の引数でワークアイテムのサイズを設定し、以下のように kernel を書けばよい。

```
__kernel void
multi(__global const float *i_data1,
      __global const float *i_data2,
      __global float *o_data)
{
    int i = get_global_id(0);
    o_data[i] = i_data1[i]*i_data2[i];
}
```

kernel を複数生成し、各 kernel は自分が担当する index を get_global_id API で取得し、その部分だけ計算を行う。CPU で実行する場合も GPU 実行時の kernel となるべく近い形式で記述できるようにする。

5.1 データ並列実行の機構

データ並列で実行する場合は spawn API ではなく、iterate API で Task を生成すればよい。Scheduler 内で引数分の Task を生成し、それぞれに自分が担当する index をパラメタとして設定していく。iterate には length を引数として渡し、length の値と渡した length の個数で次元数やワークアイテムのサイズを Scheduler が計算する。CPU 実行時の kernel は以下のように記述する。

```
static int // kernel
run(SchedTask *s,void *rbuf, void *wbuf)
{
    float *indata1,*indata2,*outdata;

    indata1 = (float*)s->get_input(rbuf, 0);
    indata2 = (float*)s->get_input(rbuf, 1);
    outdata = (float*)s->get_output(wbuf, 0);

    long i = (long)s->get_param(0);
    outdata[i]=indata1[i]*indata2[i];
    return 0;
}
```

5.2 データ並列における index 割り当ての実装

Task を生成するとき、dimension とワークアイテムのサイズをもとに各 Task が担当する index を計算し、set_param する。kernel は get_param でその index を取得してデータ並列で実行する。get_param API が OpenCL の get_global_id API に相当する。

例として、cpu 数 4、一次元で 10 個の data にたいしてデータ並列実行を行った場合、各 CPU が担当する index は表:5.2 のようになる。

この例だと各 CPU に対する index の割り当ては、CPU0 は index0、4、8、CPU1 は index1、5、9、CPU2 は index2、6、CPU3 は index3、7 となっている。

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

表 4 data 並列実行時の index の割り当て

この実装により、Cerium でデータ並列の実行が可能になった。並列プログラミングだと、並列化する Task が全部同一であるという事は少なくない。その際、Task を生成する部分をループで回すことなく、簡単な syntax で記述できる。

データ並列で実行する場合は、input と output を各 Task で共有するため、少ないコピーですむ。CPU ならメモリ領域が Task と manager で同じなので、data のコピーで大きいオーバーヘッドにはならない。しかし Cell と GPU はメモリ領域が異なるため、data コピーのオーバーヘッドが大きく、データ並列による高速化が見込める。

6. Benchmark

続いて、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を用いてベンチマークを行った。512*512 の画像を High Pass Filter で変換する例題である。

実験環境

- OS : MacOS X 10.9
- CPU : 2x2.66GHz 6-CoreIntel Xeon
- Memory : 16GB
- Compiler : Apple clang version 4.1 (based on LLVM 3.1svn)
- GPU : AMD ATI Radeon HD 5870 1024MB

6.1 Run Time

	Run Time
1 CPU	453 ms
2 CPU	261 ms
4 CPU	178 ms
8 CPU	117 ms
GPU	94 ms

表 5 Run Time

表 6.1 は CPU,GPU 上，及び CPU + GPU 上で同時実行して比較を行った。1 CPU を利用した場合と比較して，2 CPU では約 1.7 倍，GPU では約 4.8 倍の速度向上が見られる。しかしながら，8 CPU を利用した場合，4 CPU を利用した場合と比較して速度はあがっているが速度上昇率は約 1.5 倍に落ちている。これはアムダールの法則から，並列化率が低いために速度向上が頭打ちになっていると考えられる。

6.2 Busy Time

次に，RDTSC 命令を用い Busy Time の測定を行った。

	Time Stamp	Busy Time
1 CPU	1202282702	451 ms
2 CPU	687813186	258 ms
4 CPU	421398464	158 ms
8 CPU	265192153	99 ms
GPU	3532807	1.3 ms

表 6 Busy Time

CPU を利用した場合，表 6.1 とほぼ同様の結果が得られた。しかしながら，GPU を利用した場合，Busy Time が 1.3 ms なのに対し，表 6.1 の Run Time は 94 ms となっている。この結果から，GPU 上で実行する場合，データの転送がネックになっていることがわかる。

7. 新たに実装した CPU と GPU の同時実行

今回、 CPU と GPU 両方のデバイスで Task を実行させることに成功した。CPU と GPU で同時実行する場合、実行速度には差が出る。GPU はホスト側と PCI バスを通してデータの転送を行う。CPU 側の計算量によってバスの転送を圧迫する事が考えられる。各アーキテクチャの得意とする計算が異なる事からも実行速度に差異が出ることは自明である。これらの事から、 CPU と GPU に対して均等に Task を割り振ると並列度は低くなると予想できる。それを確認するため、ベンチマークを行った。

7.1 Benchmark

	Run Time
GPU & 1 CPU	542 ms
GPU & 2 CPU	357 ms
GPU & 4 CPU	256 ms
GPU & 8 CPU	180 ms

表 7 Run Time for Heterogeneous

セクション 6 で用いた例題を使ってベンチマークを行う。CPU と GPU で Task を同時実行して時間を測定を行った (表 7.1)。なお、 CPU と GPU に交互に Task を割り振っている。GPU & 1 CPU を利用した場合，表 6.1 の 1 CPU を利用した場合と比較して約 0.8 倍の速度低下が見られる。これは各 Task に依存関係があることが原因である。CPU と GPU を交互に動かしているため、デバイス間でのデータの転送が増えたことがネックになっていると考えられる。

7.2 Scheduling

並列度を維持するためにアーキテクチャのコア数によってどちらかに優先して Task を振る必要がある。

Scheduling 手法の提案を行う。まず並列実行する Task を CPU と GPU で事前に一度実行し、実行速度を測定する。Task の実行速度の割合でアーキテクチャに重みをつけて Task を振り分けていく。

片方が先に終わり、もう一方はまだ動いている、という状態が好ましくない。Task が終了するときは CPU も GPU も両方同時に実行終了するように Scheduling する事が望ましい。基本的には GPU の方がコア数が多いので、優先して Task を振ることになる。データの転送がオーバーヘッドになる際に CPU が Task を動かすことで並列度の向上が見込める。

8. ま と め

本研究では Cerium Task Manager をデータ並列による実行に対応させ、更にベンチマークも行った。これにより、大量のデータに対して同一の処理を繰り返し行うという GPU が得意とする処理が可能となった。しかし、 GPU 実行の場合は充分な性能が出ているが、 Multi Core の場合は並列度が充分に出ているとは言えない。データの転送や測定の見直し等、チューニングが必要である。

更に複数ある Task を CPU と GPU、両方のアーキテクチャで実行できるように実装した。しかし、CPU と GPU は同じ性能が出るわけではない。更に GPU はデータの転送がネックになりやすいので、各 Task の

計算量と実行時間が比例しない場合がある。そういった場合に CPU と GPU に均等に Task を割り振ってしまうと、並列度が出ない。そこで CPU と GPU に対して最適に Task を割り振る Scheduling の手法を提案した。

参 考 文 献

1) Akira KAMIZATO.: Cell を用いたゲームフレームワークの提案，琉球大学工学部情報工学科平成 19 年度学位論文 (修士) (2008).
2) Yasuhiko OGATA, Toshio Endo, Naoya MARUYAMA, 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，東京工業大学.
3) Daichi TOMA and Shinji KONO:Cerium Task Manager におけるマルチコア上での並列実行機構の実装 (2012)
4) Wataru MIYAGUNI,Shinji KONO:Cell 用の Fine-grain Task Manager の実装
5) Aaftab Munshi, Khronos OpenCL Working Group:The OpenCL Specification Version 1.0 (2007)
6) Sony Corporation.: *Cell broadband engine architecture* (2005).
7) Keisuke INOUE.: Cell プロセッサ向け実行環境 (SPU Centric Execution Model)，先進的計算基盤システムシンポジウム SACSIS (2006).
8) Shinji KONO.: 検証を自身で表現できるハードウェア、ソフトウェア記述言語 Continuation based C と、その Cell への応用，電子情報通信学会 VLSI 設計技術研究会 (2008).
9) Yoshihiko, FUCHITA.: 分散プログラミングモデル Federated Linda と分散デバッグ開発，琉球大学工学部情報工学科平成 19 年度学位論文 (修士) (2008).