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

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。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,sort,wordcount につ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] 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 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[?] 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。Cerium は演算資源として GPU の使用が可能である。

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, sort, wordcount につ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。

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。

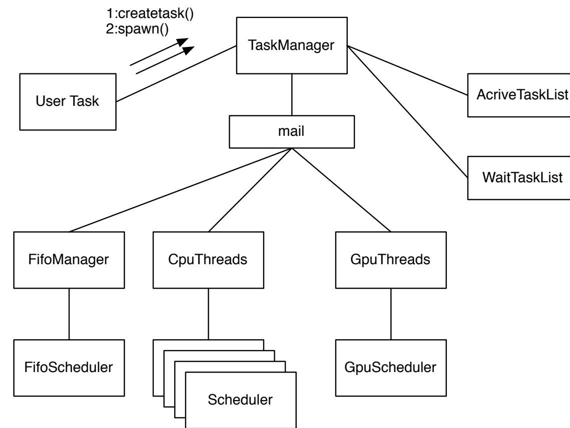

図 1　Task Manager

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

†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.

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();
}
```

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。

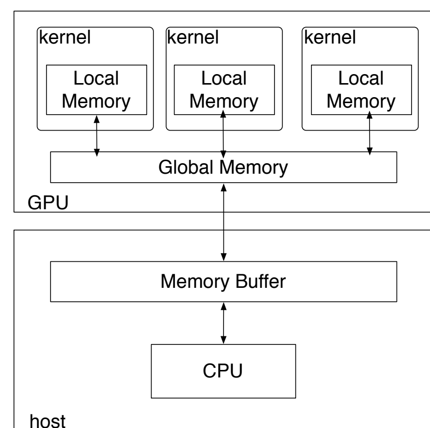

図 2 Gpu Architecture

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。

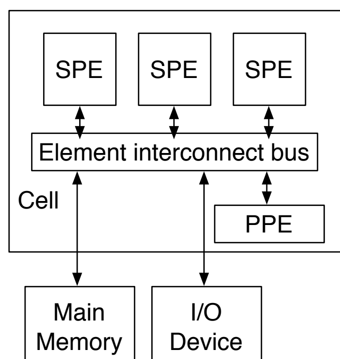

図 3 Cell Architecture

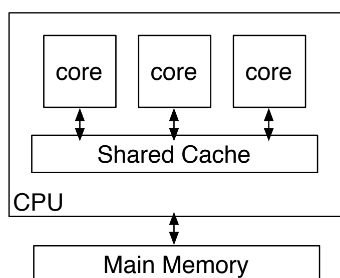

図 4 Cpu Architecture

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。

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:6)。

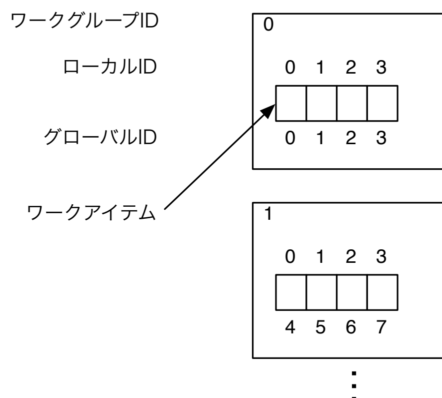

図 5 WorkItem ID

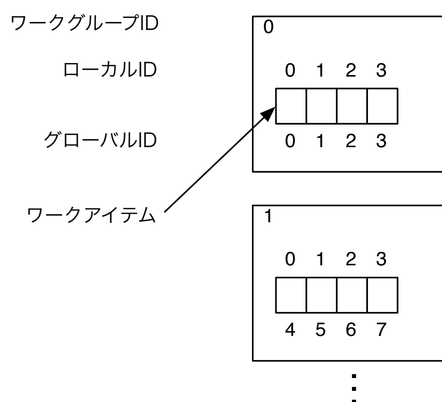

図 6 WorkItem ID

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:6)。

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 、global_id を取得する API は引

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

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. 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、clEnqueueWriteBuffer, clEnqueueTask, clEnqueueReadBuffer 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. 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

6. ま と め