Cerium Task Manager の Multi Platform 対応

平成25年度 卒業論文

琉球大学　工学部　情報工学科

105744J 小久保 翔平
指導教員 河野 真治

目 次

第1章 序論 **1**
1.1 研究背景と目的 . . . 1

第2章 Parallel Computing Platform **2**
2.1 OpenCL . . . 2
2.1.1 Command Queue . . . 2
2.1.2 Memory Access . . . 3
2.1.3 Data Parallel Execution . . . 4
2.1.4 Work Group . . . 5
2.2 CUDA . . . 6
2.2.1 Stream . . . 6
2.2.2 Memory Access . . . 7
2.2.3 Data Parallel Execution . . . 7
2.3 Porting to OpenCL to CUDA . . . 8
2.3.1 Sequential Execution . . . 10
2.3.2 Parallel Execution . . . 16
2.4 StarPU . . . 20

第3章 Cerium **21**
3.1 Cerium TaskManager . . . 21
3.2 Cerium における Task . . . 21
3.3 Task のスケジューリング . . . 23
3.4 Task の入出力 . . . 24
3.5 Cerium におけるデータ並列 . . . 24
3.5.1 データ並列による実行 . . . 25

第4章 Cerium TaskManager の GPGPU への対応 **26**
4.1 GPU Programming の問題点 . . . 26
4.2 OpenCL 実装 . . . 26
4.3 CUDA 実装 . . . 27

第 5 章 Benchmark **28**
5.1 WordCount 28
5.2 FFT 30

第 6 章 結論 **32**
6.1 まとめ 32
6.2 今後の課題 32
6.2.1 CPU と GPU による同時実行のスケジューリング 32
6.2.2 パイプライン実行の改良 32
6.2.3 kernel の自動変換 33

参考文献 **35**

図 目 次

2.1 Gpu Architecture 3
2.2 Cell Architecture 4
2.3 Cpu Architecture 4
2.4 WorkItem ID 5
2.5 Calculate Index 8
2.6 Parallel Execution 16

3.1 Task Manager 22
3.2 Task Scheduler 24

表 目 次

2.1 kernel で使用する ID 取得の API 5
2.2 用語 8
2.3 修飾子 8
2.4 kernel Indexing 9
2.5 Objects 9
2.6 APIs 9

3.1 TaskManager API 21
3.2 Task 側で使用する API 23
3.3 Data 並列実行時の index の割り当て 25

5.1 WordCount 29
5.2 FFT 30

第1章　序論

1.1　研究背景と目的

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[1] の開発・改良を行なっている。

Cell だけでなく、GPU のような異なる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利用する GPGPU(GPU による汎目的計算) が注目されている [2]。

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、高い並列度を出すという研究は様々な分野で行われている。しかし、Cerium Task Manager は特定の計算に限らず、GPU を用いた汎用計算できるフレームワークを目指している。

GPU のような数百個の Core を持つようなプロセッサではデータ並列と呼ばれる実行方法が推奨されている。データ並列とは多次元のデータ構造に対して、それを分割して各要素に対して処理を行うことを指す。また、GPU は CPU とメモリ空間が異なるため、データに直接アクセスすることができない。データにアクセスするためにはメモリ空間ごとコピーする必要がある。このデータ転送部分が大きなオーバーヘッドになる。これらに対応するために今回では OpenCL と CUDA を用いて GPU 上で実行するための機構を実装した。OpenCL と CUDA にはデータ並列を行うための API が存在する。また、自動的にデータ転送をオーバーラップするためパイプライン実行するように Scheduler を設計し、実装した。WordCaunt および FFT を例題として用いて、評価を行なった。

第2章　Parallel Computing Platform

2.1　OpenCL

OpenCL とは、Multi Core 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。
OpenCL には主に 2 つの仕様がある。

- OpenCL C
- OpenCL Runtime 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 Runtime 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) 上で利用する API である。
OpenCL ではオペレーティングシステムなどが処理されるメイン CPU などのことを host、GPGPU を搭載したグラフィックボードなどのことを device と定義している。OpenCL では device に CPU を割り当てることも可能である。OpenCL Application は host 側のプログラムと device 側のプログラムが一体となって動作する。この device 上で動作するプログラムを OpenCL では、特別に kernel と呼ぶ。

2.1.1　Command Queue

OpenCL では、デバイスの操作に Command Queue を使用する。Command Queue は device に OpenCL の Operation を送るために仕組みである。Command Queue は clCreateCommandQueue という OpenCL API に所属するコンテキストと実行対象となる device を指定することで生成される。
Command Queue では kernel の実行、input buffer の読み込み、output buffer への書き込みといった Operation が投入された順序で実行される。Command Queue を作成するとき CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE のプロパティを指定することで Operation を順序を無視して実行することが可能になる。Operation を out of order で

実行する場合、データの依存関係を記述する必要がある。各 Operation には event_wait_list と event を指定することができ、これらを利用してデータの依存関係を記述することができる。しかし、この CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE のプロパティをサポートしている device は少なく、Mac OS X では OS レベルでサポートしていない。パイプライン実行を行うためには kernel の実行やデータ転送を out of order で実行する必要がある。CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE のプロパティが無効の場合、複数の Command Queue を生成し、Command Queue を複数投入することで Operation を out of order で実行することが可能になる。

2.1.2 Memory Access

host 側は主にデータを input/output する Memory の確保を行う。GPU の Memory 空間 (図:2.1) や Cell の Memory 空間 (図:2.2) は Multi Core CPU(図:2.3) とは異なり、Shared Memory ではないため host と device 間でデータの共有ができない。アクセスするには Memory 空間ごとコピーしなければならない。

OpenCL では host 側で Memory Buffer を作成して Memory のコピーを行う。データの読み込みは clEnqueueReadBuffer、書き込みは clEnqueueWriteBuffer という API でそれぞれ行われる。前節で述べた通り、これらの Operation を Command Queue に Enqueue する。そして、データの依存解消が解消されると実行される。

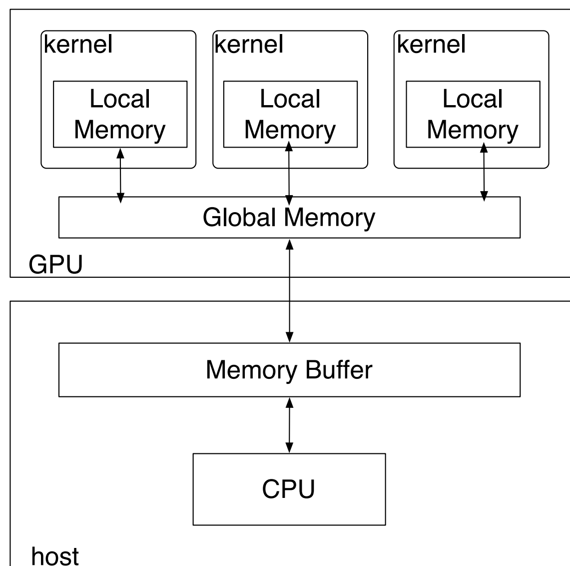
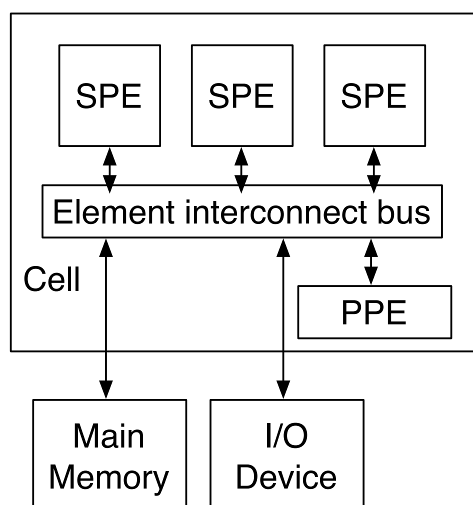


図 2.1: Gpu Architecture



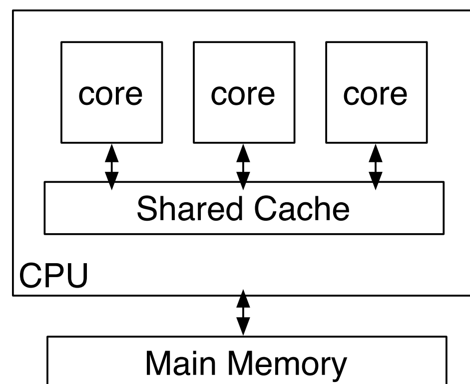


図 2.3: Cpu Architecture

図 2.2: Cell Architecture

2.1.3 Data Parallel Execution

3D グラッフィクのような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 ではデータ並列と呼んでいる。OpenCL では次元数に対応する index があり、OpenCL は一つの記述から index の異なる複数の kernel を自動生成する。その添字を global_id と呼ぶ。このとき入力されたデータは WorkItem という処理単位に分割される。

OpenCL は WorkItem 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 という 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うことでデータ並列を実現する。また、WorkItem は 3 次元まで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、clEnqueueNDRangeKernel API を使用する。この関数の引数として WorkItem の数と次元数を指定することでデータ並列で実行できる。

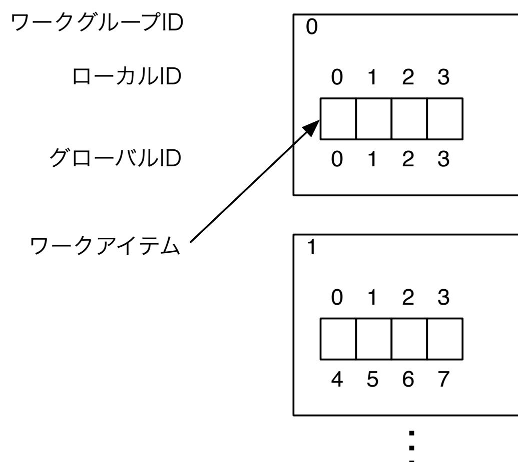


図 2.4: WorkItem ID

2.1.4 Work Group

前節で WorkItem という処理単位について述べたが、さらに複数個の WorkItem を WorkGroup という単位にまとめることができる。WorkGroup 内では同期やローカルメモリの共有が可能になる。

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 の呼び出しは失敗する。

ローカルアイテムに 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、WorkGroup を設定した場合、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:2.4)。

kernel 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:2.1 は kernel で使用できる ID を取得するための API となる。

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 2.1: kernel で使用する ID 取得の API

なお、local_id, global_id を取得する API は引数に 0, 1, 2 の値を指定することができる。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(2) と呼び出した場合は z 座標の global_id を取得する。

2.2 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コンパイラ、ライブラリ、デバッガなどから構成される。プログラム言語である CUDA C は C 言語ベースに拡張を加えたものである。

CUDA には CUDA Runtime API と CUDA Driver API の 2 種類がある。Driver API は Runtime API と比べてプログラマが管理すべきリソースが多い。しかし、Runtime API より柔軟な処理を行うことができる。今回は Driver API を使用して実装した。

CUDA も OpenCL と同様に、制御を行う CPU 側を host、GPU 側を device と定義している。また、device 上で動作するプログラムも OpenCL と同様に kernel と呼ぶ。

2.2.1 Stream

CUDA には OpenCL の Command Queue と似たような仕組みとして Stream がある。Stream は host 側の発行された Operation を一連の動作として device で実行する。Stream 内の 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。異なる Stream での Operation の依存関係が解消され実行可能な場合、Operation を同時に実行することができる。例として、ある Stream に kernel を実行する Operation があり、それとは異なる Stream に依存関係がないデータを転送する Operation があった場合、kernel の実行中にデータ転送を行うことが可能になる。

Stream は cuStreamCreate という Driver API で生成される。OpenCL と異なり、コンテキストと実行対象となる device を指定する必要はないが、コンテキストを作成した Thread と同一の Thread でないと Stream が生成できないという制約がある。引数に Stream を指定しない API はすべて 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処理を行う API を利用する必要がある。

Stream 内の Operation を同期する方法はいくつかある。一つ目は cuStreamSynchronize API を利用した同期方法である。cuStraemSynchronize API の引数に Stream を指定すると、指定した Stream に発行されたすべての Operation が終了するまで host をブロックする。host をブロックすることなく、Stream に発行された Operation が終了したかどうかを調べるには cuStreamQuery API を利用する。二つ目は event を利用した同期

方法である。各 Operation に event_wait_list や event を指定して同期を取る OpenCL とは異なり、CUDA では cuStreamWaitEvent で Stream と event を指定して同期を取る。cuStreamWaitEvent は指定した event が cuEventRecord されるまで指定された Stream をブロックする。event は cuEventCreate で作成され、Operation の直後で cuEventRecord に event を指定することで直前に行う Operation の終了を待つことが可能になる。この event は別の Stream で cuEventRecord されるものでも待つことができる。

2.2.2 Memory Access

CUDA も OpenCL と同様に Shared Memory ではないため host と device 間でデータの共有ができない。アクセスするには Memory 空間ごとコピーする必要がある。CUDA でのデータの読み込みは cuMemcpyHtoD、書き込みは cuMemcpyDtoH という API でそれぞれ行われる。しかし、これらの API は同期的に実行されてしまう。非同期処理にしたい場合、読み込みで cuMemcpyHtoDAsync、書き込みで cuMemcpyDtoHAsync という API をそれぞれ利用することで非同期に行うことができる。

2.2.3 Data Parallel Execution

CUDA では OpenCL の WorkItem に相当する単位を Thread と定義してる。この Thread をまとめたものを block と呼ぶ。CUDA でデータ並列による kernel 実行を行う場合、cuLaunchKernel API を使用する。この関数の引数として各座標の block 数と各座標の block 一つ当たりの thread 数を指定することでデータ並列で実行できる。cuLaunchKernel で kernel を起動すると各 Thread に対して block ID と Thread ID が付与される。CUDA には OpenCL とは異なり、ID を取得する API は存在しない。代わりとして、kernel には組み込み変数が準備されており、それを参照することで対応するデータに対し処理を行うことでデータ並列を実現する。組み込み変数は以下の通りである。

- uint3 blockDim
- uint3 blockIdx
- uint3 threadIdx

各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。blockIdx.x とすると x 座標の block ID が参照でき、threadIdx.x とすると x 座標の thread ID を参照することができる。blockDim.x * blockIdx.x + threadIdx.x を計算すると OpenCL の get_global_id(0) で取得できる ID に相当する ID を得ることができる。例えば、ある kernel で get_global_id(0) の返り値が 13 の場合、 CUDA では図:2.5 のようにすることで ID を算出することができる。

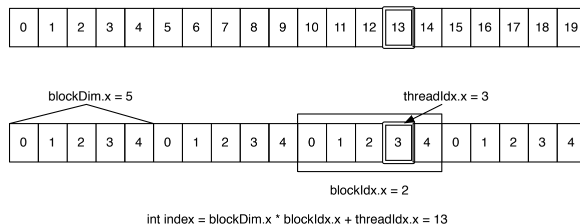


図 2.5: Calculate Index

2.3　Porting to OpenCL to CUDA

本項では OpenCL で記述された Application を CUDA に移植する方法について説明する。以下の表は OpenCL と CUDA の用語および修飾子、ID の参照、Object、API の対応表である。

OpenCL	CUDA
WorkItem	Thread
WorkGroup	Block
Global Memory	Global Memory
Local Memory	Shared Memory
Private Memory	Local Memory

表 2.2: 用語

OpenCL	CUDA
__kernel function	__global__ function
No necessary	__device__ function (not callable from host)
__constant variable	__constant__ variable
__global variable	__device__ variable
__local variable	__shared__ variable

表 2.3: 修飾子

OpenCL	CUDA
get_num_groups()	gridDim
get_local_size()	blockDim
get_group_id()	blockIdx
get_local_id()	threadIdx
get_global_id()	blockDim * blockIdx + threadIdx
get_global_size()	gridDim * blockDim

表 2.4: kernel Indexing

OpenCL	CUDA
cl_device_id	CUdevice
cl_context	CUcontext
cl_program	CUmodule
cl_kernel	CUfunction
cl_mem	CUdeviceptr
cl_command_queue	CUstream(but imperfection)

表 2.5: Objects

OpenCL	CUDA
No required	cuInit()
clGetContextInfo()	cuDeviceGet()
clCreateContext()	cuCtxCreate()
clCreateCommandQueue()	cuStreamCreate()
clCreateProgramWithSource()	cuModuleLoad()
clBuildProgram()	CUDA programs are compiled offline
clCreateKernel()	cuModuleGetFunction()
clCreateBuffer()	cuMemAlloc()
clEnqueueWriteBuffer()	cuMemcpyHtoD()
clEnqueueReadBuffer()	cuMemcpyDtoH()
clEnqueueNDRangeKernel()	cuLaunchKernel()
clSetKernelArg()	Functonality in cuLaunchKernel()
clReleaseMemObj()	cuMemFree()

表 2.6: APIs

2.3.1 Sequential Execution

OpenCL および CUDA で逐次実行するプログラムを例として変換方法を説明する。与えられた二つの input data を乗算し、指定された領域に output する kernel を複数回起動している。

Initialize

```
// initialize(OpenCL)
cl_platform_id platform_id;
cl_uint num_platforms;
cl_device_id device_id;
cl_uint num_devices;
cl_uint ret;
cl_command_queue command_queue;

clGetPlatformIDs(1, &platform_id, &num_platforms);
clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU,
               1, &device_id, &num_devices);
cl_context context = clCreateContext(NULL, 1, &device_id,
                                     NULL, NULL, &ret);
command_queue = clCreateCommandQueue(context, device_id, 0, &ret);
```

OpenCL での初期化は上記のようになる。
OpenCL は様々なメーカー (NVIDIA, AMD など) GPU の対応しているため platform_id を取得し、それをもとに Context を生成する。

```
// initialize(CUDA)
CUdevice device;
CUcontext context;

cuInit(0);
cuDeviceGet(&device, 0);
cuCtxCreate(&context, CU_CTX_SCHED_SPIN, device);
```

CUDA での初期化は上記のようになる。
CUDA は NVIDIA が提供する GPU にのみ対応しているので個別の id を取得する必要がない。また、CUDA には Null Stream と呼ばれるデフォルトの Stream がある。OpenCL の Command Queue のように必ず生成する必要はない。

Load Kernel

```
// load kernel(OpenCL)
const char* filename = "multiply.cl";
const char* functionname = "multiply";

int fp = open(filename, O_RDONLY);

struct stat stats;
fstat(fp,&stats);
off_t size = stats.st_size;

char *kernel_src_str = (char*)alloca(size+1);
size_t kernel_code_size = read(fp, kernel_src_str, size);
close(fp);
kernel_src_str[size] = 0;


cl_program program;
program = clCreateProgramWithSource(context, 1,
                                    (const char **)&kernel_src_str, 0, &ret);
clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
cl_kernel kernel = clCreateKernel(program,functionname, &ret);
```

OpenCL での kernel の読み込みは上記のようになる。

OpenCL ではプログラム内で kernel をコンパイルすることが可能である。専用のコンパイラを必要としないため、様々なプラットフォームへの展開が可能になる。また、Linux 環境では Fixstars 社が提供する foxc、Mac OS X では openclc を利用することで事前にコンパイルすることもできる。

```
// load kernel(CUDA)
CUmodule module;
CUfunction function;

cuModuleLoad(&module, "multiply.ptx");
cuModuleGetFunction(&function, module, "multiply");
```

CUDA での kernel の読み込みは上記のようになる。

CUDA では CUDA に付属されている専用コンパイラ nvcc を使って事前に kernel をコンパイルする必要がある。

Memory Allocate

```
// memory allcate(OpenCL)
cl_mem memA = clCreateBuffer(context, CL_MEM_READ_ONLY,
                             WORKS*sizeof(float), NULL, &ret);
cl_mem memB[num_exec];
cl_mem memOut[num_exec];
for (int i=0;i<num_exec;i++) {
  memB[i] = clCreateBuffer(context, CL_MEM_READ_ONLY,
                           sizeof(float), NULL, &ret);
  memOut[i] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                             WORKS*sizeof(float), NULL, &ret);
}
```

OpenCL での device 上の memory 領域の確保は上記のようになる。
今回のプログラムでは一つ目の input data は共通の値なので、一つ目の input data を格納する領域は一つのみ確保している。二つ目の input data および output data は kernel ごとに異なる値なので、実行する kernel の数だけ領域を確保している。

```
// memory allocate(CUDA)
CUdeviceptr devA;
CUdeviceptr devB[num_exec];
CUdeviceptr devOut[num_exec];

cuMemAlloc(&devA, LENGTH*THREAD*sizeof(float));
for (int i=0;i<num_exec;i++) {
  cuMemAlloc(&devB[i], sizeof(float));
  cuMemAlloc(&devOut[i], LENGTH*THREAD*sizeof(float));
}
```

CUDA での device 上の memory 領域の確保は上記のようになる。
CUDA では memory 領域を確保するとき、context を生成した Thread と同一の Thread である必要があるため引数に context を指定する必要はない。

Data Transfer(host to device)

```
// data transfer(OpenCL)
clEnqueueWriteBuffer(command_queue, memA, CL_TRUE, 0,
                     WORKS*sizeof(float), A, 0, NULL, NULL);

for (int i = 0;i<num_exec;i++){
  B[i] = (float)(i+1);
  clEnqueueWriteBuffer(command_queue, memB[i], CL_TRUE, 0,
  sizeof(float), &B[i], 0, NULL, NULL);
}
```

OpenCL での host から device へのデータ転送は上記のようになる。
clEnqueueWriteBuffer の第三引数に CL_TRUE を指定することで、データ転送が完了するまで host をブロックする同期的な Operation となる。

```
// data transfer(CUDA)
cuMemcpyHtoD(devA, A, LENGTH*THREAD*sizeof(float));

for (int i=0;i<num_exec;i++) {
  B[i] = (float)(i+1);
  cuMemcpyHtoD(devB[i], &B[i], sizeof(float));
}
```

CUDA での host から device へのデータ転送は上記のようになる。
cuMemcpyHtoD はデータ転送が完了するまで host をブロックする同期的な Operation である。

Launch Kernel

```
// launch kernel(OpenCL)
clSetKernelArg(kernel, 0, sizeof(cl_mem), &memA);
for (int i = 0;i<num_exec;i++){
  clSetKernelArg(kernel, 1, sizeof(cl_mem), &memB[i]);
  clSetKernelArg(kernel, 2, sizeof(cl_mem), &memOut[i]);
  clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
                         &num_work, NULL, 0, NULL, NULL);
  }
```

OpenCL での kernel の実行は上記のようになる。
clSetKernelArg で kernel の引数に対して値を渡すことができる。
clEnqueueNDRangeKernel の第三引数に WorkItem の次元数を指定し、第五引数に WorkItem の数を指定することでデータ並列で実行される。

```
// launch kernel(CUDA)
for (int i=0;i<num_exec;i++) {
  void* args[] = {&devA, &devB[i], &devOut[i]};
  cuLaunchKernel(function,
                 LENGTH, 1, 1,
                 THREAD, 1, 1,
                 0, 0, args, NULL);
}
```

CUDA での kernel の実行は上記のようになる。
clLaunchKernel の第二、第三、第四引数に各次元のブロック数を指定し、第五、第六、第七引数に各次元の Thread 数を指定することでデータ並列で実行される。今回は、LENGTH 個のブロックがあり、各ブロックに THREAD 個の Thread が起動する。また、CUDA には clSetKernelArg に相当する API はない。clLaunchKernel の第十引数に指定することで kernel の引数に対して値を渡すことができる。

Data Transfer(device to host)

```
// data transfer(OpenCL)
for (int i=0;i<num_exec;i++)
  clEnqueueReadBuffer(command_queue, memOut[i], CL_TRUE, 0,
                      WORKS*sizeof(float), result[i], 0, NULL, NULL);
```

OpenCL での device から host へのデータ転送は上記のようになる。
clEnqueueWriteBuffer と同様に第三引数に CL_TRUE を指定することでデータ転送が完了するまで host をブロックする同期的な Operation となる。

```
// data transfer(CUDA)
for (int i=0;i<num_exec;i++)
  cuMemcpyDtoH(result[i], devOut[i], LENGTH*THREAD*sizeof(float));
```

CUDA での device から host へのデータ転送は上記のようになる。
cuMemcpyDtoH は cuMemcpyHtoD と同様にデータ転送が完了するまで host をブロックする同期的な Operation である。

Memory Release

```
// memory release(OpenCL)
clReleaseMemObject(memA);
for (int i=0;i<num_exec;i++) {
  clReleaseMemObject(memB[i]);
  clReleaseMemObject(memOut[i]);
}
clReleaseCommandQueue(command_queue);
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseContext(context);
```

OpenCL での memory 領域の開放やオブジェクトの開放は上記のようになる。

```
// memory release(CUDA)
cuMemFree(devA);
for (int i=0;i<num_exec;i++) {
  cuMemFree(devB[i]);
  cuMemFree(devOut[i]);
}
cuModuleUnload(module);
cuCtxDestroy(context);
```

CUDA での memory 領域の開放やオブジェクトの開放は上記のようになる。

Kenrel

```
// kernel(OpenCL)
__kernel void
multiply(__global float* A, __global float* B, __global float* C) {
  long index = (long)get_global_id(0);
  C[index] = A[index] * B[0];
}
```

OpenCL での kernel の記述は上記のようになる。get_global_id で index を取得し、担当する部分の演算を行っている。

```
// kernel(CUDA)
extern "C" {
  __global__ void multiply(float* A, float* B, float* C) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    C[index] = A[index] * B[0];
  }
}
```

CUDA での kernel の記述は上記のようになる。組み込み変数である blockIdx, blockDim, threadIdx から index を算出し、担当する部分の演算を行なっている。

2.3.2 Parallel Execution

項:2.3.1 で例として用いたプログラムは一つ目の input data 以外に依存関係が存在しない。つまり、OpenCL では Command Queue、CUDA では Stream を複数生成し、Operation を並列に実行して良い。Operation を並列に実行することでデータ転送をオーバーラップすることができる。このプログラムを kernel を実行する数だけ Command Queue および Stream を生成し、並列実行されるように変更する。変更がある部分のみ記載する。それ以外の部分は項:2.3.1 と同じである。実行の様子は図:2.6 のようになる。

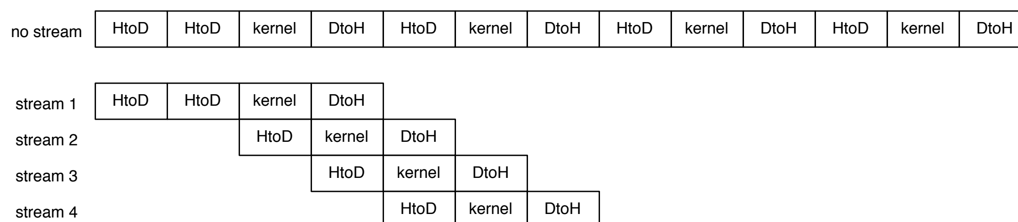


図 2.6: Parallel Execution

Initialize

```
// initialize(OpenCL)
cl_command_queue command_queue[num_exec];

for (int i=0;i<num_exec;i++)
  command_queue[i] = clCreateCommandQueue(context, device_id, 0, &ret);
```

OpenCL では上記のように変更する。
Command Queue を配列として宣言し、要素の数だけ Command Queue を生成する。

```
// initialize(CUDA)
CUstream stream[num_exec];

for (int i=0;i<num_exec;i++)
  cuStreamCreate(&stream[i], 0);
```

CUDA では上記のように変更する。
Stream を生成するとき、OpenCL と同じように context と device_id を渡す必要はない。これは項:2.2.1 で述べたとおり、コンテキストを作成した Thread と同一の Thread でないと Stream を生成できないからである。
OpenCL と同様に Stream を配列として宣言し、要素の数だけ Stream を生成する。

Data Transfer(host to device)

```
// data transfer(OpenCL)

for (int i = 0;i<num_exec;i++){
  B[i] = (float)(i+1);
  clEnqueueWriteBuffer(command_queue[i], memB[i], CL_FALSE, 0,
  sizeof(float), &B[i], 0, NULL, NULL);
}
```

OpenCL では上記のように変更する。
clEnqueueWriteBuffer の第三引数に CL_FALSE を指定することで、非同期な Operation として実行できる。つまり、Command Queue に Operation を投入し、投入した Operation の完了を待たずに host に制御を返す。

```
// data transfer(CUDA)

for (int i=0;i<num_exec;i++) {
  B[i] = (float)(i+1);
  cuMemcpyHtoDAsync(devB[i], &B[i], sizeof(float), stream[i]);
}
```

CUDA では上記のように変更する。
cuMemcpyHtoDAsync は非同期な Operation として実行される。第四引数に関連する Stream を指定することができる。

Launch Kernel

```
// launch kernel(OpenCL)
for (int i = 0;i<num_exec;i++){
  clEnqueueNDRangeKernel(command_queue[i], kernel, 1, NULL,
                         &num_work, NULL, 0, NULL, NULL);
  }
```

OpenCL では上記のように変更する。
関連する Command Queue を指定するだけで良い。

```
// launch kernel(CUDA)
for (int i=0;i<num_exec;i++) {
  cuLaunchKernel(function,
                 LENGTH, 1, 1,
                 THREAD, 1, 1,
                 0, stream[i], args, NULL);
}
```

CUDA では上記のように変更する。
第九引数に関連する Stream を指定するだけで良い。

Data Transfer(device to host)

```
// data transfer(OpenCL)
for (int i=0;i<num_exec;i++)
  clEnqueueReadBuffer(command_queue[i], memOut[i], CL_FALSE, 0,
                      WORKS*sizeof(float), result[i], 0, NULL, NULL);
```

OpenCL では上記のように変更する。
clEnqueueWriteBuffer と同様に第三引数に CL_FALSE を指定することで非同期な Operation となる。

```
// data transfer(CUDA)
for (int i=0;i<num_exec;i++)
  cuMemcpyDtoHAsync(result[i], devOut[i], LENGTH*THREAD*sizeof(float), stream[i]);
```

CUDA では上記のように変更する。
cuMemcpyDtoHAscyn は cuMemcpyHtoDAsync と同様に非同期な Operation として実行される。

Wait for Exec

host をブロックしない非同期な Operation はどのタイミングで実行されるかわからない。全ての Operation が完了していないのに memory やオブジェクトを開放してしまう場合がある。その場合、処理が正しく行われない。この問題を回避するために、全ての Operation を待つという処理を host 側に記述する必要がある。

```
// wait for exec(OpenCL)
for (int i=0;i<num_exec;i++)
  clFinish(command_queue[i]);
```

OpenCL での投入された Operation の完了を待つのは上記のようになる。
clFinish に完了を待ちたい Command Queue を指定することで、投入された Operation が全て完了するのを待つことができる。

```
// wait for exec(CUDA)
for (int i=0;i<num_exec;i++)
    cuStreamSynchronize(stream[i]);
```

CUDA での投入された Operation の完了を待つのは上記のようになる。

OpenCL と同様に cuStreamSynchronize に完了を待ちたい Steram を指定することで、Operation の終了を待つことができる。cuStreamSynchronize は指定された Stream が完了するまで host をブロックする。完了したかどうかだけを知りたい場合は、cuStreamQuery を利用する。

2.4 StarPU

ヘテロジニアス環境下でのタスクベースなプログラミングをサポートするものとして INRIA から提供されている StarPU というものがある。StarPU は CPU, CUDA device, OpenCL device など複数のアーキテクチャに対応している。StarPU には Codelet と呼ばれる device に Task を渡す仕組みがある。Task は非同期に実行され、host をブロックしない。しかし、Task の依存関係やスケジューリングをプログラマが記述する必要がある。この記述は非常に煩雑でプログラマに大きな負担をかけることになる。Cerium ではそれらを自動化することでプログラマの負担を軽減したい。

第3章　Cerium

3.1　Cerium TaskManager

TaskManager は、Task と呼ばれる分割されたプログラムを管理する。Task の単位はサブルーチンまたは関数として、Task 同士の依存関係を考慮しながら実行される。TaskManager の API を表:3.1 に示す。

create_task	Task を生成する
allocate	環境のアライメントに考慮した allocator
set_inData	Task への入力データのアドレスを追加
set_outData	Task からのデータ出力先アドレスを追加
set_param	Task のパラメータ (32 bits)
wait_for	Task の依存関係の考慮
set_cpu	Task を実行する Device の設定
spawn	Task を Queue に登録する
iterate	データ並列で実行する Task として Queue に登録する

表 3.1: TaskManager API

3.2　Cerium における Task

Task は TaskManager を使って生成する。Task を生成すると際に、以下のような要素を設定することができる。

- input data
- output data
- parameter
- cpu type
- dependency

input, output, parameter は関数でいうところの引数に相当する。cpu type は Task がどのような Device の組み合わせで実行されるかを示す。dependency は他の Task との依存関係を示している。Task の終了が通知され、その通知に従って依存関係を処理していく。例えば、Task B には Task A との依存関係がある場合、Task B は Wait Queue に投入され依存関係が解消されるのを待つ。Task A は Active Queue に投入され、cpu type によって示された Device に転送され実行される。Task A の処理が終了すると Mail で Task B へ通知される。Task B はその通知を受けると Active Queue に投入され、Task A と同様に指定された Device 上で実行される。図:3.1 は Cerium が Task を生成/実行する場合のクラスの構成である。

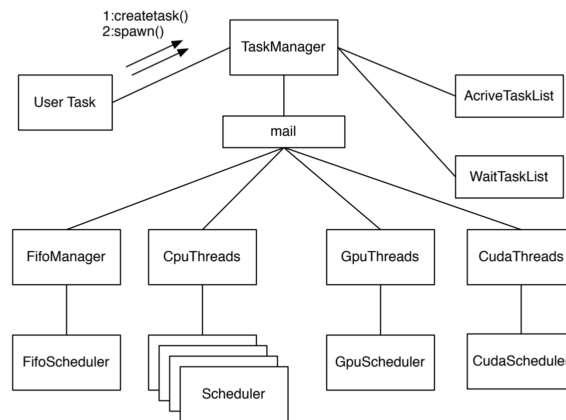


図 3.1: Task Manager

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 という例題である。

```
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();
}
```

Task の記述は以下のようになる。表:3.2 は Task 側で使用する API である。

```
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;
}
```

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 3.2: Task 側で使用する API

3.3 Task のスケジューリング

Device は、Task を一つずつ受け取るのではなく、ある程度まとめて受け取る。それを TaskList と呼んでいる。TaskList は各 Device 用に TaskManager 側で生成される。受け取った TaskList に沿って Task をパイプラインで実行していく。Task 毎に実行完了の Mail を送る。TaskList の Task をすべて実行すると、次の TaskList を要求する Mail を送る。実行の様子は図:3.2 のようになる。

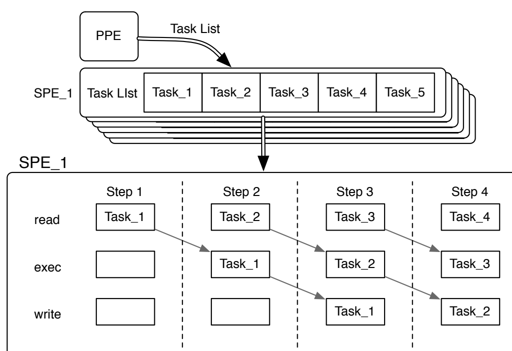


図 3.2: Task Scheduler

3.4 Task の入出力

Task の入出力の API として、set_inData, set_outData がある。

set_inData(index, addr, size) は、データを受け取る buffer の配列番号とデータのアドレス、そのデータのサイズを引数として入力する。このデータは DMA 転送されるため、addr は 16 byte alignment が取れており、size は 16 byte の倍数である必要がある。

set_param(index, param) は、データを受け取る buffer の配列番号と 32bit のデータを渡す。set_inData で渡すには小さいデータを送るのに適している。param はアドレスとしてではなく、値を Task オブジェクトが直接持っているので、 DMA 転送は行わない。

set_outData(index, addr, size) は、Task のデータの出力先を指定する。使用方法は set_inData と同じで、alignment, byte 数に気をつける必要がある。

3.5 Cerium におけるデータ並列

Cerium でデータ並列を利用するために、OpenCL の API に合わせた iterate という API を用意した。iterate は length を引数として受け取り、Scheduler で length の値と受け取った引数の個数を次元数として Task 数を計算し、データ並列として実行する。

例として、CPU 数 4、一次元で 10 個のデータに対してデータ並列実行を行なった場合、各 CPU が担当する index は表:3.5 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 index 0,4,8、CPU1 は index 1,5,9、CPU2 は index 2,6、CPU3 は index 3,7 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 3.3: Data 並列実行時の index の割り当て

3.5.1 データ並列による実行

Scheduler で、データ並列による実行を行う Task を検出し、各 CPU で Task を実行する、各 CPU が担当する index は SchedTask に格納してある。Task は以下のように記述する。

```
static int // Task
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);

    uisigned long i = s->x;
    outdata[i]=indata1[i]*indata2[i];
    return 0;
}
```

並列プログラミングでは、並列化する Task が全部同一であるということは少なくない。itrerate を実装したことで、Task を生成する部分をループで回す必要はなくなり、OpenCL と同様に一つの記述で異なる index を持つ Task を CPU 上で実行することができるようになった。

第4章 Cerium TaskManager の GPGPU への対応

本章では、まずはじめに GPU Programming の問題点について述べ、Cerium TaskManager への実装でそれをどう改善したのかについて説明する。

4.1 GPU Programming の問題点

まず Many Core に対するプログラミングと同様にプログラムの性能を向上させるためにプログラム全体を対象にした並列度を高くしなければならない。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はデータ並列で処理を行うことで CPU 単体で全てを計算させる場合より圧倒的に高速で演算することができる。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を行うだけでプログラムの性能は十分に向上することになる。しかし、多くのプログラムはその限りではない。GPU 上での実行においてネックになる部分はデータ転送である。項:2.1.2 で述べたとおり、Shared Memory ではないため host と device でデータの共有ができない。データを参照するためには Memory 空間のコピーを行うしかない。これが大きなオーバーヘッドになるので、パイプラインで実行することでデータ転送をオーバーラップする必要がある。しかし、パイプライン実行の記述は非常に煩雑で、また、パイプラインで実行されていることを証明することは難しくプログラマに大きな負担をかけることになる。今回、これら問題を解決するため、Cerium TaskManager から受け取った TaskList に従って自動でパイプライン実行するように OpenCL および CUDA で Scheduler を実装した。

4.2 OpenCL 実装

Scheduler と CpuThreads に対応させる形で、GpuScheduler, GpuThreads を実装した。TaskList から memory 領域を確保し、input data の数だけ clEnqueueWriteBuffer、実行する形式 (タスク並列、データ並列) に合わせて clEnqueueTask または clEnqueueNDRangeKernel、output data の数だけ clEnqueueReadBuffer の順序で Command Queue に Operation を発行する。Operation を投入する Command Queue は任意の数に変更す

ることができる。デフォルトでは8段のパイプラインとして実行を行う。Operation 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:3.1)。

GpuScheduler 内で platform_id や device_id の取得、Context, Command Queue の生成、device 上での memory 領域の確保、kernel の build と load 等を行なっている。

現在 kernel の記述は、CPU 上で実行する場合と GPU 上で実行する場合はほとんど同じであるが、修飾子など若干形式が異なる。これらは将来的に自動変換など行うのが望ましい。

4.3 CUDA 実装

CUDA の場合も Scheduler と CpuThreads に対応させる形で、CudaScheduler, CudaThreads を実装した。OpenCL での実装と同様に、TaskList から memory 領域を確保し、input data の数だけ cuMemcpyHtoDAsync、実行する形式 (タスク並列、データ並列) に合わせて cuLaunckKernel のブロック数およびスレッド数、次元数を変更し、output data の数だけ cuMemcpyDtoHAsync の順序で Stream に Opration を発行する。Operation を投入する Stream は任意の数に変更することができ、OpenCL での実装と同様に、デフォルトでは8段のパイプラインとして実行を行う。Operation の終了は、cuStreamQuery によって検出し、終了を待つ必要があるときは cuStreamSynchronize で Stream に発行された全ての Operation の完了を待つ。TaskManager 間の通信は OpenCL での実装と同様に、同期キューである mail を使って通知する (図:3.1)。

CUDA 実装も同様に CudaScheduler 内で Context, Stream の生成、device 上での memory 領域の確保、kernel の load 等を行なっている。Cerium を用いて GPU Programming を行う場合、OpenCL, CUDA のどちらも host 側のコードは同等の記述ができ、プログラマは並列計算のみに集中できる。

kernel の記述は、OpenCL と同様にほとんど同じであるが、修飾子など若干形式が異なる。こちらも CPU 版の kernel から OpenCL 版の kernel などに自動で変換されるようにするのが望ましい。

第5章　Benchmark

本章では、WordCount, FFT を例題として使用し、本研究で実装した GpuScheduler および CudaScheduler の測定を行う。
実験環境

- OS : MacOS 10.9.1
- CPU : 2*2.66GHz 6-Core Intel Xeon
- GPU : NVIDIA Quadro K5000 4096MB
- Memory : 16GB 1333MHz DDR3
- Compiler : Apple LLVM version 5.0 (clang-500.2.79) (based on LLVM 3.3svn)

5.1　WordCount

WordCount の Task の構成は以下のようになる。

- WordCountTask
- DataParallelWordCountTask
- PrintTask

WordCountTask は、input された data を word count し、単語数と行数を output に指定された data 領域に書き込む Task である。word count 対象として入力されたファイルは、mmap を用いて memory に展開され、その後データを 16kbyte の大きさに分割して、Task に割り当てられる。分割されたデータが送られてくるため、分割された前後のテキストがどうなっているかはわからない。そのため担当範囲であるデータの先頭と末尾のパラメータを単語数と行数の他に付け加える。後にそのデータを他の word count 結果と照らし合わせ、分割されたテキストを正しく整合する。

DataParallelWordCountTask も WordCountTask と同様に input された data を word count し、単語数と行数を output に指定された data 領域に書き込む Task である。WordCountTask と異なる点は、送られてくるデータが分割されてなく、Task 内で index から担当する範囲を算出し、output に書き込む点である。この Task はデータ並列で実行される。GPU 上で実行する場合のデータ転送の回数を減らすことでオーバーヘッドを削減している。

PrintTask は WordCountTask または DataPrallelWordCountTask 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。WordCountTask または DataParallelWordCountTask の終了を待ち、動作する。

今回は 100MB のテキストファイルに対して wordcount を行なった。表:5.1 はその実行時間である。

	Run Time
1 CPU	0.69s
2 CPU	0.35s
4 CPU	0.18s
8 CPU	0.09s
12 CPU	0.07s
OpenCL(no pipeline)	48.32s
OpenCL(pipeline)	46.74s
OpenCL Data Parallel	0.50s
CUDA(no pipeline)	55.71s
CUDA(pipeline)	53.30s
CUDA Data Parallel	0.73s

表 5.1: WordCount

パイプラインありの方が OpenCL では 1.03 倍、CUDA では 1.04 倍の性能向上が見られた。しかし、期待するほどの効果が出ていない。これは綺麗なパイプラインで実行されていないことが考えられる。パイプラインで効果を出すためにはデータ転送と Task の実行が同じ時間で終了することが必要になる。一方、データ並列による実行ではタスク並列の実行に比べ、OpenCL は 93 倍、CUDA では 73 倍の性能向上が見られた。

5.2 FFT

次に、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利用し測定する。FFT の Task は構成は以下のようになる。

- spinFact
- bitReverse
- norm
- butterfly
- transpose
- highPassFilter

spinFact はバタフライ演算で乗算要素として用いられる回転因子を計算する Task である。この値はバタフライ演算するたびに毎回計算してもよいが、サンプル数が同じであれば同じ回転因子を使うのであらかじめ計算しておけばよい。

bitReverse はバタフライ演算の前に各要素をビット逆順に並び替える Task である。データ並列によって実行したいので、index のビットを実際に反転させることで入れ替え先の index を生成します。

norm はデータを使いやすくするため正規化する Task である。

butterfly はペアリングされた一つのバタフライ演算を計算する Task である。

transpose は行と列の index を入れ替える Task である。

highPassFilter は高周波数領域のみを通すフィルタをかける Task である。

今回、使用する画像のサイズは 512*512 で、それに対して High Pass Filter をかけて変換する。表:5.2 はその実行時間である。

	Run Time
1 CPU	0.45s
2 CPU	0.26s
4 CPU	0.17s
8 CPU	0.11s
12 CPU	0.07s
OpenCL	0.09s
CUDA	0.21s

表 5.2: FFT

OpenCL では 1 CPU の 5 倍、CUDA では 2.1 倍の性能向上が見られた。しかし、OpenCL と CUDA で差がある。これは、GpuScheduler に対して CudaScheduler の同期がまだ不十分であることが考えられる。また、OpenCL の kernel では cl_float2 というベクター型を用いている。しかし、CUDA では cl_float2 を float に変換して演算している。OpenCL ではベクターの演算なので、その部分に最適化がかかっている可能性もある。

第6章　結論

6.1　まとめ

本研究では並列プログラミングフレームワーク Cerium の改良を行なった。Ceirum Task Manager 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である。CPU/Cell だけでなく、GPU も計算資源として利用できる。今回、新たに iterate API を実装し、Multi Core CPU でのデータ並列での実行を可能にした。OpenCL による GPGPU に加え、CUDA による GPGPU を可能にした。また、host と device 間でのデータ転送をオーバーラップするため GpuSschduler および CudaScheduler にパイプライン実行機構を実装した。WordCount および FFT を例題として用いて、実装した GpuScheduler および CudaScheduler の評価を行なった。これらの改良によって、パイプライン実行を行うと OpenCL で 1.03 倍、CUDE で 1.04 倍の性能向上に成功した。

6.2　今後の課題

6.2.1　CPU と GPU による同時実行のスケジューリング

現在、Cerium では CPU と GPU の同時実行に対応している。しかし、現在 Task を CPU, GPU に対し交互に割り振っているため CPU 単体、GPU 単体で実行するより遅くなる。Task の割り当てを最適にスケジューリングすることでより性能向上させることができる。スケジューリングの方法として、一度 Task を CPU のみ、GPU のみ実行し、プロファイルを取りそれを元に Task の割り当てを決定するなどが考えられる。

6.2.2　パイプライン実行の改良

パイプラインを入れたことに性能は向上したが、期待した通りの効果が出ていない。まず、パイプラインで実行されているか調査する必要がある。綺麗なパイプラインで実行するためには Task の実行時間とデータ転送が同じ時間で終了することが望ましい。Task の粒度や転送するデータのサイズを考える必要がある。

6.2.3 kernel の自動変換

現在、kernel の記述は CPU, OpenCL, CUDA でほぼ同じ記述である。しかし、修飾子など多少の違いがある。どれか 1 種類 kernel を記述すれば他の kernel も生成できるようにするのが望ましい。

謝辞

本研究を行うにあたって、ご多忙にも関わらず日頃より多くのご助言を、ご指導を頂きました河野真治准教授に心より感謝いたします。また、本研究に大変貴重なご意見を下さった情報工学科の先生方に感謝いたします。また本研究に大きく役立つ技術的指導を賜りました、大城信康さん、當眞大千さん、谷成雄さん、並びに研究室の配属生の皆様に深く感謝いたします。最後に、長年に渡り理解を示し、支援してくださった家族に感謝いたします。

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, April 2008.

[2] 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. 2008.

[3] 河野真治. 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, March 2008.

[4] 神里晃. Cell を用いたゲームフレームワークの提案. Master's thesis, 琉球大学理工学研究科情報工学専攻, Feb 2008.

[5] 杉山千秋. Scenegraph と statepattern を用いたゲームフレームワークの設計と実装. 琉球大学工学部情報工学科 平成 19 年度卒業論文, 2008.

[6] 多賀野海人. Cell task manager cerium における task を用いたパイプラインの改良. Master's thesis, 琉球大学理工学研究科情報工学専攻, Feb 2011.

[7] 金城裕, 河野真治. Fine grain task manager cerium のチューニング. 日本ソフトウェア科学会第 27 会大会, Sep 2010.

[8] 金城裕, 河野真治. 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, Apr 2011.

[9] 金城裕, 河野真治. Cerium における datasegment api の設計. 日本ソフトウェア科学会第 28 会大会, Sep 2011.

[10] G. Jaeschke. On strong pseudoprimes to several bases. Vol. 61, pp. 915–926, 1993.

[11] 高山征大. 「cell regza」が搭載する並列化技術「molatominum」. Dec 2009.

[12] Yellow Dog Linux for PowerPC Computers. http://us.fixstars.com/products/ydl/.

[13] SourceForge.JP: Cerium Rendering Engine. https//sourceforge.jp/projects/cerium/.

[14] Sony Corporation. Cell BroadbandEngine ™ アーキテクチャ, 2006.

[15] International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation. *C/C++ Language Extensions for Cell Broadband Engine Architecture Version 2.6*, 2008.

[16] OpenCL. https://www.khronos.org/opencl/.

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