並列プログラミングフレームワーク Cerium の OpenCL, CUDA 対応

小久保 翔平[†1]　河　野　真　治[†2]

当研究室では、PS3, Linux 及び MacOS X 上で動作する並列プログラミングフレームワーク Cerium を提案している。MacOS X 上で GPGPU を行うには、OpenCL または CUDA を用いる方法が考えられる。OpenCL, CUDA の API に対応した API を Cerium に用意することでデータ並列に対応した。タスク並列で実行する場合、データ転送がオーバーヘッドになる。このオーバヘッドを解決するためには、kernel の実行中にデータ転送を行うことでデータ転送をオーバーラップする必要がある。OpenCL では CommandQueue、CUDA では Stream を複数用いることでデータ転送や kernel の実行を並列に行うことができる。複数の CommandQueue, Stream を用いて、自動で並列実行を行うスケジューラーを実装した。実装したスケジューラーを WordCount、FFT を例題に測定し、考察を行う。

Supporting OpenCL, CUDA in Parallel programming framework Cerium

Shohei KOKUBO[†1] and Shinji KONO [†2]

We are developing parallel programming framework Cerium, that is running on the PS3, Linux, Mac OS X and GPGPU. As GPGPU support on Mac OS X, OpenCL or CUDA can be used in an API set. Data Parallel on GPUGPU is also working on CPUs. In Task Parallel, data transfer causes overhead. To resolve this, pipeline data transfer is used both on GPGPU and Many Cores. OpenCL and CUDA have slightly different behavior on Data Parallel and out of order task execution. In WordCount and FFT examples, we show the different and its analysis.

1. はじめに

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。[1] GPU 以外にも Cell[2], SpursEngine, Xeon Phi など様々なプロセッサが存在する。それぞれのプロセッサを利用するにはそれぞれ異なる API を利用する必要があり、それらの対応に多くの時間を取られてしまいプログラムの性能改善に集中することができない。また、GPU や Cell などメモリ空間が異なるプロセッサはデータの転送がオーバーヘッドとなるので、データ転送を効率的に行えるかどうかで処理時間が大きく変わる。

当研究室で開発・改良が行われている並列プログラミングフレームワーク Cerium[3] は様々なプロセッサを統合して扱えるフレームワークを目指している。様々なプロセッサを統合して扱えるフレームワークとしてフランス国立情報学自動制御研究所 (INRIA) が開発している StarPU[4] がある。StarPU は Cerium と同じタスクベースの非同期フレームワークである。タスクという単位で記述することで処理とデータを分離し、より効率的に処理を行うことができる。StarPU にはパイプラインでの実行機構は入ってなく、パイプライン処理を行いたい場合は自分で実装するしかない。しかし、パイプライン処理を書くことは非常に煩雑で難しい。そこで、今回 Cerium に OpenCL, CUDA を用いた Scheduler を新たに実装した。Scheduler は自動でデータ転送をオーバーラップし、パイプラインで処理を行うように設計した。

本論文では、まず OpenCL, CUDA について説明する。その後、既存の Cerium の実装および新たに実装した GPU 実行の機構について説明する。最後に WordCount, FFT を例題として測定し、評価を行う。

†1 琉球大学大学院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. 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。演算用のプロセッサ (本研究では GPU) 上で動作するプログラム OpenCL C という言語で記述する。OpenCL C で記述したプログラムを GPU 上で実行させるために OpenCL Runtime API を利用する。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 のことを host、GPGPU が可能なグラフィックボードなどのことを device と定義している。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特別に kernel と呼ぶ。

2.1.1 CommandQueue

OpenCL では、device の操作に CommandQueue を使用する。CommandQueue は device に Operation を送るための仕組みである。kernel の実行、input buffer の読み込み、output buffer への書き込みなどが Operation となる。

CommandQueue に投入された Operation は投入された順序で実行される。CommandQueue を生成するときプロパティを指定することで Operation を投入された順序を無視して (out of order) 実行することが可能になる。また複数の CommandQueue を生成し、device に投入することでも our of order で実行することが可能である。

out of order で実行する場合、データの依存関係を設定する必要がある。各 Operation を発行する関数には 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設定することができる。out of order 実行を可能にするプロパティをサポートしている device が少ないため、今回は複数の CommandQueue を用いる方法で実装を行なった。

2.1.2 OpenCL におけるデータ並列

3D グラフィックのような多次元のデータを処理する場合に高い並列度を保つには、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 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 1 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。

データ並列による kernel 実行の場合、clEnqueueNDRangeKernel 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することができる。

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。プログラミング言語である 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 に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 CommandQueue と似たような仕組みとして 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。Stream に発行された 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 に発行された Operation に依存関係が存在しない場合、Operation を並列に実行することができる。

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な処理を行う API を利用する必要がある。

2.2.2 CUDA におけるデータ並列

CUDA では OpenCL の WorkItem に相当する単位を thread と定義している。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 実行をする場合、cuLaunchKernel API を使用する。この関数は引数として各座標の block 数と各座標の block 1 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。

cuLaunckKernel で kernel を実行すると各 thread に対して block ID と thread ID が割り当てられる。CUDA には OpenCL とは異なり、ID を取得する API は存在しない。代わりに、kernel に組み込み変数が準備されており、それを参照し、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 で OpenCL の get_global_id(0) で取得できる ID に相当する ID を算出することができる。例として、ある kernel で get_global_id(0) の返り値が 13 の場合、CUDA では図:1 のようになる。

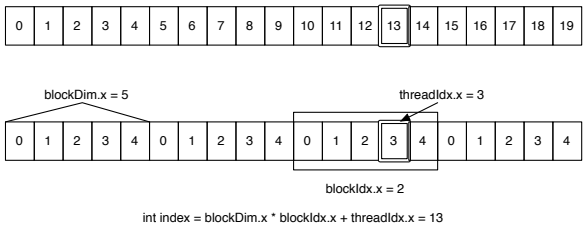

図 1 Calculate Index

3. Cerium

Cerium は、当初 Cell 用の Fine-Grain Task Manager[5] として当研究室で開発された。TaskManager, SceneGraph, Rendering Engine の 3 つの要素から構成されており、今では、PS3 および Linux, MacOS X 上で動作する。GPGPU の Data Parallel を含めて、同じ形式で並列プログラムを記述することができる。

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。関数やサブルーチンを Task として扱い、Task 同士の依存関係を考慮しながら実行される。Task は TaskManager を使って生成する。Task を生成する際に、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。cpy type は Task がどのようなの 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。

図:2 は Cerium が Task を生成/実行する場合のクラスの構成である。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。ActiveTaskList 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序で実行されても問題ない。Task は Scheduler に転送しやすい TaskList に変換してから cpy type に対応する Scheduler に Synchronized Queue である mail を通して転送される。Scheduler ではパイプラインで task が処理される (図:3)。Task が終了すると Scheduler から TaskManager に mail を通して通知される。その通知に従って依存関係が処理され、再び TaskManager から Scheduler に Task が転送される。

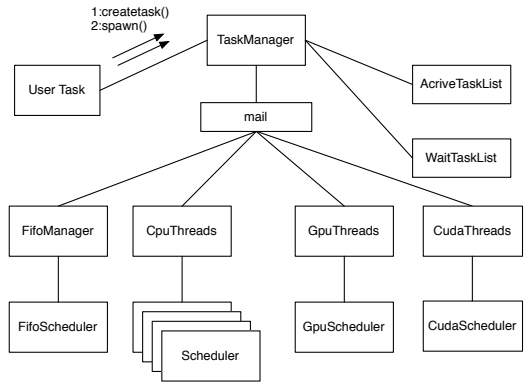

図 2 Task Manager

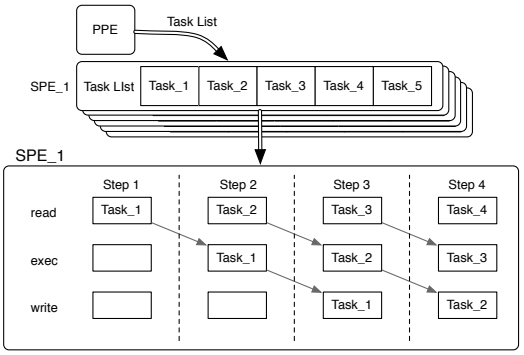

図 3 Task Scheduler

以下に Task を生成する例題を示す。表:1 は Task を生成に用いる API を示している。input データを 2 つ用意し、input データ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。

```
void
multi_init(TaskManager *manager)
{
    A = new float[length];
    B = new float[length];
    C = new float[length];
    for(int i=0; i<length; i++) {
        A[i]=(float)(i+1000);
        B[i]=(float)(i+1)/10.f;
    }

    // create task
    HTask* multiply = manager->create_task(
        MULTIPLY_TASK);
    // set cputype
    multiply->set_cpu(spe_cpu);
```

```
    // set indata
    multiply->set_inData(0,(memaddr)A,
        sizeof(float)*length);
    multiply->set_inData(1,(memaddr)B,
        sizeof(float)*length);
    // set outdata
    multiply->set_outData(0,(memaddr)C,
        sizeof(float)*length);
    multiply->set_param(0,(long)length);
    // spawn task
    multiply->iterate(length);
}
```

Code 1　multiply

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 からのデータ出力先アドレスを追加
set_param	Task のパラメータ
wait_for	Task の依存関係
set_cpu	Task を実行する Device の設定
spawn	Task を登録する
iterate	データ並列で実行する Task として登録する

表 1　Task 生成に用いる API

CPU で実行される Task(OpenCL, CUDA でいう kernel) の記述は以下のようになる。表:2 は Task 側で使用する API である。

```
static int
run(SchedTask *s)
{
    // get input
    float* A = (float*)s->get_input(0);
    float* B = (float*)s->get_input(1);
    // get output
    float* C = (float*)s->get_output(0);
    // get parameter
    long length = (long)s->get_param(0);

    for(int i=0;i<length;i++)
        C[i]=A[i]*B[i];

    return 0;
}
```

Code 2　task

get_input	入力データのアドレスを取得
get_output	データ出力先のアドレスを取得
get_param	パラメータを取得

表 2　Task 側で使用する API

3.2　Cerium におけるデータ並列

Cerium でデータ並列による実行をサポートするために、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、データ並列で実行する Task として生成する。

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 のようになる。

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

表 3　Data 並列実行時の index の割り当て

各 CPU が担当する index は SchedTask に格納してある。データ並列で実行する Task の記述は以下のようになる。

```
static int
run(SchedTask *s)
{
    // get input
    float* A = (float*)s->get_input(0);
    float* B = (float*)s->get_input(1);
    // get output
    float* C = (float*)s->get_output(0);
    // get index
    long i = (long)s->x;

    C[i]=A[i]*B[i];

    return 0;
}
```

Code 3　example

並列プログラムでは、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。iterate 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 1 つの記述で異なる index を持つ Task を Multi Core CPU 上で実行することが可能になった。

4.　Cerium の GPGPU への対応

本章では、まずはじめに GPU プログラミングの特徴および問題について述べ、Cerium への実装でどのように対応したかについて説明する。

4.1　GPU プログラミングの特徴および問題

まず Multi Core CPU に対するプログラミングと同様に性能を向上させるためには、プログラム全体を対象と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データ並列で処理を行うことで CPU より高速で演算を行うことができる。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だけでプログラムの性能は向上する。しかし、多くのプログラムはその限りではない。GPGPU においてネックになる部分はデータ転送である。GPU の Memory 空間 (図:4) は CPU(図:5) 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。データにアクセスするためには Memory 空間ごとコピーするしかない。これが大きなオーバーヘッドになるので、データ転送をオーバーラップする必要がある。今回新たに、デー

タ転送を自動でオーバーラップするように OpenCL および CUDA を用い Scheduler を実装した。

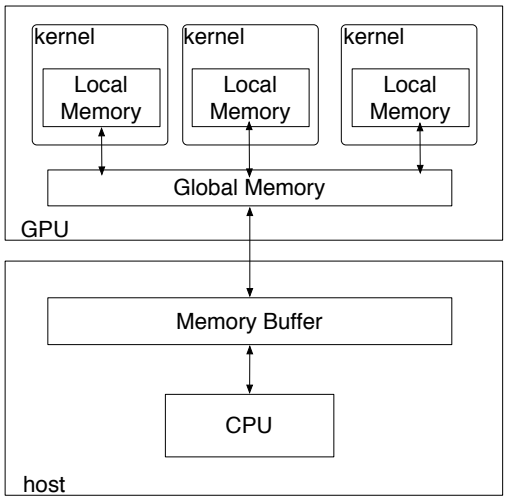

図 4 Gpu Architecture

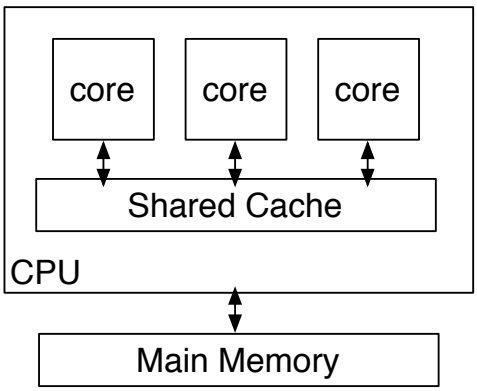

図 5 Cpu Architecture

4.2 OpenCL および CUDA を用いた Scheduler の実装

Scheduler と CpuThreads に対応させる形で OpenCL を用いた GpuScheduler, GpuThreads、CUDA を用いた CudaScheduler, CudaThreads を実装した。TaskManager から転送された TaskList の情報をもとに device 上のメモリ領域を確保する。その後、OpenCL ならば CommandQueue、CUDA ならば Stream に Operation を発行していく。Operation は発行された順序で実行されるので、host から device へのデータ転送、kernel の実行、device から host へのデータ転送の順に発行する。非同期 API を用いることでデータ転送や kernel の実行を並列に行うことができる。通常、非同期 API を用いる場合は依存関係を考慮した同期が必要になるが転送されてくる Task の依存関係は TaskManager ですべて解消されているので Scheduler 側では順番を考えず Task を実行して問題ない。host から device へのデータ転送は、OpenCL では clEnqueueWriteBuffer、CUDA では cuMempcyHtoDAsync を用いて行われる。clEnqueueWriteBuffer は第三引数に CL_FALSE を指定することで非同期なデータ転送を行う。転送されてきた TaskList からデータ並列またはタスク並列で実行するか決定する。データ並列で実行する場合は、OpenCL では clEnqueueTaskNDRangeKernel、CUDA では cuLaunchKernel を用いる。タスク並列で実行する場合は、OpenCL では clEnqueueTask、CUDA では cuLaunckKernel の引数を 1 に設定することで実行することができる。device から host へのデータ転送は、OpenCL では clEnqueuReadBuffer、CUDA では cuMemcpyDtoHAsync を用いて行われる。clEnqueueReadBuffer も clEnqueueWriteBuffer と同様に第三引数に CL_FALSE を指定することで非同期実行となる。転送されてきた Task がすべて終了すると Synchronized Queue 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 で依存関係が解消し、再び TaskList を転送する。GpuScheduler および CudaScheduler は複数の CommandQueue および Stream を持っており、パイプラインで実行される。

kernel の記述は以下のようになる。

```
__kernel void
multi(__global const long *params, __global
      const float* A, __global const float* B,
      __global float* C)
{
    // get index
    long id = get_global_id(0);
    C[id]=A[id]*B[id];
}
```

Code 4 multiply(OpenCL)

```
extern "C" {
    __global__ void multi(long* params, float*
          A, float* B, float* C) {
        // calculate index
        int id = blockIdx.x * blockDim.x +
             threadIdx.x;
        C[id]=A[id]*B[id];
    }
}
```

Code 5 multiply(CUDA)

修飾子など若干の違いはあるが、ほぼ同じ記述で書くことができるが CPU, OpenCL, CUDA のどれか 1 つの記述から残りのコードも生成できるようにすることが望ましい。

5. Benchmark

本章では、WordCount, FFT を例題として用い、

本研究で実装した GpuScheduler および CudaScheduler の測定を行う。

測定環境

- OS : MacOS 10.9.2
- CPU : 2*2.66GHz 6-Core Intel Xeon
- GPU : NVIDIA Quadro K5000 4096MB
- Memory : 16GB 1333MHz DDR3
- Compiler : Apple LLVM version 5.1 (clang-503.0.40) (based on LLVM 3.4svn)

6. WordCount

今回は 100MB のテキストファイルに対して WordCount を行なった。表:4 は実行結果である。

	Run Time
1 CPU	0.73s
2 CPU	0.38s
4 CPU	0.21s
8 CPU	0.12s
OpenCL(no pipeline)	48.32s
OpenCL(pipeline)	46.74s
OpenCL Data Parallel	0.38s
CUDA(no pipeline)	55.71s
CUDA(pipeline)	10.26s
CUDA Data Parallel	0.71s

表 4 WordCount

パイプライン処理を行うことで CUDA では 5.4 倍の性能向上が見られた。しかし、OpenCL ではパイプライン処理による性能向上が見られなかった。OpenCL と CUDA を用いたそれぞれの Scheduler はほぼ同等な実装である。OpenCL でパイプライン処理を行うために実行機構を見直す必要がある。一方で、データ並列による実行は 1CPU に対して OpenCL では 1.9 倍、CUDA では 1.02 倍という結果になった。どちらもタスク並列による実行よりは優れた結果になっている。CUDA によるデータ並列実行の機構を見直す必要がある。

6.1 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を行う。使用する画像のサイズは 512*512 で、画像に対して High Pass Filter をかけて変換を行う。表:6.1 は実行結果である。

	Run Time
1 CPU	0.48s
2 CPU	0.26s
4 CPU	0.17s
8 CPU	0.11s
OpenCL	0.09s
CUDA	0.21s

表 5 FFT

1CPU に対して OpenCL ではの 5.3 倍、CUDA では 2.2 倍の性能向上が見られた。しかし、WordCount の場合と同様に OpenCL と CUDA で差がある。WordCount と FFT の結果から CudaScheduler によるデータ並列実行機構を見直す必要がある。また、FFT の OpenCL の kernel は cl_float2 というベクター型を用いている。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おり結果が良くなっている可能性がある。

7. 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 を OpenCL および CUDA に対応させた。OpenCL および CUDA に対応させたことで Cerium は単一の記述から CPU および GPU 上での実行が可能になった。WordCount, FFT を例題に用い、Scheduler の測定も行なった。OpenCL と CUDA で異なる結果が出たことからそれぞれで最適なチューニングの方法が違うことがわかる。どちらもチューニングを行えば同等な結果が出ると考えられるのでプロファイラなどを用いて、実装を見直すことが今後の課題となる。また、Cerium は CPU と GPU の同時実に対応している。しかし、スケジューリングを行わず 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結果が悪くなる。Task の割り当てを最適化することで性能を向上させることが予想される。スケジューリングの方法として、一度 Task を CPU のみ、GPU のみで実行し、プロファイルを取ることで Task の割り当てを決定するなどが考えられる。

参 考 文 献

1) 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, 情報処理学会論文誌コンピューティングシステム (2008).
2) Sony Corporation: Cell broadband engine architecture (2005).
3) : SourceForge.JP: Cerium Rendering Engine, https//sourceforge.jp/projects/cerium/.
4) Augonnet, C., Thibault, S., Namyst, R. and Wacrenier, P.-A.: 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 (2011).
5) 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).
6) Chiaki SUGIYAMA: SceneGraph と StatePattern を用いたゲームフレームワークの設計と実装 (2008).
7) 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).
8) 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).
9) Aaftab Munshi, Khronos OpenCL Working Group: *The OpenCL Specification Version 1.0* (2007).
10) Khronos OpenCL Working Group: *OpenCL 1.2 Reference Pages* (2012).
11) : CUDA, https://developer.nvidia.com/category/zone/cuda-zone/.