CPUとGPUの性能比較
～ 行列計算およびN体問題を用いて ～

大阪工業大学 情報科学部 コンピュータ科学科
学生番号 Q07-065
富久友樹

2011 年 2 月 17 日

目 次

1 序論

1.1 背景

GPU とは Graphics Processing Unit の略称であり、グラフィックボードに搭載されている 3D グラフィックスに必要な計算処理を行うプロセッサのことである。。

近年、GPU の演算能力を画像処理以外の目的に活用することが進められている。なぜならば、GPU は高い並列計算能力を保持しているとされており、一度に大量のデータを単純な計算で処理する場合、劇的な効果が得られるとされているからである。また、GPU は CPU に比べコストパフォーマンスが格段に良いのも注目されている理由である。

1.2 本研究の目的・概要

本研究の目的は、科学的数値計算において、従来の CPU を用いるシミュレーションに対し、どの程度 CPU と GPU の性能が違うかを調べることである。

本研究では科学的数値計算として N 体問題と行列の計算として n 行 m 列の成分が n+m-1 の正方行列（サイズ N*N）の積の計算を用いる。N 体問題とは宇宙空間に質点を N 個ばらまいたとき、万有引力で互いに相互作用し合い、それによってどのように変化していくのかをシミュレートする問題である。行列計算は、整数値計算で単純に計算量が見積もれるモデルとして、N 体問題は実際のシミュレーションとして、浮動小数点処理の効率を見積もるモデルとして考えることにした。また、N 体問題を解くために、4 次の Runge-Kutta 法を用いて CPU と GPU の性能比較を行う。開発言語は C 言語と CUDA を用いる。C 言語で CPU の性能、CUDA で GPU の性能を調べる。

1.3 本研究の実行環境

表 1: 実行環境

マシン名	einstein
OS	Vine Linux5.1 64bit
CPU	Core2 Quad Q9650 3.00GHz
メモリ	8GB
GPU	GTX285
GPU メモリ	1GB
ストリーミングプロセッサ数	240 基

本研究では、この einstein を使っていく。

2　基本用語

本研究では GPU や CUDA など専門的な用語が多数出てくるが、参考のため、この章ではその用語について説明していく。

2.1　GPU

2.1.1　GPU について

GPU は Graphics Processing Unit の略称であり、グラフィックボードに搭載されている 3D グラフィックスの表示に必要な計算処理を行うプロセッサのことである。また、GPU は高い並列処理能力を持っている。並列処理能力とは、複数のプロセッサに処理を分散して、同時に演算を行うことで演算速度を速くすることができる能力である。例えば、本研究で使用した GPU は“ GeForce GTX285（2009 年製）”では [1] によると、1062.72GFLOPS もの演算能力を有している。この演算能力は非常に高く、本研究で使用した CPU（Core2 Quad CPU Q9650）は [2] によると、48GFLOPS なのでおよそ 22 倍にもなる。この能力をグラフィック描画だけでなく汎用の数値計算にも利用でき、コストパフォーマンスが非常に良い。また、DirectX 9.0 の登場（2002 12/20）以降、NVIDIA の「CUDA」や AMD の「ATI Stream」などが登場し、GPGPU 活用の幅が広がったことによって注目が集まっている。

2.1.2　CPU との違い

CPU はプログラムにより各装置の制御や数値計算などができる。しかし、数値計算もできるが画像処理など大量に演算が必要なものを処理させると莫大な時間がかかってしまう。例えば、本研究で行った性能比較テストでは最大で 3000 倍以上 CPU のほうが遅くなることもあった。

また、CPU は汎用性が高く、複雑な命令でも処理出来るように設計されており、かつ GPU に比べ大量生産ができないため、どうしても高価なものになってしまう。例えば、本研究で使用した CPU と GPU では GPU 側の最新データが見つからなかったため、少し前になるが 2010 年 8 月現在、CPU は最安値で 31231 円、GPU は最安値で 29800 円であった。それぞれ性能は理論値ではおよそ 22 倍、GPU のほうが良いが、値段では CPU のほうが少し高い。

2.1.3　近年の動向

近年、GPU はスーパーコンピュータにも数多く搭載され、2011 年 2 月現在、世界最速と言われている中国の天河一号 A にも NVIDIA 社の Tesla M2050 が 7168 個搭載されている。また、3D ゲームなどの普及により高性能な GPU を求める一般ユーザーも増えている。

2.2 CUDA

2.2.1 CUDA について

NVIDIA 社が提供する GPU 向けの C 言語の統合開発環境であり、コンパイラやライブラリなどから構成されている。ただし、CUDA で GPU 向けのプログラムを開発するには NVIDIA 社のグラフィックボードと対応している OS（Windows XP,Vista,7/Fedora 7 以降/OpenSUSE 10.1 以降/Ubuntu 7.04 以降/Mac OS X 10.5.2 以降など）が必要である。また、Windows で CUDA を使用してプログラムを開発するには Microsoft が提供する Visual Studio(Visual C++) を使用するのが一般的である。Visual Studio Express Edition は無償で提供されており、Microsoft の Web サイトからダウンロードできる。

開発ツールキットは http://developer.nvidia.com/object/cuda_3_2_toolkit_rc.html よりダウンロード可能である。

本研究では Linux 64bit のバージョン 3.2 を使用

2.2.2 CUDA の応用分野

近年、科学技術計算や金融分野、グラフィック分野など演算に時間がかかる分野で利用されている。例えば、今まで科学技術計算で高速に演算を行うために、演算しようとしている科学技術計算に特化したスーパーコンピュータが使われていた。だが、そのスーパーコンピュータを作るには莫大な費用がかかりコストパフォーマンスが非常に悪かった。そこで、安価で高速に演算を行うことができる GPU に着目し、CUDA を用いることで GPU による演算を行えるようにし、コストパフォーマンスを格段に良くした。

2.2.3 CUDA に対応したソフトウェア

近年、CUDA に対応したソフトウェアが実用化され製品として売りだされている。例えば、Adobe Photoshop CS4 (Adobe)、PowerDirector (CyberLink)、VideoStudio Pro X3 (COREL)、LoiLo-Touch (LoiLo) などがある。これらは画像処理や動画編集のソフトである。また、NVIDIA 社からも Badaboom Media Converter という動画編集ソフトが出ている。2010 年 12 月現在、確認できた唯一のフリーソフトウェアとしては、MediaCoder という動画編集ソフトが存在した。

現時点では、CUDA に対応したソフトウェアは動画や画像を編集するソフトがほとんどであるが、Mathematica という数式処理システムが Ver8 で対応すると言われている。このように、近々演算に時間がかかるソフトウェアで活用されるのではないかと予想される。

2.3 並列処理

並列処理とは複数のマイクロプロセッサなどに処理を分散して、同時に演算を行うことで演算速度を速くする技術のことをいう。ただし、並列処理をすれば必ずしも演算速度が速くなるというわけではない。並列処理において最もパフォーマンスを発揮するのは、複数のプロセッサが全ての力を 100 %使いきり、プロセッサ間の通信を極力少なくする場合である。また、並列の効率を上げるためには、実行順番に依存しないプログラム又はアルゴリズムが必要である。

しかしながら、このようなプログラムはほとんど存在しないのも事実である。よって、並列処理に向いていないプログラムでも出来る限りパフォーマンスを良くするためにアルゴリズムを考えて効率よく並列処理を行わなければならない。そうしなければ、かえって遅くなることもありうるからだ。

例えば、図のように A1～Z100 までのタスクがあるとする。A～Z は実行順番に依存せず、1～100 は実行順番に依存する。また、プロセッサはすべての力を使いきれると仮定する。

まず、並列処理がイメージしやすいように並列処理とは反対の処理法、逐次処理について説明する。逐次処理とは、

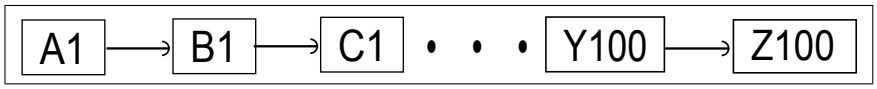

図 1: 逐次処理イメージ図

図のように、一つのプロセッサでデータが保存された順番で処理を実行する。今回の図では A～Z,1～100 の処理を実行するので 26 × 100 個の処理が実行され、26 × 100 個分の処理時間がかかってしまう。

一方、並列処理では、例えば図 2 のように、複数のプロセッサで同時に処理を実行する。今回の図では A～Z,1～100 の処理を実行するので 26 × 100 個の処理が実行されるが、同時に A～Z の処理を行っているので 100 個分の処理時間しかかからないことになるので、逐次処理の 26 倍もの速さで処理が行えることになる。

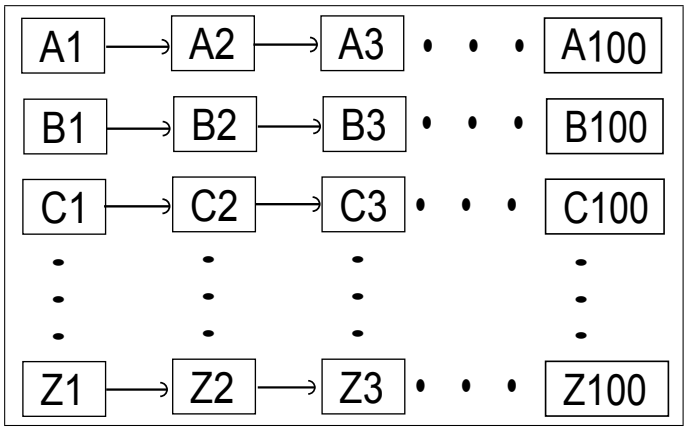

図 2: 並列処理イメージ図

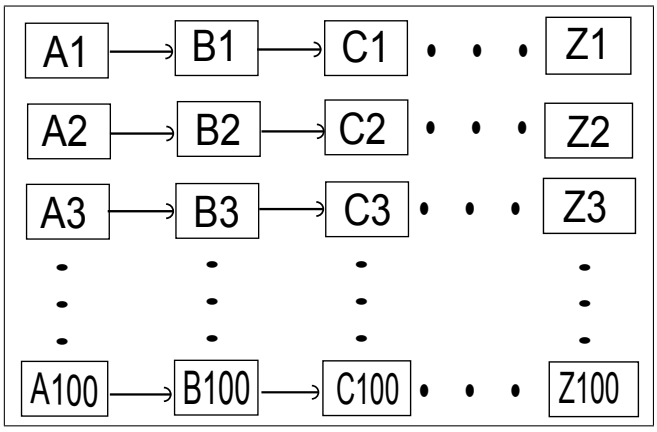

図 3: 並列処理イメージ図 2

図 3 中でも図 2 と同様に、複数のプロセッサで同時に処理を実行している。しかし、今回の仮定では A〜Z は実行順番に依存しないが、1〜100 は実行順番に依存するので、例えば A2 の演算をするには A1 の演算結果が必要になるので並列処理の恩恵をあまり受けられず、図 2 の場合より演算速度が遅くなると予想される。

このようにただ単純にデータを分散して並列処理を行えばいいというものではないのである。並列処理では出来る限りパフォーマンスが発揮できるように実行順番やアルゴリズムなどを工夫して並列処理を行えば、格段に演算速度をあげることもできるのである。

3　プログラミングの特徴

3.1　CUDA のプログラミングモデル

GPU を用いたプログラミングでは演算処理単位を考えたコードを書く必要がある。CUDA の概念上、一つのデータを処理する単位を「スレッド」という。また、同じサイズのスレッドをまとめたものを「ブロック」と呼び、最大 3 次元のスレッドの配列をブロックとすることができる。そして、同じサイズのブロックをまとめたものを「グリッド」と呼び、最大 2 次元のブロックの配列をグリッドとすることができる。このグリッドがホストから実行を指令する単位で、グリッド内の全スレッドはカーネル（デバイス上で実行する機能）を実行する。

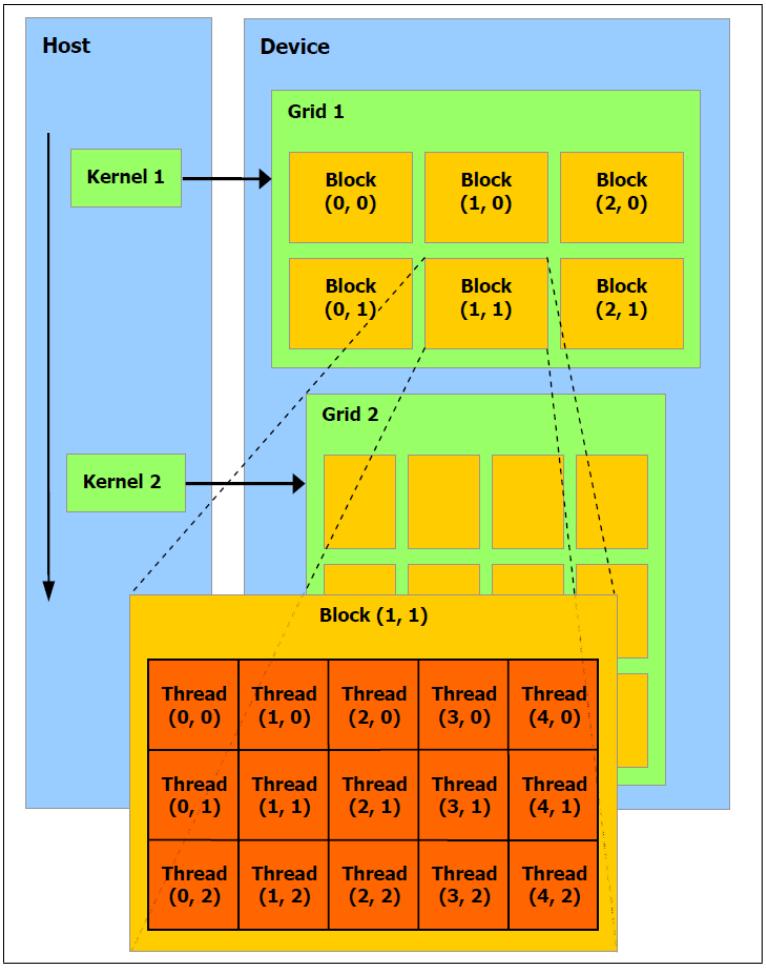

図 4: CUDA のプログラミングモデル (出典：NVIDIA の CUDA プログラミングガイド)[3]

図 4 は、Host(CPU) 側から Kernel(GPU で実行させるプログラム) を呼び、Device(GPU) でそのプログラムを実行させる、という流れについて描いているのである。また、図 4 中の Block(0,0) や Thread(0,0) はそれぞれ 2 次元でブロックとスレッドが生成されていることを表している。

4.2 節で実際のプログラムを用いて、より詳しい流れを説明する。

3.2　CUDA のメモリモデル

各スレッドはレジスタとローカルメモリを持っており、さらに同一ブロック内のスレッド間でデータの共有ができるようにシェアードメモリというものを持っている。そして、同一グリッド内のブロック間でグローバルメモリとコンスタントメモリ、テクスチャメモリを持っている。コンスタントメモリとテクスチャメモリとは、キャッシュメモリを所有しており、ホスト側からのアクセスは書き込み、GPU 側からのアクセスは読み込みしかできないメモリのことである。CUDA でプログラムを書くとき、基本的にはグローバルメモリとシェアードメモリについて考えながら書く。グローバルメモリは CPU 側からのデータ入力、GPU 側からの計算結果の出力に使われ、演算にも用いることができる。低速で大容量なのが特徴である。一方、シェアードメモリはグローバルメモリに比べて非常に高速な演算が可能である。複数のプロセッサで共有し、アクセスが高速だが容量は小さいのが特徴である。GPU 上で演算させる場合、アクセスが高速に行えるシェアードメモリにデータを移してから演算させる方が当然、高速に演算を行うことができる。

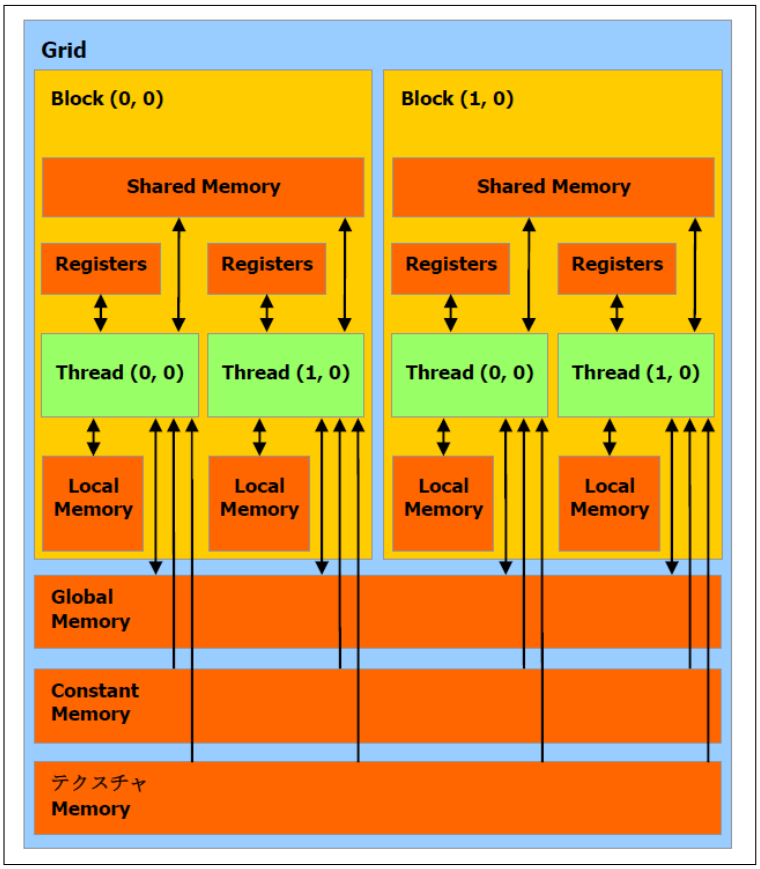

図 5: CUDA のメモリモデル (出典：NVIDIA の CUDA プログラミングガイド)[3]

この図について補足すると、この図は同じグリッド内での各ブロック内のスレッドがどのメモリにアクセスできるのか、また、どのメモリがスレッドにアクセスできるかについて描いているのである。まず、グローバルメモリやコンスタントメモリ、テクスチャメモリから伸びて

いる矢印付きの線を見てほしい。この三つのメモリはブロックに関係なく同じグリッド内なら全てのスレッドへアクセスできる。しかし、スレッド側からこの三つのメモリに伸びている矢印はグローバルメモリにしかない。これは、スレッド側からはグローバルメモリにしかアクセスは出来ないことを意味している。同様に、ローカルメモリとレジスタは各スレッドごとにお互いにアクセスでき、あるスレッドから他のスレッドのローカルメモリやレジスタにはお互いにアクセスできない。シェアードメモリは各ブロック内で共有できるので、同じブロック内のスレッド同士ならデータを共有できる。

3.3 プログラミング

実際にプログラミングを行うとき、スレッドやブロックをいくつ生成する必要があるのか、どのメモリを使えばより速い演算を可能にするのかなどを考える必要がある。

4　行列計算

4.1　行列計算について

本研究では、まず、実際にどの程度 CPU と GPU では演算速度が違うかを検証するために行列計算を用いて性能比較を行った。行列計算として n 行 m 列の成分が n+m-1 の正方行列（サイズ N*N）の積を計算した。今回、行列計算を用いた理由は要素ごとに計算を分解することが容易で、演算速度の違いを比較するのに向いていると考えたからである。

例えば、行列計算とは数式 4.1 のように、

$$\begin{aligned}\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}\begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} &= \begin{pmatrix} 1*5+2*7 & 1*6+2*8 \\ 3*5+4*7 & 3*6+4*8 \end{pmatrix} \\ &= \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix} \end{aligned} \tag{4.1}$$

計算するものである。また、要素ごとに計算を分解することが容易というのは、例えば数式 4.1 の答えの 2 行 1 列目にある 22 を求めるとき、1 行 1 列目の答え 19 は不必要なので、19 を求める計算と 22 を求める計算を同時にしても正しい答えを得ることができる。このように、要素ごとに計算できるので並列計算に向いていると考えた。

4.2　行列計算プログラムの実例

CUDA でのプログラミングの流れと演算速度の違いを把握するために、Interface 誌のサイト [4] からプロジェクト (CQ_CUDA_matrix) を用いた。http://www.cqpub.co.jp/interface/download/contents.htm の 2008 年 8 月号よりダウンロードができる。

本研究では、二つのカーネル (グローバルメモリ、シェアードメモリ) を使用するうえで、プログラミングの仕方の違いと演算速度がどの程度違うのかということに調べていく。この節では、二つのカーネルのプログラミングの仕方の違いに焦点を当てていく。

まずは、ホスト (CPU) 側のソースコードを見ていく。

リスト 1: ホスト側

```
1  #include <stdio.h>
2  #include <cutil.h>
3
4  //マクロ宣言
5  #define BLOCK 16
6  #define WIDTH 512
7
8  //プロトタイプ宣言
9  void Host(float *a, float *b, float *c);
10 __global__ void Kernel1(float *A, float *B, float *C);
11 __global__ void Kernel2(float *A, float *B, float *C);
12
13 //行列データと計算結果格納用配列
14 float h_a[WIDTH*WIDTH];
15 float h_b[WIDTH*WIDTH];
16 float h_c[WIDTH*WIDTH];
17
```

```
18 //メイン関数
19 int main()
20 {
21   int i;
22   unsigned int timer;
23
24   //GPUの初期化
25   CUT_DEVICE_INIT();
26
27   //GPU上にメモリを確保(1)
28   float *d_a, *d_b, *d_c;
29   cudaMalloc((void**) &d_a, sizeof(float)*WIDTH*WIDTH);
30   cudaMalloc((void**) &d_b, sizeof(float)*WIDTH*WIDTH);
31   cudaMalloc((void**) &d_c, sizeof(float)*WIDTH*WIDTH);
32   cudaMemset(d_c, 0, sizeof(float)*WIDTH*WIDTH);
33
34   //行列データ作成
35   for(i=0; i<WIDTH*WIDTH; i++){
36     h_a[i]=(float)i;
37     h_b[i]=(float)i;
38   }
39
40
41   //変数をGPU上のメモリへコピー(2)
42   cudaMemcpy(d_a, h_a, sizeof(float)*WIDTH*WIDTH, cudaMemcpyHostToDevice);
43     cudaMemcpy(d_b, h_b, sizeof(float)*WIDTH*WIDTH, cudaMemcpyHostToDevice);
44
45   //ブロックとグリッドの定義(3)
46     dim3 grid(WIDTH/BLOCK, WIDTH/BLOCK, 1);
47     dim3 threads(BLOCK, BLOCK, 1);
48
49   //カーネル起動(4)
50   Kernel1<<< grid, threads >>>(d_a, d_b, d_c);
51   //Kernel2<<< grid, threads >>>(d_a, d_b, d_c);
52
53   //計算結果を取得(5)
54   cudaMemcpy(h_c, d_c, sizeof(float)*WIDTH*WIDTH,cudaMemcpyDeviceToHost);
55
56   printf("GPU計算結果 = %f\n",h_c[WIDTH*WIDTH-1]);
57
58   //GPU上のメモリを解放(6)
59   cudaFree(d_a);
60   cudaFree(d_b);
61   cudaFree(d_c);
62
63   //ホスト側での計算（比較用）
64   Host(h_a, h_b, h_c);
65   printf("ホスト計算結果  = %f\n",h_c[WIDTH*WIDTH-1]);
66
67 }
```

CUDA でのプログラムの流れは、

- デバイス (GPU) の初期化
- デバイス (GPU) 上にメモリを確保
- ホスト (CPU) 側で演算するデータを生成
- ホスト (CPU) からデバイス (GPU) メモリへデータを移行
- スレッド数とブロック数を必要な分だけ指定
- カーネルを実行
- ホスト (CPU) がデバイス (GPU) からデータを回収

- デバイス (GPU) メモリを解放

という流れである。ここから重要な部分について補足する。
(1)GPU 上のグローバルメモリに行列データを格納する領域をポインタを使用して動的に確保する。この確保されたメモリの中のデータで GPU は演算を行う。
(2) デバイス (GPU) 側からコンピュータ上のメインメモリには直接アクセスできない。そのため、ホスト (CPU) 側に用意したデータを (1) で確保したデバイス (GPU) メモリへ移行させる必要がある。
(3) スレッド数とブロック数を必要な分（メモリに確保した大きさ分）だけ指定。必要な分とは、例えば今回のプログラムでは WIDTH*WIDTH 分のメモリを確保したので、スレッド数×ブロック数がこの WIDTH*WIDTH 分になるように指定する。多くても少なくてもバグの原因になることがあるので、必ず必要な分だけを指定する。
(4) カーネルに必要なスレッド数、ブロック数と (2) でデバイス側に確保したデータを転送して、デバイス側で演算させる。
(5) ホスト側からグローバルメモリには直接アクセスできないので、(4) の計算結果をホスト側へ移行させる。
(6)GPU に関する作業が終わったら (1) で確保したメモリを解放する。解放をしないままだとグローバルメモリにデータが格納されたままになる可能性があるので必ず解放する。
次に、デバイス (GPU) 側のソースコードを見ていく。

リスト 2: デバイス側 (グローバルメモリ)

```
__global__ void Kernel1(float *A, float *B, float *C)
{
  //GPUでの行列乗算（グローバルメモリのみ使用）
  int x=blockIdx.x*blockDim.x + threadIdx.x;(1)
  int y=blockIdx.y*blockDim.y + threadIdx.y;(2)
  float tmp=0.0;

  for(int k=0; k<WIDTH; k++){
    int row=k+y*WIDTH;
    int col=x+k*WIDTH;
    tmp+=A[row]*B[col];
  }

  C[x+y*WIDTH]=tmp;
}
```

まずは、グローバルメモリを使用したプログラムを見ていく。
(1),(2) はそれぞれ x,y のスレッド Id を計算している。
・blockIdx：ブロックのインデックスを持っている。インデックスとは各次元で何番目のブロックなのかを指している。例えば 2 次元のブロックが生成されているとき、(2,5) は blockIdx.x=2,blockIdx.y=5 となり、これによってブロックが指定できる。
・blockDim：ブロックの次元数を持っている。次元数とは x,y がそれぞれ何次元なのかを指している。
・threadIdx：スレッドのインデックスを持っている。blockIdx と説明がほとんど同じなので説明は省く。

リスト 3: デバイス側 (シェアードメモリ)

```
__global__ void Kernel2(float *A, float *B, float *C)
{
  //GPUでの行列乗算（シェアードメモリを使用）
  int bx = blockIdx.x;
  int by = blockIdx.y;
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  float tmp = 0;

  __shared__ float As[BLOCK][BLOCK];(1)
  __shared__ float Bs[BLOCK][BLOCK];(2)

  for (int a = 0, b = 0 ; a < WIDTH; a += BLOCK, b += BLOCK) {

    int a_adr = WIDTH * BLOCK * by + a;
    int b_adr = BLOCK * bx + WIDTH * b;

    As[ty][tx] = A[a_adr + WIDTH*ty + tx];
    Bs[ty][tx] = B[b_adr + WIDTH*ty + tx];
    __syncthreads();(3)

    for (int k = 0; k < BLOCK; k++) {
      tmp += As[ty][k] * Bs[k][tx];
    }
    __syncthreads();
  }

  int adr = WIDTH * BLOCK * by + BLOCK * bx;
  C[adr + WIDTH * ty + tx] = tmp;

}
```

(1),(2) がシェアードメモリを使うための宣言である。この宣言があることによりシェアードメモリを使い演算の高速化が可能になる。他の部分は、基本的にはグローバルメモリと一緒である。シェアードメモリ使うためにスレッド Id の指定の仕方が変わっただけである。しかし、(3) はグローバルメモリにはなかったものでこれは同じブロック内のスレッドを同期させるためのものである。CUDA でプログラムを書くと非同期にスレッドが動いていくので必要ならばこのように同期させる必要がある。

次の節では、実際にどの程度二つのカーネルで演算速度が違うのかを調べていく。

4.3 行列計算による性能比較

4.3.1 スレッド数による性能比較

CUDA でプログラミングする上でスレッド数やブロック数などを考えることはとても重要である。ここでは実際に一つのブロックに格納されているスレッド数を変化させるとどのように演算速度が変化するのかを調べる。ただし、このスレッド数の増減によって、予め用意しておいたスレッド数が変わるわけではない。全体のスレッド数は一つのブロック内のスレッド数×ブロック数によって決まるのでそれぞれの値が変わるだけである。

G_ global_ 1 のように末尾についている数字がスレッド数を表している。今回使用したプログラムではスレッド数を 2 次元で生成しているので、例えば末尾についている数字が 1 ならば 1 × 1 でスレッド数は一つのブロック内に 1 個あるということになり、末尾が 2 ならば 2 × 2 でスレッド数は一つのブロック内に 4 個あるということになる。これは G_ shared_ 1 についても同じである。数字の前の G_ global はグローバルメモリを使用して演算したこと示し、G_ shared

はシェアードメモリを使用して演算したことを示している。

図 6,7 中では演算量で示しているので、行列の大きさと演算量の関係は、

表 2: 行列の大きさと演算回数の関係

行列の大きさ	演算回数
16	7,936
32	64,512
64	520,192
128	4,177,920
256	33,488,896
512	268,173,312
1024	2,146,435,072

である。

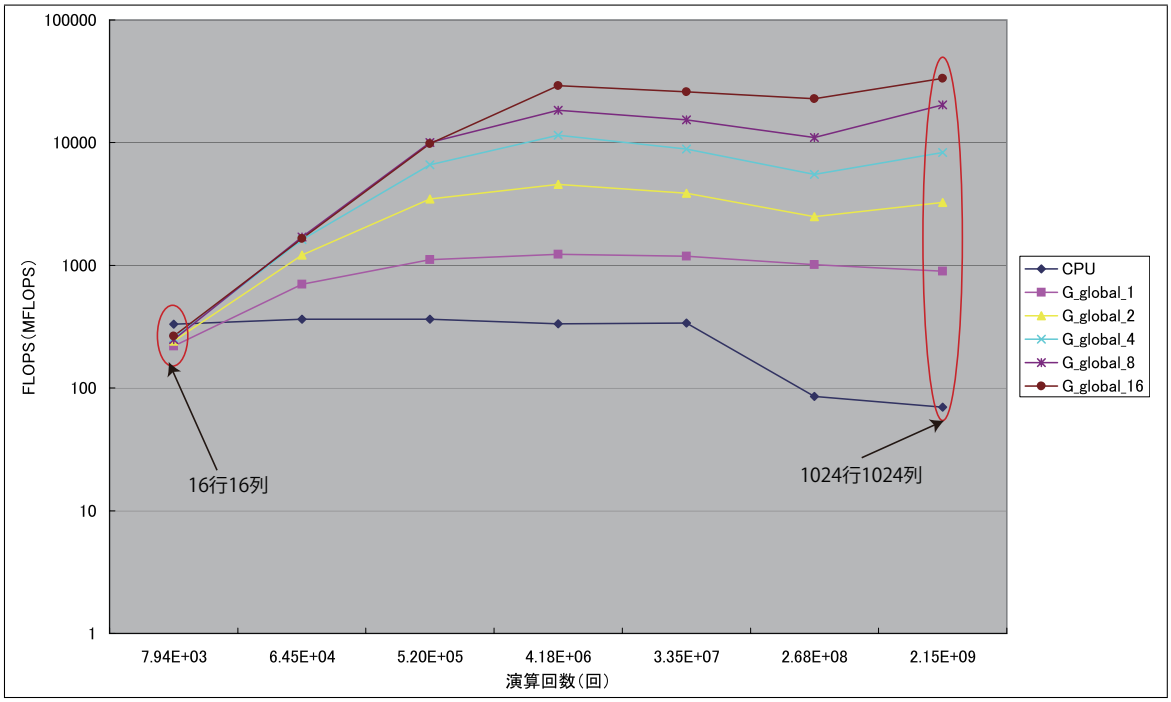

図 6: グローバルメモリのスレッド数による性能比較

図 6 はグローバルメモリを使用したときの一つのブロック内のスレッド数を変化させていったときの性能比較をしている。CPU による計算に対してグローバルメモリを使用した GPU では一つのブロック内のスレッド数が各次元に対して 1 個で N=16 のときに 0.67 倍 (最小値)、一つのブロック内のスレッド数が各次元に対して 16 個で N=1024 のときに 479.1 倍 (最大値) 計算時間が短縮できた。

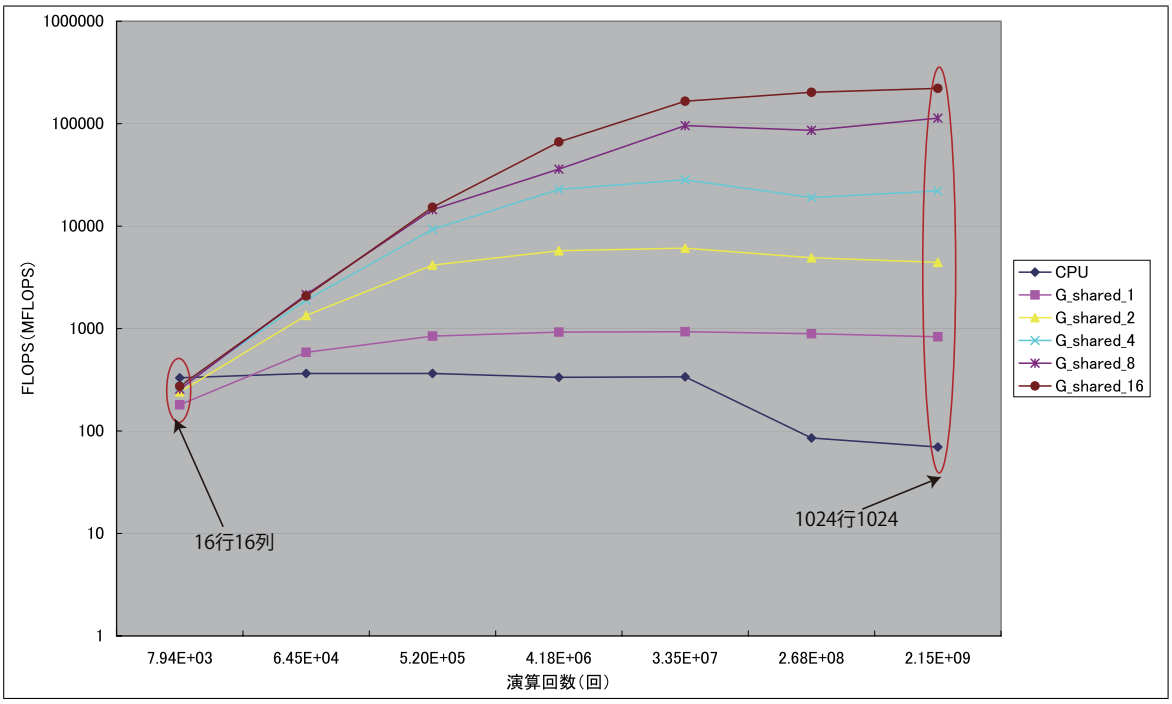

図 7: シェアードメモリのスレッド数による性能比較

図 7 はシェアードメモリを使用したときの一つのブロック内のスレッド数を変化させていったときの性能比較をしている。CPU による計算に対してシェアードメモリを使用した GPU では一つのブロック内のスレッド数が各次元に対して 1 個で N=16 のときに 0.55 倍 (最小値)、一つのブロック内のスレッド数が各次元に対して 16 個で N=1024 のときに 3167 倍 (最大値) 計算時間が短縮できた。

図 6,7 中の各スレッド数ごとのグラフを見てみると、グローバルメモリ、シェアードメモリともに一つのブロック内に生成するスレッド数によってグローバルメモリでは一つのブロック内のスレッド数が各次元に対して 1 個と 16 個のときに最大で 37 倍、シェアードメモリでは一つのブロック内のスレッド数が各次元に対して 1 個と 16 個のときに最大で 266 倍ほど演算速度が違っている。このことからブロックを多く生成するよりも一つのブロック内に生成するスレッド数を多く生成したほうが計算時間の短縮ができるとわかる。

演算速度がこれだけ違う理由は、メモリへのアクセス速度が関係していると考えられる。ブロックが多くなるとグローバルメモリなどアクセス速度が遅いほうにアクセスする回数が増えてしまう。その結果、演算速度が遅くなったのだと考えられる。

4.3.2 最適化した状態の性能比較

図 8 中では、GPU_ global はグローバルメモリを使用して演算したこと示し、GPU_ shared はシェアードメモリを使用して演算したことを示している。それぞれ最適化されたスレッド数を使用している。本研究の性能比較の結果、N=16 の時には CPU の方が演算速度が速いので GPU の演算速度が必ず CPU より早くなるとは限らない。また、GPU は N=256 を超えたあたりから演算速度の伸びが悪くなる。よって、実測値は理論値のようにはいかないことがわかった。

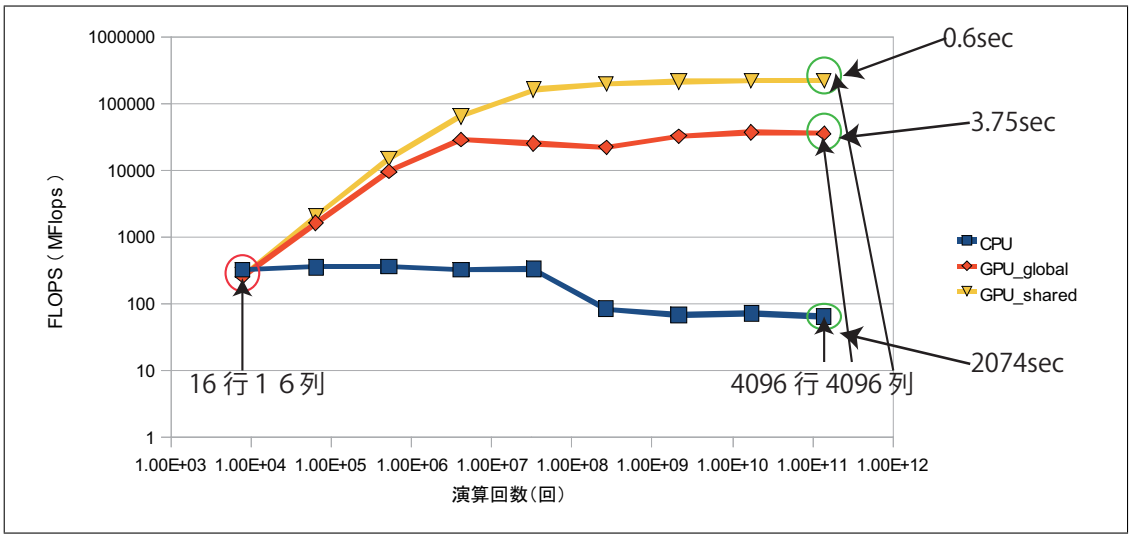

図 8: 行列計算による性能比較図

4.4 性能比較のまとめ

GPU の演算速度が CPU の演算速度を必ず上回るわけではない。上回るにはある程度演算回数が多くなくてはいけない。今回試した行列計算問題では 32 × 32 以上の行列計算で上回った。GPU をうまく使えれば、科学的数値計算において、大幅な演算時間の短縮が可能になると考えられる。

ただし、プログラミング上の工夫やメモリ使用のチューニングが必要である。最適なチューニングとは、一つのブロック内のスレッド数をできるだけ多くし、シェアードメモリをうまく使うことである。

5　N体問題

5.1　N体問題について

N体問題は物理学で相互作用するN個の質点からなる運動を規定する問題で、万有引力で互いに相互作用し合い、それによってどのように変化していくのかを数値解析を利用して求める問題である。また、二体問題までは厳密に解けることはできるが、三体問題以上になると一般的には解析的には解けない。ただし、例外として制限三体問題では解が存在する。本研究では、3次元のN体問題について考える。

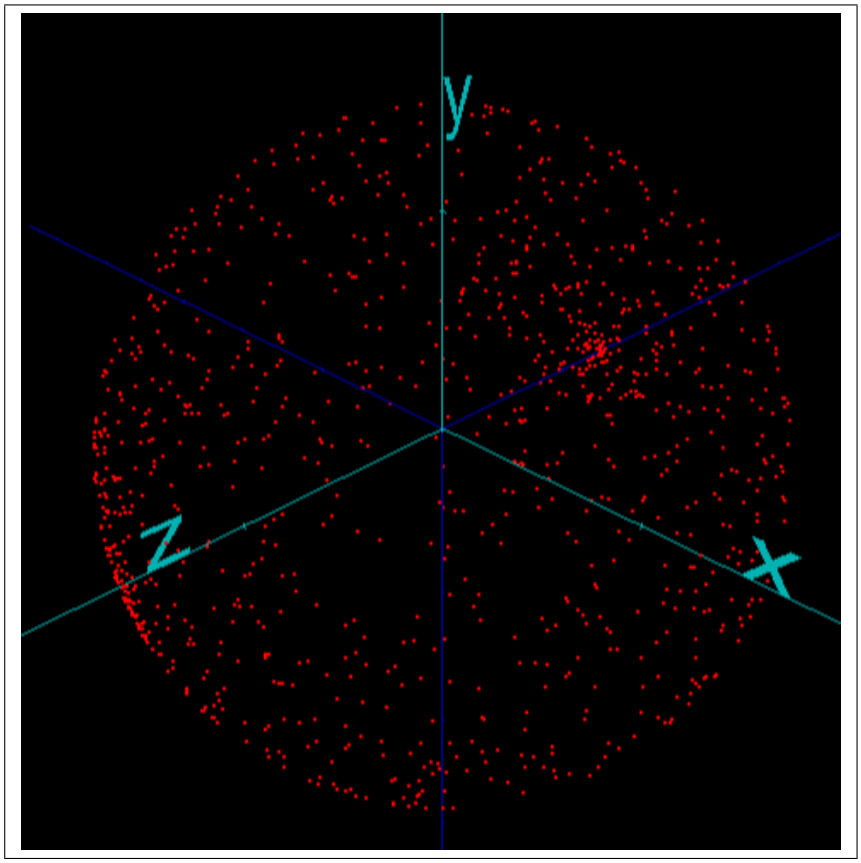

図 9: 1024 体を球状に配置した図

このように3次元にN体の質点を配置し、万有引力でN体の質点がどのように動いていくのかを数値解析で求めていく。

5.2　ニュートンの運動方程式

ニュートンが定めた運動法則は、

・慣性の法則

力を加えなければ、物体は静止したままか、等速直線運動を行う。
・運動方程式
物体に力 F を及ぼすと、物体の質量 m に反比例した加速度 a が生じる
・作用反作用の法則
物体に力 F を及ぼすと、その物体は同じ大きさで逆向きの反作用 $-F$ を作用物体に及ぼす。

の三つである。以下、運動方程式について述べる。

質量 M と m の天体の距離を r とし、質量 M の物体から受ける万有引力で、質量 m の物体の運動を考えると、以下の式になる。

$$m\frac{d^2r}{dt^2} = -G\frac{Mm}{r^2} \tag{5.1}$$

G は万有引力定数である。なお、式 (5.1) は二体問題であり、本研究では N 体問題を扱う。では実際に n 個の星があるとき、i 番目の星（質量 m_i）の運動方程式は以下のようになる。

$$m_i\frac{d^2r_i}{dt^2} = -\sum_{j=1}^{n} G\frac{m_i m_j}{r_{ij}^2} \tag{5.2}$$

ここで、r_{ij} は i 番目と j 番目の惑星の重心間の距離である。ただし、$i \neq j$ で $i, j = 1, 2, 3, \cdots, n$ である。また、運動方程式は大きさだけでなく、向きも含んだ方程式である。本研究では x, y, z の三次元の運動方程式を解く。x, y, z それぞれの力の成分は以下のようになる。

$$m_i\frac{d^2x_i}{dt^2} = -\sum_{j=1}^{n} G\frac{m_i m_j}{r_{ij}^2}\frac{x_{ij}}{r_{ij}} \tag{5.3}$$

$$m_i\frac{d^2y_i}{dt^2} = -\sum_{j=1}^{n} G\frac{m_i m_j}{r_{ij}^2}\frac{y_{ij}}{r_{ij}} \tag{5.4}$$

$$m_i\frac{d^2z_i}{dt^2} = -\sum_{j=1}^{n} G\frac{m_i m_j}{r_{ij}^2}\frac{z_{ij}}{r_{ij}} \tag{5.5}$$

ここで、x, y, z, r は

$$x_{ij} = x_j - x_i \tag{5.6}$$

$$y_{ij} = y_j - y_i \tag{5.7}$$

$$z_{ij} = z_j - z_i \tag{5.8}$$

$$r = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2} \tag{5.9}$$

である。したがって、x, y, z の力の成分は

$$m_i \frac{d^2 x_i}{dt^2} = -\sum_{j=1}^{n} G \frac{m_i m_j (x_j - x_i)}{(\sqrt{(x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2})^3} \tag{5.10}$$

$$m_i \frac{d^2 y_i}{dt^2} = -\sum_{j=1}^{n} G \frac{m_i m_j (y_j - y_i)}{(\sqrt{(x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2})^3} \tag{5.11}$$

$$m_i \frac{d^2 z_i}{dt^2} = -\sum_{j=1}^{n} G \frac{m_i m_j (z_j - z_i)}{(\sqrt{(x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2})^3} \tag{5.12}$$

となる。

5.3 Runge-Kutta 法

Runge-Kutta 法は高精度な計算を簡単に行う方法として、数値積分の世界で用いられている常微分方程式の数値解法の一つである。Runge-Kutta 法は、Euler 法の要領で 1 ステップ Δx 進むとき、何回か推測値を出して加重平均をとり精度を上げるものである。

最も簡単な 2 次の Runge-Kutta 法は、微分方程式の

$$\frac{dy}{dx} = f(x, y) \tag{5.13}$$

を解くときに、点 (x_n, y_n) から次の点 $x_{n+1} = x_n + \Delta x$ における y_{n+1} の値を求める計算を、

$$y_{n+1} = y_n + \frac{1}{2}(k_1 + k_2) \tag{5.14}$$

$$\begin{aligned} \text{ただし} \quad k_1 &= \Delta x f(x_n, y_n) \\ k_2 &= \Delta x f(x_n + \Delta x, y_n + k_1) \end{aligned}$$

とする。

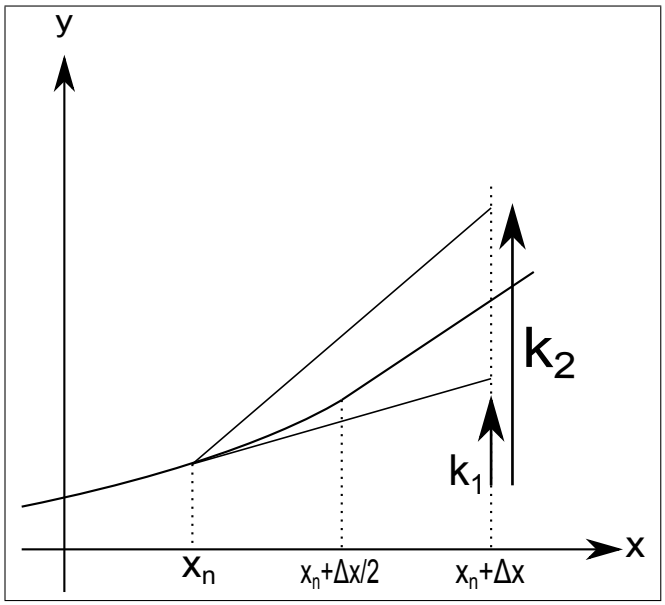

図 10: 2 次の Runge-Kutta イメージ図

k_1 は x_n での右辺を用いた y_{n+1} の推測値、k_2 は k_1 を利用した第 2 の推測値で、それらの平均をとる形になっている。このときの局所的な計算誤差は、$O((\Delta x)^3)$ であり、2 次精度といえる。

理論値には、一ステップを多段階に分割し、さらに各段階での y_n の予測値を加重平均で算出することが可能である。一般的な Runge-Kutta 法は、

$$y_{n+1} = y_n + \sum_{i=1}^{s} b_i k_i \tag{5.15}$$

$$\text{ただし} \quad k_i = \Delta x f(x_n + c_i \Delta x, y_n + \sum_{j=1}^{s} a_{ij} k_j)$$

とできる。a_{ij}, b_i, c_i は係数であり、段数と呼ばれる s と、これらの係数に応じて様々なスキームが可能になる。本研究で用いる 4 次精度の Runge-Kutta 法はよく使われ、次のようになる。

$$\begin{aligned} k_1 &= \Delta x f(x_n, y_n) \\ k_2 &= \Delta x f(x_n + \frac{\Delta x}{2}, y_n + \frac{1}{2}k_1) \\ k_3 &= \Delta x f(x_n + \frac{\Delta x}{2}, y_n + \frac{1}{2}k_2) \\ k_4 &= \Delta x f(x_n + \Delta x, y_n + k_3) \\ y_{n+1} &= y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned} \tag{5.16}$$

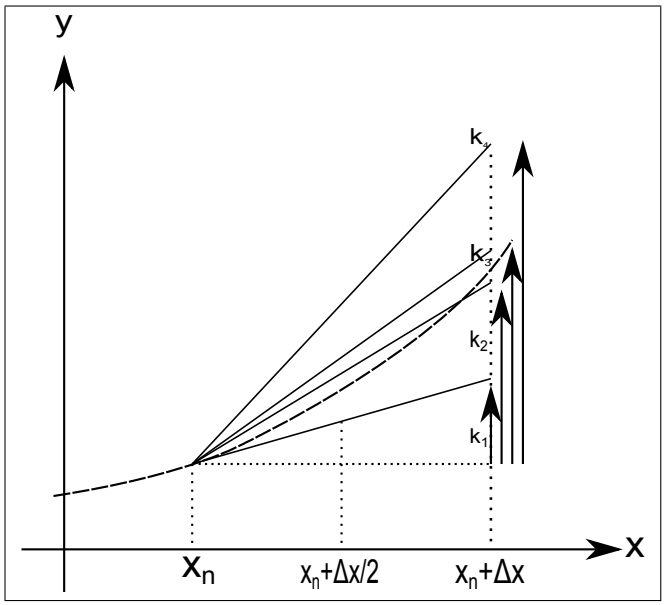

図 11: 4 次の Runge-Kutta イメージ図

x_n から Δx の値を予想するのに k_1, k_2, k_3, k_4 の加重平均を用いる方法である。また、誤差は $O((\Delta x)^5)$ であり、精度が 2 次の Runge-Kutta より良くなっている。

Runge-Kutta 法は 1 階の微分方程式を解くために用いる手法なので、2 階の微分方程式である、式 (5.10)(5.11)(5.12) にはこのままでは用いることが出来ない。そこで 2 階の微分方程式を 1 階の微分方程式へ変換する必要がある。本研究で用いる式 (5.10)(5.11)(5.12) は

$$\frac{dx}{dt} = V_x \tag{5.17}$$

$$m_i \frac{dV_x}{dt} = -\sum_{j=1}^{n} G \frac{m_i m_j (x_j - x_i)}{(\sqrt{(x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2})^3} \tag{5.18}$$

$$\frac{dy}{dt} = V_y \tag{5.19}$$

$$m_i \frac{dV_y}{dt} = -\sum_{j=1}^{n} G \frac{m_i m_j (y_j - y_i)}{(\sqrt{(x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2})^3} \tag{5.20}$$

$$\frac{dz}{dt} = V_z \tag{5.21}$$

$$m_i \frac{dV_z}{dt} = -\sum_{j=1}^{n} G \frac{m_i m_j (z_j - z_i)}{(\sqrt{(x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2})^3} \tag{5.22}$$

のように 1 階の微分方程式へ変換し、Runge-Kutta 法を用いれる状態にして解く。

5.4　N 体問題による性能比較

5.4.1　最適化した状態の性能比較（単精度）

図 12 中では、global_ GPU はグローバルメモリを使用して演算したこと示し、shared_ GPU はシェアードメモリを使用して演算したことを示している。それぞれ最適化されたスレッド数を使用している。グラフでは N=256 体,1024 体,4096 体でそれぞれに対して五つずつプロットされている。これは時刻によるループ回数でプロットしている。ループ回数はそれぞれ左から 1 回,10 回,100 回,1000 回,10000 回となっている。例えば、刻み幅 dt が 0.01、初期時刻 t が 0.0 としてループ 100 回とすると時刻 t が 1.0 になるまで回したことを意味する。また、図 12 は N 体問題を単精度で解いたときのグラフである。

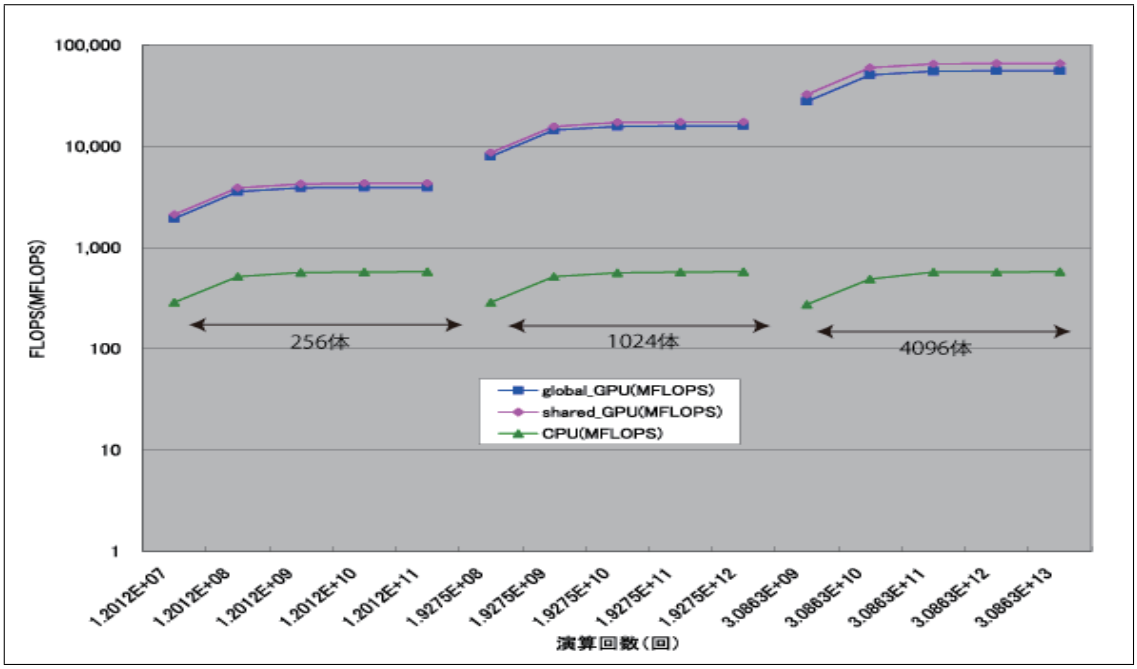

図 12: N 体問題による性能比較図（単精度）

本研究の性能比較の結果、CPU による計算に対して、N=256 体でループが 1 回のときにそれぞれの最小値はグローバルメモリで 6.73 倍、シェアードメモリで 7.37 倍となった。N=4096 体でループが 10 回のときにそれぞれの最大値はグローバルメモリで 103 倍、シェアードメモリで 121 倍となった。

5.4.2　最適化した状態の性能比較（倍精度）

図 13 中でも図 12 と同様にプロットしている。図 13 は N 体問題を倍精度で解いたときのグラフである。

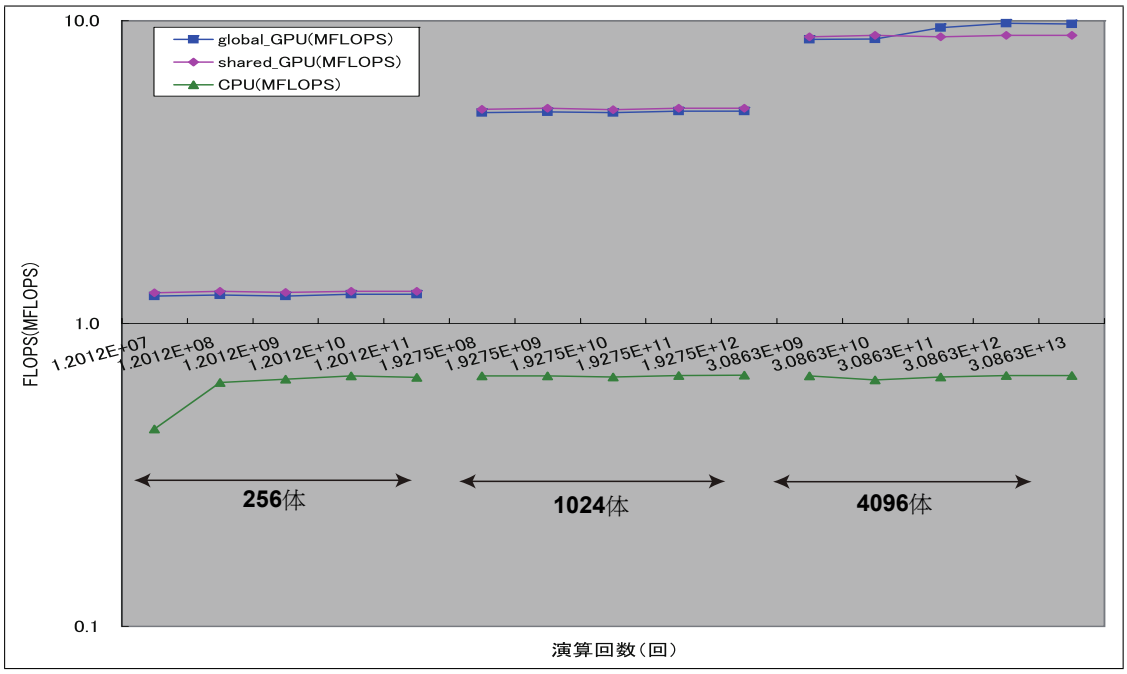

図 13: N 体問題による性能比較図（倍精度）

本研究の性能比較の結果、CPU による計算に対して、N=256 体でループが 1000 回のときにそれぞれの最小値はグローバルメモリで 1.85 倍、シェアードメモリで 1.9 倍となった。それぞれの最大値は N=4096 体でループが 1000 回のときにグローバルメモリで 14.56 倍、N=4096 体でループが 10 回のときにシェアードメモリで 13.7 倍となった。

5.5 精度

NVIDIA の GPU は IEEE754 に準拠している。IEEE754 とは浮動小数点数の計算で最も広く使用されている標準規格である。今回使用した CPU も IEEE754 に準拠している。しかし、浮動小数点計算をしたとき CPU と GPU で計算結果が一致しないという結果が出た。この原因がプログラムのバグなのか GPU の仕様なのかという問題に直面した。

[6]、[7] によると、NVIDIA の GPU は専用の積和演算搭載しており、コンパイラが自動的に浮動小数点数の積和を最適化してくれている。これにより GPU は高い演算能力を獲得している。丸め誤差は積と和は IEEE に準拠しており、最大で 0.5ulp とのことであり、これは通常の CPU と同じである。ulp とは Units in the Last Place の略称であり、ある浮動小数点数の値に対して、その次に大きく表現可能な浮動小数点数や次に小さく表現可能な浮動小数点の差分を表すものである。しかし、NVIDIA の GPU では Add と Multiple という二つの演算を組み合わせて積和を行った場合は積和の中間結果を切り捨てるようになっており、通常の CPU と演算結果が異なる可能性があるということである。このことから本研究で起こった誤差はこれが原因だと考えられる。

そこで、この問題を解決するために CUDA ではある関数が用意されている。単精度の場合は加法に__fadd_rn(x,y)、乗法に__fmul_rn(x,y) が、倍精度の場合は加法に__dadd_rn(x,y)、乗法

に__dmul_rn(x,y) が用意されている。これらの関数を使用することにより、コンパイラは積和への最適化を行わなくなり、CPU との誤差がなくなる。_rn は ”Round Nearest ”といい、IEEE754 で規定されている丸めモードを指定するためのものである。ただし、積和命令を使うことができなくなるので、GPU の演算能力をフルに発揮できなくなる可能性があり、演算時間が遅くなることもある。本研究では、この関数を使用したとき 3〜4 倍ほど遅くなるという結果が得られた。

5.6 性能比較まとめ

N 体問題を用いて性能比較を行った結果、単精度では最大で 121 倍、倍精度で 14.56 倍という結果が得られた。しかし、この結果は演算回数が一番多いところではなかった。このことから科学的数値計算においては演算回数を増やせば演算速度が必ず速くなるというわけではないことがわかった。また、単精度に比べて倍精度ではかなり演算速度が遅くなるということがわかった。

通常、倍精度で計算すると CPU と同様に GPU も遅くなる。ただ、本研究で遅くなった原因は他にもあると考えられる。それは一つのブロック内に生成できるスレッド数が減少したことである。本来一つのブロック内に生成できるスレッド数は最大 3 次元で 512 個である。本研究では、2 次元でスレッドを生成しており、単精度では 256 個を使用して実験を行った。しかし、倍精度では 256 個生成できず、2 次元で最大 169 個であった。これにより単精度に比べて一つのブロック内のスレッド数が減少したことも単精度比べて遅くなったと考えられる。

6　まとめ

6.1　GPU の将来性

現在、3D ゲームだけでなく動画編集や画像編集などにも GPU は利用されてきている。また、本研究のように GPU を利用してプログラムを開発することも増えてきており、科学的数値計算にも利用されたりもしている。このように、次々と GPU の利用が多様化していくことが考えられるので、十分に将来性はあると考えられる。

6.2　まとめ

本研究では整数値処理のモデルとして行列計算と浮動小数点処理のモデルとして N 体問題を用いた。整数値処理のモデルである行列計算では、CPU に対してピーク時で 3000 倍以上の演算速度がでることがわかり、誤差もなかった。また、浮動小数点処理のモデルである N 体問題では整数値処理ほど演算速度がでなかったが CPU の計算よりも速いという結果が得られた。ただし、精度の問題があるので気をつける必要があることがわかった。

また、CUDA で速い演算を可能にするには一つのブロック内に生成するスレッド数を増やしたり、シェアードメモリを使うなどのチューニングが必要だということもわかった。

これらのことより、従来の CPU を用いるシミュレーションに対して、GPU のほうが性能が高く計算時間の短縮に成功し、また、どの程度性能が違うかについても検証できたので当初の目的を達成できた。

参考文献

[1] Mike Thomas,Steve McBarnes：「GPUReview」：
http://www.gpureview.com/GeForce-GTX-285-card-605.html(2010/12/19 アクセス)

[2] Intel：「Intel Support Home」：
http://www.intel.com/support/processors/sb/cs-023143.htm#3(2010/12/19 アクセス)

[3] NVIDIA：「CUDA プログラミングガイド Ver1.1」：
http://www.nvidia.co.jp/docs/IO/51174/NVIDIA_CUDA_Programming_Guide_1.1_JPN.pdf(2010/12/19 アクセス)

[4] CQ 出版株式会社：「Interface」：
http://interface.cqpub.co.jp/(2010/12/17 アクセス)

[5] 下馬場朋禄，伊藤智義：「CUDA 技術を利用した GPU コンピューティングの実際（後編）」：
http://www.kumikomi.net/archives/2008/10/22gpu2.php?page=1(2010/12/17 アクセス)

[6] フィックスターズ：「NVIDIA CUDA Information Site」：
http://gpu.fixstars.com/index.php/GPU%E3%81%AE%E8%A8%88%E7%AE%97%E7%B2%BE%E5%BA%A6%E3%81%AB%E3%81%A4%E3%81%84%E3%81%A6(2010/1/21 アクセス)

[7] NVIDIA：「CUDA プログラミングガイド Ver2.3-Appendix C」：
http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf(2010/1/21 アクセス)

[8] 真貝寿明：「徹底攻略常微分方程式」

[9] 青木尊之，額田彰：平成 21 年 11 月 20 日初版：「はじめての CUDA(クーダ) プログラミング」：工学社