

PS3 上でのゲームプログラミング

河野 真 治†

本大学では、学生の実習に PS3 の実機を用いたゲーム作成を導入している。Cell.B.E は、安価に手に入る Many Core 機であるが、従来とは異なるプログラミングを要求される。特に、PS3 の GPU である RSX の仕様は公開されていないので、教育目的を含めて、Cerium Rendering Engine を作成した。Cerium Task Manager はメインメモリとのやりとりを DMA で隠した関数である Task の Queue を管理し、恒常的な並列性を維持する。Cerium Engine での若干の並列プログラミングの例について報告する。

Game Programming on PS3

SHINJI KONO†

Game Programming on PS3 is introduced in our University as programming training. PS3 is a cheap Many Core Architecture, but it requires different programming from conventional one. Especially specification of PS3's GPU RSX is not opened, so we decided to create Cerium Rendering Engine as an educational tool. In order to keep constant parallelism, Cerium Task Manager provides Task queue management of Task Functions, which has hiding mechanism of memory copy from main memory using DMA. We report examples of parallel programming on our Cerium Engine.

1. 歴史的経緯

当研究室では、主に PlayStation を用いた実機上のゲームプログラミングを取り扱って来た。2007 年から、PS3 上のゲーム作成を行っており、Cell B.E.¹⁾でのプログラミングを学生に行わせている。

PS3 上では (最近の PS3 ではサポートされていないが)、Fedora Linux が動作し、その上で、6 個の Cell にアクセスすることが可能である。Graphic Processor RSX の仕様は公開されないため、初期は、Open GL Mesa を用いて、PPE で描画を行っていた。

Open GL Mesa の描画部分を Cell の SPE を行なったが、Open GL Mesa の実装は、描画ルーチンを「すべて C のマクロで記述」するような手法であり、一度、マクロを展開したソースを変更して SPE に対応させるような手法となる。全体の対応は現実的なものでないと判断した。Galium²⁾では、同様のアプローチが行われている。

そこで、Open GL とは独立な Engine を、SPE 上の Task を管理するマネージャ (Cerium Engine) と一緒

に作成することにした。その際に、SPURS Engine³⁾を参考にしている。ただし、SPURS Engine の仕様も公開されていないために、独自実装となっている。

その後、2009 年になって Open CL の最初の仕様が開示されたが、Cerium Engine は、比較的似た構成となっているが、どちらかと言えば、Cell に特化した構成となっている。

狙いとしては、SPE の複雑な SIMD 処理は、なるべく使わずに、学生に Many Core CPU のプログラムに入っていけるライブラリを目指している。特定のアプリケーションで最高速を目指すのではなく、ゲームで出て来る汎用的な処理を自然に並列に実行させることが出来る。

現在は簡単なゲームをプログラムを作成できるようになっており、Cell だけでなく、Mac OS X 上、Linux 上でも動作する。Core i7 等のマルチコアにも対応可能であるが、現在は実機がないので対応していない。

本論文では、Many Core Programming の実際の問題 (特に、Rendering Engine と Word Count を例題に) を取り上げ、Cell 固有の問題と、Cerium Engine での解決を示す。

また、Scala / Erlang などのアプローチや、Open CL との比較を行う。

† 琉球大学

University of the Ryukyus

2. Many Core Programming の特徴

PS3 で用いられた Cell B.E. では、2 thread の Power PC である PPE と、128bit vector register を持つ SPE が 6 個付いている。SPE は 256Kbyte Local Store を PPE のメインメモリとは別に持っていて、明示的な DMA で通信を行う。GeForce GTX 280 等の GPGPU では、16kB を共有する 8 個の演算ユニット SPE をまとめた SM を 30 個用意しており、メインメモリにはキャッシュを通してアクセスする。Intel の Nehalem では、256KB の L2 キャッシュがあり、1 Chip 内の 4 Core が 8MB L3 キャッシュを共有し、さらに、Quick Path と呼ばれる Interconnect で Chip 間を相互接続する。個々の Core での処理は、256Kbyte 程度にまとめるとキャッシュに収まることになる (図 2)。

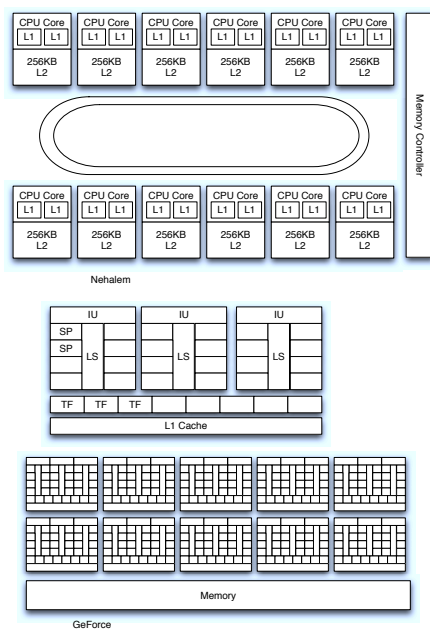


図 1 manycore

並列処理では、Amdahl 則があり、部分的に並列化するのは、全体的な速度の向上の比率が低くなってしまふ。プロセッサの数 S 、並列化の比率 P に対して、処理時間は、 $(1 - P) + (P/S)$ となる。並列化が 50% では、どんなに多くのプロセッサを投入しても効果は 2 倍程度になってしまう。従って、並列化率を上げることが重要になる。

つまり、一般的なアプリケーションで、for 文等で

部分的に並列実行するのでは十分な効果を上げることは出来ない。しかし、科学技術計算や画像処理などでは、巨大なメインループがあるので、並列に実行する処理自体は十分にある。

処理は、データと処理の組で規定する Task で構成する。Open/CL では Kernel と呼ばれている。

実際に問題になるのは、メモリの待ち時間である。Many Core では、Local Store やキャッシュは限られていて、メインメモリから、そこにデータを持って来る必要がある。これらのコピー時間の間、プロセッサが待っているようだと、並列化率が落ちてしまう。

これを防ぐには、キャッシュや Local Store へのデータ転送の間に、別な処理を行えば良い。つまり、ソフトウェアパイプライン的にデータ転送と計算処理を重ねて処理する。つまり、

$$\text{Many Core Programming} = \text{Data Parallel} + \text{Pipe Line}$$

と言う構成になる。東芝/SCEI では、Cell 用に SPURS Engine³⁾ というフレームワークを提供している。Apple の Open/CL⁴⁾ では、Kernel と呼ばれるタスクを Queue に入れることにより、パイプライン実行を可能にしている。

ここでは、キャッシュと Local Store を合わせて LS と呼ぶ。

2.1 メモリのアクセスパターン

汎用的なプログラムでは、Local Store に閉じた処理を行うことは、ほとんどない。従来のように、巨大なメモリをランダムアクセスするような処理では、Many Core Programming の効率上がることはない。

コードは、Task を実行している間、必ず LS に常駐する必要がある。これは、前もって予測することは用意である。しかし、Task は細かく分割する必要がある。64kbyte 程度が望ましいが、これは経験的には 5,000 行程度であり、単一の Task では、足りなくなることは少ない。しかし、全ての Task のコードが LS に収まることはありえない。

順序良くアクセスされるデータは、処理をしながら並列に DMA や、キャッシュのアクセスなどでロードすれば良い。(図 2.1)

問題は、処理によってアクセスする場所が変わる場合である。描画処理する時の Texture 画像はプログラムの実行時に初めて、どこが必要かが判明する。これらは、キャッシュと同様の扱いをする必要がある。Cell では、SPE がキャッシュを持たないので、これをソフトウェアで実現する必要がある。キャッシュを暗黙に

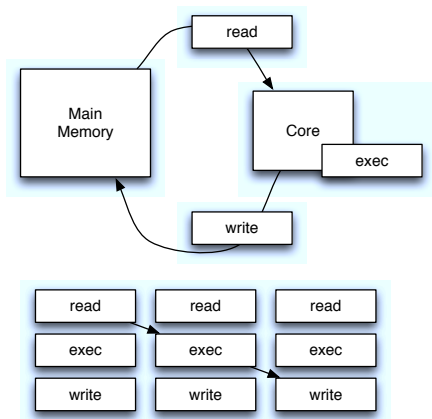


図 2 task

扱う場合でも、何らかの工夫が必要な場合が多い。

データの入力時だけでなく、データの書き出し場所が実行時に決まる場合がある。データを決まった形でメモリ上に分散させなければ、次の Task に引き渡す処理が複雑になる。この処理は、メモリ上を広くアクセスする必要があり、SPE や GPGPU では自分自身では処理できないか、非常に遅くなる。この場合には、書込キャッシュを実装する必要がある。

3. Copy するかしないか

従来の逐次型プログラムでは、データのコピーはコストであり、コピーをいかに減らすかが重要だった。並列処理が多用される Many Core では、データは基本的に LS にコピーする必要がある。計算主導なアプリケーションでは、CPU が動いている限り、コピーのコストは見えない。実際、Cache への移動や、DMA により隠されることになる。

時分割処理では、同期コストは、待ち時間中に他の Task が走るので、それほど高くはない。しかし、Many Core の場合は、Cache または、LS 中の Task の入れ換えを伴う。例えば、Core 側から read / write のシステムコールを呼ぶことは、明示的に待つてしまう。

待ち合わせや競合を防ぐには、コピーを取ることが簡単な解決になる。実際、LS にコピーされて処理され、メインメモリに書き戻すという処理なので、自動的にコピーされることになる。

パイプライン処理は、命令レベル、Core レベル、Task レベルで行われるが、パイプラインバッファは読み書きペアが必要になる。同じ場所に書き戻す方法ではパイプライン処理を実現することは出来ない。(図 3)

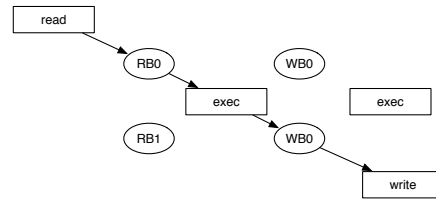


図 3 pipeline

つまり、オブジェクト指向プログラムををするとしても、ポインタを ID として持ち、インスタンス変数をその場で書き換えるような方法は Many Core との相性は良くない。

4. Scene Graph

ゲームプログラミングは、3D グラフィックス処理と Real-time プログラミングの組合せとなる。

ゲームプログラムで、複雑な並列処理に集中するのは望ましくない。

一つの方法は、3D グラフィックス部分をオブジェクトのノードとする Scene Graph にすることである。ゲームプログラムは、Scene Graph のノードの動作 (Move) と、相互作用 (collision) からなる。これらと、動的な Scene Graph の構成により、ほとんどのゲームプログラムを行うことが出来る。

Scene Graph には、Open Scene Graph や Java 3D などがある。Scene Graph のノードには、視点 (カメラ)、光源 (複数)、親子関係がある。Scene Graph が持つ情報には、3D の Polygon と、その normal vector (Polygon の向き、曲面である場合は複数)、そして、Polygon に貼られた 2 次元の画像 (Texture) からなる。

Collision と Move を記述するだけで、十分な並列性をゲームプログラムから自動的に抽出するのが、Cerium Engine の一つの目標である。(図 4)

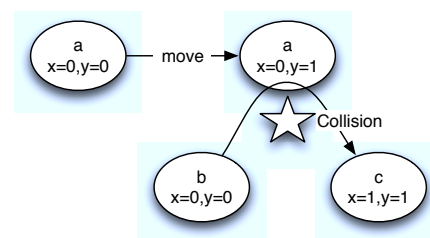


図 4 move

5. Cell のアーキテクチャ

PS3 の Cell では、メインメモリは 256MB ある。使用可能な 6 個の SPE には Local Store が 256Kbyte ずつある。SPE には、128bit register が 128 個あり、Local Store はキャッシュされていないがレジスタ並に高速にアクセス出来る。LS はレジスタと考えると良い。(図 5)

SPE と PPE はリングバスで結ばれている。MFC(Memory Flow Controller) に、32bit Mail によりコマンドを送ることにより、複数の DMA を起動し、SPE とメインメモリ間での転送を行うことが出来る。Mail は FIFO であり、SPE と PPE の通信に使うことも出来る。

DMA には、転送メモリの位置と数を指定する DMA と、複数の位置と数を LS 上に置き、それをまとめて転送する List DMA が用意されている。

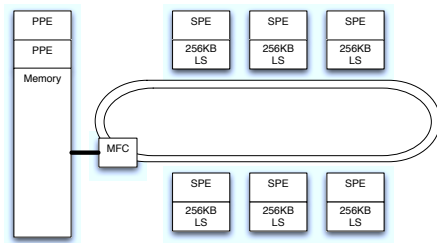


図 5 cell

6. Cell B.E. 上のプログラミング

libSPE2 とは、PPE が SPE を扱うためのライブラリ群である⁵⁾。libSPE2 は SPE Context Creation、SPE Program Image Handling、SPE Run Control、SPE Event Handling、SPE MFC Problem State Facilities、Direct SPE Access for Applications という基本構成でできている。Cell の基本プログラムは次のようになる。

- (1) create N SPE context
- (2) Load the appropriate SPE executable object into each SPE context's local store
- (3) Create N threads
- (4) Wait for all N threads to terminate

各 SPE は、TLB を持っていて、起動した PPE の Thread が持つ仮想メモリ空間 (64bit addressing) に DMA でアクセスすることが出来る。

6.1 SPE C/C++ 言語拡張

SPE では基本的な C 言語の機能の他に、Cell 特有の拡張が行われている⁶⁾。表 1 に主な API を記す。

表 1 SPE C/C++ 言語拡張 API

spu_mfcdma32	DMA 転送を開始する
spu_read_in_mbox	PPE からの mail を取得する
spu_write_out_mbox	PPE へ mail を送信する
spu_add, spu_sub, spu_mul	SIMD 演算 (加算、減算、乗算)

gcc は、SPE の Vector 演算をサポートしており、`__attribute__((vector_size(16)))` を使用することにより適切な命令を出してくれる場合がある。`spu_add()` 等で手動で処理することも可能である。

7. Cerium Engine

Cerium Engine は、PS3 上の Scene Graph の描画と変更によって記述されたゲームを実行する Engine である。Task の生成を管理する TaskManager、Task を Schedule する Scheduler、ソフト的なキャッシュを管理する MemorySegment そして、SceneGraph ライブラリ、複数の Task で実行される Rendering Engine からなる。

C++ で記述されており、Mac OS X、Linux、PS3 上で同一のソースで動作する。

SceneGraph は、Blender (Open Source な 3D modeling Tool) から、Python Script を作って生成された XML である。内部でも生成変更可能である。

SceneGraph のパラメータを変更することによりゲームが進行し、それと同時に Pipeline 的に SceneGraph の描画を行う。

Cerium Engine は、Open CL とは独立に設計しているが、結果的には似たものになっている。

7.1 Task Manager

Cerium の Task は、型のない入力と出力を持つシンプルな関数である。入力と出力は、Cell の DMA によって用意される。List DMA があるので、断片化されたデータを読み書きすることも可能になっている。

`create_task()` で、HTask という構造を作成し、`spawn()` することで active queue に登録する。

Task は、番号で登録される。SPE と PPE では独立な番号を用いている。手動による Overlay で、SPE 上のコードの入れ換えを行っているため、ポインタで指定することは出来ない。Open/CL では、文字列で指定したプログラムを llvm に引き渡す形式だが、その点は異なる。

Cerium は、PPE の Task と、SPE の Task の二種類を持っており、それぞれ別なキューで管理されている。さらに、Task の終了時に呼ばれる Task がある。

PPE Task は相互の Dependency を持ち、Dependency が満たされた時点で PPE または SPE 上で実行される。

普通の OS と異なり、高度なスケジューリングは行われず、Task は十分小さく、基本的に preempt されずに実行されるので、Round Robin などの工夫は不要である。通常のスレッドやプロセスとは異なる。

SPE 上には最小限の Scheduler が存在している。SPE 上では基本的に Task は生成しない。

SPE Task からはメインメモリは参照できないが、MemHash という LRU キャッシュが実装されている。しかし、特に構文的な制限があるわけではなく、PPE Task からはメインメモリ全域をアクセスすることが出来る。

一つの PPE Task から生成された SPE/PPE Task は、まとめて投入される。投入した Task は Cell の Mail を使って、SPE/PPE スケジューラに伝えられる。終了した Task は、Dependency を解消し、Active になった Task を Active Queue に移動する。

DMA は、Task の中から起動することも可能になっている。PPE や、Mac OS X 上でも、DMA の Emulation を行っており、同じコードで、PS3、Mac OS X で動作させることが出来る。

HTask は、実行時は SchedTask という型のオブジェクトになる。Cerium Engine に対する処理は、SchedTask へのメソッドで行われる。

7.2 Task 間の同期

Task はメインメモリ上に Queue を持っており、`task->wait_for(another_task)`

という形で待ち合わせを行う。これは、task queue に waiting queue を持つこととで実現されている。実際の実行は、Core 上で行われる。Core から見たメインメモリはかなり遠いので注意が必要である。

```
task->set_post(post_task,in, out)
```

という形で、Task の継続を指定することが出来る。継続の Task では、DMA は行われず。主に、終わったタスクが書き出したデータの整理、次のタスクの起動などに使われる。スケジューラの前に起動されるので、継続で接続された Task のチェーンは、見掛け上、シングルスレッドで実行される。

何もしない Dummy task を作り、それを wait_for することにより、同期を取るような手法が多用される

ことになる。これは、一種のバリアである。

task の終了は Mail による通知であり、Single Thread な Task Manager が Mail 待ちループを持っていて処理する。

```
do {
    ppeManager->schedule(ppeTaskList);
    ppeTaskList = mail_check(waitTaskQueue);
} while (ppeTaskList || spe_running >0);
```

つまり、wait_for/set_post はメインメモリ上のシングルスレッド Task として実行される。Task Manager を複数のスレッド、あるいは分散したプロセスとして実装することも可能だと思われるが、それが、wait_for/set_post の意味にどう係わるかは、かなり難しいと思われる。現状では、シングルスレッドが使いやすい。

7.3 Task

一つの Task は、read/exec/write の三つに分解されて、それぞれが、さらにパイプライン的に実行される。TaskManager は、それが可能なように、Core に複数の Task の集合 (TaskList) を投入する。

```
void
Scheduler::run()
{
    task1 = new SchedNop();
    task2 = new SchedNop();
    task3 = new SchedNop();
    // main loop
    do {
        task3->write();
        task2->exec();
        task1->read();
        delete task3;

        task3 = task2;
        task2 = task1;
        task1 = task1->next(this, 0);
    } while (task1);
    delete task3;
    delete task2;
}
```

ここで、read/exec/write は Task の状態によって変わるステートパターン用の仮想関数である。何もしないもの、DMA の開始/待ち合わせ、次のタスクの取得、次の TaskList の取得などを行う。つまり、read/write は非同期で、ほとんど待ち時間はないと想定されている。exec が実際のユーザが定義した Task の関数を呼び出す。

TaskList は、Core 側が Empty になったのを Mail で Main 側に知らせて、Main 側から Mail により TaskList へのポインタを取得して、DMA により転

送を行う。これは一つの Task として実装されていて、上のパイプラインループの中で実行される。

7.4 Memory Segment Manager

PPE 上/SPE 上のメモリは、MemorySegment によって管理されている。これは、Hash access 可能な double linked されたメモリ領域である。

これは、set_global, get_global によって、Core(SPE) 上に常駐する領域である。これは明示的に作成削除する必要がある。

Cerium Engine の性質上、ほとんどの処理は Pipeline Buffer 上で行われる。つまり、入力 MemorySegment から出力 MemorySegment に書き出される Task の集合である。

MemorySegment は読むだけ、あるいは書くだけとなる。そうでないと、Pipeline 実行時に途中でデータを変更されてしまうことになる。

Task の実行時には、必ずメモリのコピーを伴うし、DMA のオーバヘッドのほとんどは隠されてしまうので、積極的にコピーする。メインメモリは、SPE の LS に比べれば広大なので、倍量のメモリを取っても問題ない。Texture や Polygon のデータ等、変更されない場合はコピーの必要はない。

必ず Copy を伴うので、細かく解放する必要がない。したがって、常に Copying GC を行っているようなことになる。これは、Pool を多用する Apache Web Server と同じような実装となる。

8. SPE 上の Code

SPE 上のコードは、256Kbyte しかないので、なんらかの方法で入れ換える必要がある。SPE 上のライブラリは固定されているので、その部分以外を relocatable にするようなコードをアセンブラから生成する。

コード自体は、Memory Segment Manager により、LRU/Hash で管理される。

現在は、gcc の Overlay code を流用しているので、そのまま gdb でデバッグも可能であるが、Task の単体テストが望ましい。

SPE 上のプログラムは C++ だが、g++ の仮想関数の実装が relocatable に出来ないため、SPE 上の Task はメソッドではなく、C の関数である。

Many Core 上でのオブジェクト指向的な利点はほとんどないので、特に問題ないようである。

現在は、task_list という配列に、そのまま、Task の関数へのポインタ run と、その関数を SPE 上へロードする関数を wait を用いている。

```
scheduler->dma_wait(DMA_READ);
task_list[task->command].wait(scheduler,task->command);
task_list[task->command].run(this, readbuf, writebuf);
```

この方法では、Task が増えると task_list 自体が大きくなり、メモリを圧迫してしまう。task_list 自体は、メインメモリ上に置く方法が望ましい。現状では、Overlay code 自体がテーブル持っているのも、それは意味がない。おそらく、Overlay code 自体を捨てる方が望ましいと思われる。

9. Cerium Programming

Cerium では、すべてを Task で記述する。処理は、
int TMain(TaskManager *manager, int argc, char *argv[])

から始まる。これは、SDLmain を真似ている。Cerium は SDL⁷⁾ を使用しているので、main が衝突しないようにこのようになっている。

ここでは、word count を例に取り上げる。まず、File を mmap によってメインメモリにマップする。read で書いても良いが、待ち合わせが生じるので複雑になる。mmap したメインメモリを分割し、Task に割り当てる。

```
分割された word count が出力する領域を確保する。
int out_size = division_out_size*out_task_num;
unsigned long long *o_data =
(unsigned long long*)manager->allocate(out_size);
```

まず、各 SPE の結果を合計して出力するタスクを起動する。

```
HTask *t_print = manager->create_task(TASK_PRINT);
t_print->add_inData(o_data, out_size);
t_print->add_param(out_task_num);
t_print->add_param(status_num);
```

これで入出力が確保されたので、word count する Task を起動する。

```
/*渡すデータの最後。(スペース、改行以外)*/
int word_flag = 0;
int i;
for (i = 0; i < task_num; i++) {

    t_exec = manager->create_task(TASK_EXEC);
    t_exec->add_inData(file_mmap + i*division_size,
        division_size);
    t_exec->add_outData(o_data + i*status_num,
        division_out_size);
    t_exec->set_param(0,division_size);
    t_exec->set_param(1,word_flag);
    t_exec->set_cpu(SPE_ANY);
    t_print->wait_for(t_exec);
    t_exec->spawn();

    word_flag =
```

```

((file_mmap[(i+1)*division_size-1] != 0x20)
&& (file_mmap[(i+1)*division_size-1] != 0x0A));
    size -= division_size;
}

```

add_inData/add_outData/set_paramなどで、Taskに必要なパラメータを引き渡している。

実際には端数処理が必要だが省略する。最後に、printのTaskを起動する。

```

t_print->spawn();

```

各Taskの処理は以下のように記述する。

```

/* これは必須 */
SchedDefineTask(Exec);

static int
run(SchedTask *s, void *rbuf, void *wbuf)
{
    char *i_data = (char*)s->get_input(rbuf, 0);
    unsigned long long *o_data =
        (unsigned long long*)
        s->get_output(wbuf, 0);
    long length = (long)s->get_param(0);
    long word_flag = (long)s->get_param(1);
    int word_num = 0;
    int line_num = 0;

    word_flag = 0;

    for (int i=0; i < length; i++) {
        if((i_data[i] != 0x20) &&
            (i_data[i] != 0x0A)) {
            word_num += word_flag;
            word_flag = 0;
        }
        ...
    }
    word_num += word_flag;

    o_data[0] = (unsigned long long)word_num;
    o_data[1] = (unsigned long long)line_num;
}

```

get_input/get_output/get_paramなどでTaskに必要なパラメータを取り出している。

集計Taskは簡単なので省略する。

10. Word Countの問題点

この方法では、6台のSPEで走らせるよりも、PPE側で走らせる方が高速になる。

SPE上

```

./word_count -file a.txt -cpu 6
0.07s user 1.45s system 162\% cpu 0.938 total

```

PPE上

```

./word_count -file a.txt -cpu 0
0.64s user 0.23s system 99\% cpu 0.872 total

```

まず、大量のTaskを一気に起動することになるので、Taskのデータ量が多い。PPEでのTaskQueueに線形リストを使っていると、その処理に時間がかかる。Task終了時には、WaitQueueの削除などの線形リストではO(n)かかる処理があるためである。

Double Linked Listを使う以前は、実際に、Taskを少しずつ起動すると高速になる。しかし、TaskQueueをDouble Linked Listにすると、その影響はなくなり、一辺に大量に起動しても構わない。

次に、mmapしたメモリに早めにランダムにアクセスしてしまう。これは、ファイルに高速にアクセスするには、あまり望ましくない方法である。しかし、ファイル全体がOSのキャッシュに入っている状態では、それほどペナルティは存在しない。

キャッシュに入っていない状態では問題となる。ファイル全体を高速に複数のCoreに提供するようなAPIが必要となる。

分割した際の端数処理と総計の処理は、別Taskで集計するが、これは単一のTaskとなる。これを、分割集計と並列に走らせる必要があるが、プログラミングは繁雑となる。

SPE上では、128bitレジスタが活かされるようにvector型の宣言をする必要がある。

```

typedef char *cvector
__attribute__((vector_size(16)));

```

などを使うと、

lqx	\$16,\$20,\$5	16byte 一括 load
ceqb	\$15,\$8,\$16	16byte 一括比較
gbb	\$3,\$15	gather Bits from Bytes

などのvector命令をgccが生成することが出来る。この宣言は、vector命令をサポートしないアーキテクチャでも無害だが、演算がvector同士で制限されるので注意が必要である。ただし、spu-gccの実装は、gcc 4.1.1では、まだ正しくなく、うまく動作しない。また、高速にもならない。実際、Word Countのようなものだと、あまり、うまくvector命令に落ちないようである。

パイプライン処理には、パイプラインの切替えがあり、その段階で並列度が落ちることがある。他の並行して走っているより高度なパイプライン処理があれば、それは自動的に隠されるはずである。しかし、Word Countのような例では、それを期待することは難しい。

これらは、ほとんど待ち時間となって現れるが、プログラム上、あるいは、デバッグ上で、それを確認す

ることは難しい。

Word Count が適切な例題ない (SPE 向きではない) と言うのはあるが、まだ、隠された Overhead が存在するのではないかと考えている。

11. より複雑な例 (キャッシュへのアクセス)

zbuffer を用いて、描画を行う場合、描画に必要な Texture のデータをメインメモリから取って来る必要がある。

```
g->tileList =
(TileListPtr)smanager->global_get(GLOBAL_TILE_LIST); smanager->dma_load(g->spack,
(memaddr)spackList[index],
sizeof(SpanPack), SPAN_PACK_LOAD);
}
g->prev_index = index;
smanager->dma_wait(SPAN_PACK_LOAD);
}

if (tex_z < g->zRow[localx + (rangex*localy)]) {
memaddr tex_addr;
int tex_localx;
int tex_localy;

tex_addr = getTile(tex_xpos, tex_ypos,
span->tex_width, (memaddr)span->tex_addr);
tex_localx = tex_xpos % TEXTURE_SPLIT_PIXEL;
tex_localy = tex_ypos % TEXTURE_SPLIT_PIXEL;
TilePtr tile =
smanager->get_segment(tex_addr,g->tileList);
smanager->wait_segment(tile);

updateBuffer(g, tex_z, rangex, localx, localy,
tex_localx, tex_localy,
normal_x, normal_y, normal_z, tile);
}
```

ここでは、GLOBAL_TILE_LIST が SPE 上に Task 間で共有されるメインメモリのキャッシュとして確保されている。Texture を格納している tex_addr (64bit) のアドレスが確定すると、get_segment() により、キャッシュにアクセスする。ここでは、すぐに wait_segment してしまっているが、本来は、少し時間があつた方がよい。

memaddr は、メインメモリのポインタ型を表している。メインメモリが 64bit/32bit でも、SPE のアドレス空間は 256kbyte(20bit) であり、一致しない。

Cerium では明示的にキャッシュを作成しているが、他のキャッシュをサポートしている Many Core の場合でも、待ち時間は生じるので、何らかの工夫は必要となる。

12. より複雑な例 (明示的な DMA)

以下は、ポリゴンのデータから、Span (同じ y 座標を持つ直線) を抜き出す処理に出て来る、Span-Pack(Span の集合) の書き出し部分である。

```
if (charge_y_top <= y && y <= charge_y_end) {
int index = (y-1) / split_screen_h;
/**
* 違う SpanPack を扱う場合、
```

```
* 現在の SpanPack をメインメモリに送り、
* 新しい SpanPack を取ってくる
*/
if (index != g->prev_index) {
tmp_spack = g->spack;
g->spack = g->send_spack;
g->send_spack = tmp_spack;

smanager->dma_wait(SPAN_PACK_STORE);
smanager->dma_store(g->send_spack,
(memaddr)spackList[g->prev_index],
sizeof(SpanPack), SPAN_PACK_STORE);

smanager->dma_load(g->spack,
(memaddr)spackList[index],
sizeof(SpanPack), SPAN_PACK_LOAD);
g->prev_index = index;
smanager->dma_wait(SPAN_PACK_LOAD);
}
```

ここでは、明示的に書き出し領域を DMA で、読み書きしている。この部分を get_segment で書き換えることも可能であるが、例として敢えて持ってきている。dma_wait が先行しているのは、前の dma_store との処理がパイプライン的に行われているためである。

この場合は書込処理なので、SPE 間で同期を取ることが必要になる場合があるが、そのような同期は、ここでは用意していない。

13. Scene Graph

Scene Graph の処理 (Move, Collision) と、Scene Graph の Rendering の各ステージは、それぞれ、パイプライン化される。ポインタを使ったグラフ構造をそのまま使うと、Core にコピーした時に困ることになる。したがって、Scene Graph は、コピーしながら生成する手法を取る。

Scene Graph の Move は、Core 上でノードのプロパティを変更するだけで、必要なのはユーザ入力だけである。しかし、Collision の場合は、 $O(N^2)$ で処理する必要がある。

Move/Collision は、ステートパターン、つまり、Move/Collision に状態を表すオブジェクトあるいは関数、Cerium Engine では、Task 番号を指定することになる。これらの Task は SPE 上で、Memory Segment Manager によって管理される。

14. 比較

ここでは、SPURS Engine、Open CL、並列処理言語である Erlang/Scala と比較してみる。

Cerium は、SPURS Engine の実装の一つであるが、SPURS Engine 自体の情報が公開されていないので

比較することは難しい。Cerium は、sourceforge.jp 上で公開されている。

Cerium では、SPE Task の終了を Mail により PPE に投げて、PPE 側で Task Queue の管理を行っている。Word count のような場合は、それが負荷になる場合もある。SPURS Engine 等で、どのような工夫が行われているのかは未知数である。

Open CL は、Task を登録する代わりに、文字列として渡し、Open CL 側で、GPGPU や Thread に展開される。Mac OS X では、llvm⁸⁾ を用いて展開されている。Cerium では、前もってコンパイルされた関数をマクロにより表に登録する仕組みである。

Kernel は、

```
__kernel void UniformAddKernel(
    __global float *output_data,
    __global float *input_data,
    __local float *shared_data,
    const uint group_offset,
    const uint base_index,
    const uint n)
{
    const uint local_id = get_local_id(0);
    const uint group_id = get_global_id(0)
        / get_local_size(0);
    const uint group_size = get_local_size(0);
```

と言う形を持っている。global がメインメモリ上のデータで、local が Local Store のデータである。

データの load/store は、Kernel として明示的に記述し、

```
unsigned int k = PRESCAN_NON_POWER_OF_TWO;
clSetKernelArg(ComputeKernels[k],
    a++, sizeof(cl_mem), &input_data);
clEnqueueNDRangeKernel(ComputeCommands,
    ComputeKernels[k], 1,
    NULL, global, local, 0, NULL, NULL);
```

等として、自分で Queue を管理する。実行 Queue に自分で格納する Open CL の方がやや複雑な記述となる。

Kernel 間の依存関係は、Queue で解決されているので、起動するメインルーチン側で処理することになる。

Kernel の中では、__global という形で、いつでもメインメモリにアクセスすることが可能である。なので、明示的な DMA は必要ない。しかし、見えないだけでコストや待ち時間は生じてしまう。そこで、Pre Scan のような形で、データを __local に前もって持ってきておく必要がある。

14.1 Erlang, Scala

Erlang と Scala は、Actor に似た感じで並列処理を行う。Erlang は、Prolog に似た構文を持っており、

Scala は Java 上に実装されている。

双方の言語とも通信はチャンネルで行われて、Task 上のデータは関数型言語的な意味で変更されない。再帰的な関数呼び出しにより、Task の状態を作るので、両方とも似たような (構文は随分違うが) 並列プログラミングのスタイルを提供している。

Erlang

```
ping(N, Pong_PID) ->
    Pong_PID ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping(N - 1, Pong_PID).
```

Scala

```
class Counter extends Actor
{
    override def act(): Unit = loop(0)

    def loop(value: Int): Unit = {
        receive {
            case Incr() => loop(value + 1)
            case Value(a) => a ! value; loop(value)
            case Lock(a) => a ! value
        }
        receive { case UnLock(v) => loop(v) }
        case _ => loop(value)
    }
}
```

Cerium では、データの引き渡しは、void * に cast するので、型の安全性を言語上で保証することは出来ない。Open CL でも状況は同じで、

```
__kernel void UniformAddKernel(
    __global float *output_data,
    __global float *input_data,
    __local float *shared_data,
    const uint group_offset,
    const uint base_index,
    const uint n)
```

が、

```
err |= clSetKernelArg(ComputeKernels[k],
    a++, sizeof(cl_mem), &output_data);
err |= clSetKernelArg(ComputeKernels[k],
    a++, sizeof(cl_mem), &partial_sums);
err |= clSetKernelArg(ComputeKernels[k],
    a++, sizeof(float), 0);
err |= clSetKernelArg(ComputeKernels[k],
    a++, sizeof(cl_int), &group_offset);
err |= clSetKernelArg(ComputeKernels[k],
    a++, sizeof(cl_int), &base_index);
err |= clSetKernelArg(ComputeKernels[k],
    a++, sizeof(cl_int), &n);
err |= clEnqueueNDRangeKernel(ComputeCommands,
```

```
ComputeKernels[k], 1, NULL,  
    global, local, 0, NULL, NULL);
```

などに相当することになる。これらの型を (IDE 等で) チェックすることは、難しくはないが、Scala のような言語自体がチェックする方が望ましい。

Erlang, Scala は、GC を持っているが、Open CL/Cerium では、明示的なメモリ管理を行う必要がある。

パイプライン処理をうまく動作させるには、Task の列 (Queue) を作る必要があるが、Erlang, Scala では、その列を明示的に作ることは強制されていない。分散処理、あるいは、Thread による並列処理には、Erlang, Scala が適しているが、Many Core では、さらになんらかの構文的なサポートが必要だと考えられる。

15. まとめ

Many Core でのプログラムは、科学技術計算等に重要であるが、Desktop PC や、Note PC での性能向上に使用するには、「普通の」プログラムでも並列処理を恒常的に行う必要がある。

Cerium は、Open CL と同様に、プログラム Task に分割し、Core に投入することによって実行する。

Cerium は、Open CL よりは若干ましな記述が可能となっている。

Many Core ではコピーが頻繁に行われ、そのコピーを隠すパイプライン実行が重要である。コピー自体は必須となる。

プログラムのチューニングとデバッグは自明ではない。

今後は、Task の単位として、Continuation based

C の code segment を用いた実装を行う予定である。

参考文献

- 1) Corporation, S.: Cell Broadband Engine Architecture (2005).
- 2) Tungsten Graphics: Mesa / Gallium Cell Driver (2009).
- 3) Inoue, K.: SPU Centric Execution Model (2006).
- 4) Munshi, A.: *The OpenCL Specification Version: 1.0*, Khronos OpenCL Working Group (2009).
- 5) IBM, SCEI, T.: SPE Runtime Management Library Version 2.3 (2008).
- 6) IBM, SCEI, T.: C/C++ Language Extensions for Cell Broadband Engine Architecture Version 2.6 (2008).
- 7) LLC, G.G.: Simple DirectMedia Layer.
- 8) Lattner, C. and Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California (2004).

(平成 2009 年 11 月 30 日受付)

(平成 0 年 0 月 0 日採録)

河野 真治 (正会員)

1959 年生。1989 年東京大学大学院情報工学課程修了 (工学博士) 同年 Sony Computer Science Laboratory, Inc. 入社。1996 年より琉球大学工学部准教授工学博士。ACM

会員。