Cerium Task Managerの GPGPU への対応

渡真利 勇飛[†1]　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。Open CL を用いた GPGPU では充分な性能を出すために様々なチューニングが必要になる。特に Cerium における Task ごとの依存関係を Open CL の機能で実装する必要がある。しかし、Open CL に特化してしまうとフレームワークの汎用性を損なってしまう。汎用性と性能向上のバランスを　Sort, Word count, FFT を例題に考察する。

Support GPGPU of Cerium Task Manager

Yuhi TOMARI[†1] 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 spoiles of flexibility of framework. Balance of flexibility and the performance is considered. We evaluate example Sort, Word count, and FFT.

1. 研究の目的

当研究室では Cell 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[1] の開発・改良を行っている。

Cell だけでなく、DSP や 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。そこで、 今回新たに OpenCL を用いて Cerium の演算資源として GPU の使用を可能にした。

GPGPU では通常のマルチ CPU とは異なる並列プログラミングと特別なチューニングが必要になる。Cerium を用いて、その差を吸収し、自動的なチューニングを可能にする。

2. Cerium TaskManager

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。

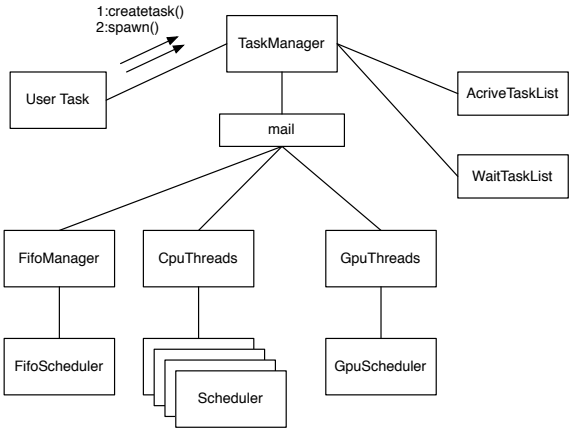

図 1　Task Manager

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

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

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();
}
```

表 1 Task 生成における API

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

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;
}
```

表 2 Task 側で使用する API

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

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。

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。

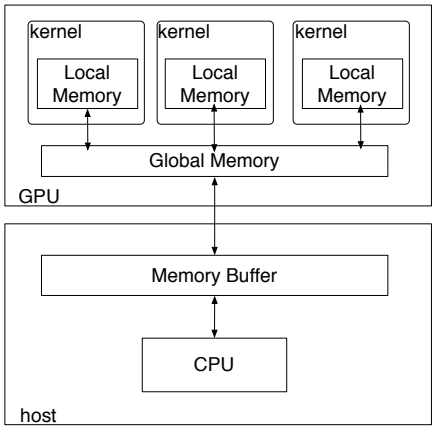

図 2 Gpu Archtecture

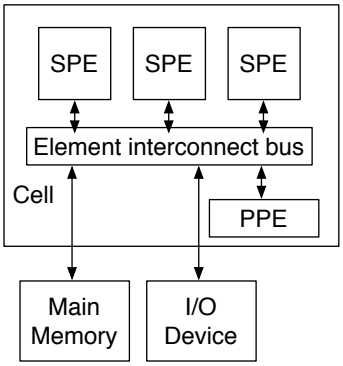

図 3 Cell Archtecture

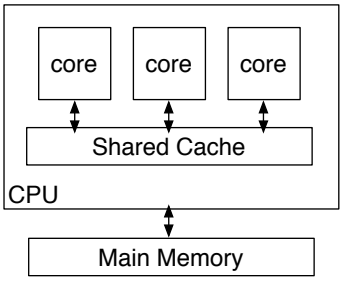

図 4 Cpu Archtecture

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 Open CL 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 とよ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。

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。

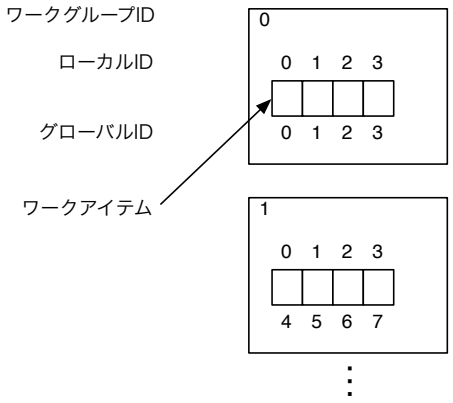

図 5 WorkItem ID

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 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、

表 3 kernel で使用する ID 取得の API

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

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. 新たに実装した 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、clEn-

queuWriteBuffer, clEnqueueTask, clEnqueuReadBuffer の順に CommandQueueu 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. 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、iterator という API を用意した。

ベンチマークをとるために、まずは CPU(many core) 上でデータ並列の機構を実装した。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 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 dimension 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 Cerium での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:4 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。

表 4 data 並列実行時の index の割り当て

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

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 生成部分を何回も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

Bitonic Sort の例題を用いて計測した。入力として 100,000 要素の配列を sort する例題である。これを GPU の比較対象としてマルチコア CPU で同様の例題の計測を行った。

実験環境

- OS : MacOS 10.8.2
- CPU : 2*2.66GHz 6-CoreIntel Xeon
- Memory : 16GB
- Compiler : Apple clang version 4.1 (based on LLVM 3.1svn)
- GPU : AMD ATI Radeon HD 5870 1024MB

この環境で実行したところ、CPU と比べて GPU の実行時間が 100 倍程かかることがわかった。Bitonic sort は data を分割してそれぞれに対して並列に sort(ここでは Quick Sort) をかけて統合を繰り返す sort である。(他に、Word count と FFT の例題を使用している)。

遅い理由としては OpenCL での build 時間が含まれていることが考えられる。また、ND range を実装してないので、並列度が足りてないのも原因の一つだと思われる。

メモリバッファによるコピーも要因の一つである。そこで、一回に送信する data 数 (BLOCK SIZE) を増やしてベンチマークを行った。表 5 が結果である。

表 5 sort による Benchmark の結果

length	100,000
1 CPU	796 ms
2 CPU	439 ms
6 CPU	153 ms
12 CPU	96 ms
24 CPU	89 ms
GPU(改良前)	330752 ms
GPU(改良後)	5306 ms

まだ CPU との性能差は開いているが、10 倍程速度が向上した。Task 並列ではなく、GPU 側でもデータ並列の実行をサポートし、Buffer への Read/Write のパイプラインが上手く動作するように同期機構の見直しを行う事が今後の課題となる。

7. ま と め

本研究では Cerium Task Manager を GPGPU に対応させ、同期機構も実装した。高い並列度を維持するには、GPU での実行時に GPU よりも上のレベル、つまり Task を割り振る段階でも並列実行する必要がある。これに Cerium Task Manager を用いる事で既存の Cerium や、 OpenCL だけで処理を行う場合よりもより高い並列度を実現できる。

さらに Cerium にデータ並列の機構を実装する事でデータや task のコピーを減らし、メモリへの負荷とオーバーヘッドを減らすことに成功した。これは many core でも有効だと考えられる。

Cerium では Task 自体に依存関係を明示的に記述しているが、OpenCL ではメモリバッファの依存関係で暗黙的に指定する方法がある。Cerium 側にもデータ依存関係を導入するのが望ましいと考えられる。

CPU と GPU を同時に使用できるように改良を行うことで、計算資源に GPU を含めることが可能となる。今回明らかになった問題をもとに、まだ GPU は充分な性能とは言えないので、更にチューニングする必要がある。可能な改良としては GPU 側にもデータ並列の機構を実装することと、同期手法の見直しが考えられる。また、高性能の GPU を用いた実験も行う。

参 考 文 献

1) 宮國　渡, 河野真治, 神里　晃, 杉山千秋：Cell 用の Fine-grain Task Manager 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 (2008).
2) 當眞大千, 河野真治：Cerium Task 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 (2012).
3) Aaftab Munshi, Khronos OpenCL Working Group: *The OpenCL Specification Version 1.0* (2007).
4) Khronos OpenCL Working Group: *OpenCL 1.2 Reference Pages* (2012).
5) 北山洋幸：OpenCL 応用 メニーコア CPU & GPGPU 時代の並列処理, カットシステム (2012).
6) 金城裕, 河野真治, 多賀野海人, 小林佑亮 (琉球大学)：ゲームフレームワーク Cerium TaskManager の改良, 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) (2011).