

CREST 「生命動態の理解と制御のための基盤技術の創出」研究領域
第 8 回数理デザイン道場

GPGPUによる高速化

武石 直樹
大阪大学大学院基礎工学研究科

2017 年 6 月 30 日
プラザ ヴェルディ (静岡県沼津市大手町 1-1-4)

はじめに

本資料は、GPU コンピューティングの基礎的なプログラミング技術を習得することを目的に、京都大学ウイルス・再生医科学研究所の安達研究室に所属する学生向けに作成されました。本資料の大部分は、既に初版化されている『はじめての CUDA プログラミング（青木尊之・額田彰著）』[8] を参考に作成され、GPU コンピューティングの基礎的かつ実践的な部分に絞ってまとめられています。より高度な GPU 計算の習得を目指す方は、この書籍を含め、是非 CUDA プログラミング専門の書籍を参照していただきたいです [7, 1]。なお、書籍出版当時（2009 年）は、CUDA2.3 が最新版でしたが、現在（2017 年 6 月）では CUDA8.0 が最新版としてリリースされており、インストールパッケージが 1 つだけで済むなど、環境設定が一段とやりやすくなりました。さらに、CUDA5.0 以降、プログラムの書き方に大きな変更があったため、本資料ではその点も踏まえ、CUDA のバージョンの違いによる記述の変更箇所を明記するようにしました（メモリの確保 <p.7>、データの転送 <p.7>、計算時間の計測 <p.25>）。プログラミングの理解は、実装・計算で深まります。本資料が、GPU 計算をこれから始める方々の手ほどきとなり、GPU 計算技術の向上に役立てられたとしたら望外の喜びです。

最後に、本資料を作成するにあたり、京都大学ウイルス・再生医科学研究所の木村健治氏には、原稿を非常に丁寧に読んでもらい、曖昧な箇所や多くのミスをご指摘いただきました。この場をお借りして心より御礼申し上げます。

目次

はじめに	i
第 1 章 GPGPU	1
1.1 なぜ GPU を使うのか？	1
1.2 GPU architecture	3
第 2 章 CUDA プログラミング	6
2.1 CUDA プログラミングの概要	6
2.2 メモリの確保	7
2.3 データの転送	7
2.4 グリッドとブロックとスレッド	8
2.5 CUDA の階層的メモリ・モデル	9
2.6 カーネル (kernel) 関数	10
2.6.1 dim3 宣言	11
2.6.2 カーネル関数の実行指示	12
2.6.3 ビルトイン変数	13
2.6.4 メモリへのアクセス	15
第 3 章 GPU 計算	17
3.1 常微分方程式 (ルンゲ・クッタ法)	17
3.1.1 メモリ確保	18
3.1.2 粒子の初期条件	18
3.1.3 粒子位置の時間積分	19
3.1.4 プログラム実行	23

3.1.5	ファイルの出力 (BMP ファイルについて)	23
3.1.6	演算性能	24
3.1.7	実行速度の比較	25
3.2	偏微分方程式 (拡散方程式)	27
3.2.1	初期条件	28
3.2.2	計算手法	28
3.2.3	境界条件	28
3.2.4	GPU 計算のプログラム	29
3.2.5	演算性能	29
3.2.6	高速化の基本概念	30
3.3	シェアード・メモリの利用	32
3.3.1	シェアード・メモリの宣言とアクセス	33
3.3.2	ブロックの境界格子の計算	35
3.3.3	シェアード・メモリの節約	37
3.3.4	変数 (レジスタ) の利用	38
3.3.5	演算性能の比較	40
	付録	42
A.	CUDA インストール方法	42
A.1.	CUDA8.0 のインストール	42
A.2.	Linux(CentOS) での CUDA8.0 のインストール	43
B.	GPU プログラミングの応用例	49
B.1.	レーリー・テラー不安定性の成長	49
B.2.	AL-Si 合金の樹枝状凝固成長	49
	参考文献	51

第1章

GPGPU

GPU は、 Graphic Processing Unit の略で、言葉の通り画像処理に特化したチップとして開発された経緯があります。この GPU を本来の用途である画像処理以外の目的で、一般的な計算に使用することを **GPGPU (General-Purpose computing on GPUs)** と言います。

1.1 なぜ GPU を使うのか？

■ 高性能

GPU の特徴は、広いバンド帯域と高い演算性能にあります。この特徴を最大限引き出せれば、(計算手法によっては)CPU 計算に対しておよそ 500 倍という驚異的な高速化倍率をひきだすことができます。これにより、従来不可能とされてきた大規模数値解析が可能になります。

GPU 計算の一例として、カプセル懸濁液の大規模粘性解析を挙げてみます [3]。カプセル個体数 256(総メッシュ数およそ 130 万) を GPU16 台で並列計算し、ピーク性能として 10TFlops を実現しています (Fig.1.1)。他にも、流体格子数 430 万、固体メッシュ数 80 万規模の流体構造連成解析を単一 GPU で行った例もあります [4]。

■ 設置の簡便性

スーパーコンピューター (スパコン) が初めて 1TFlops を超えたのが 1997 年 (Intel ASCI; 1.4TFlops) ですが、GPU を通常のデスクトップに挿入して使用するだけで、当時のスパコンの性能をパソコン 1 台で実現することが出来ます。ハイエンドでもコンシューマタイプの GPU は数万円程度ですので、一般的なスパコンの設置費用に比べるとかなり低価格です。GPU の消費電力は高いと言われても (GeForce GTX TITAN X の推奨供給電力は 850W)，消費電力に対する浮動小数点演算 (Flops/W) の観点から見ればむしろ省エネといえます。

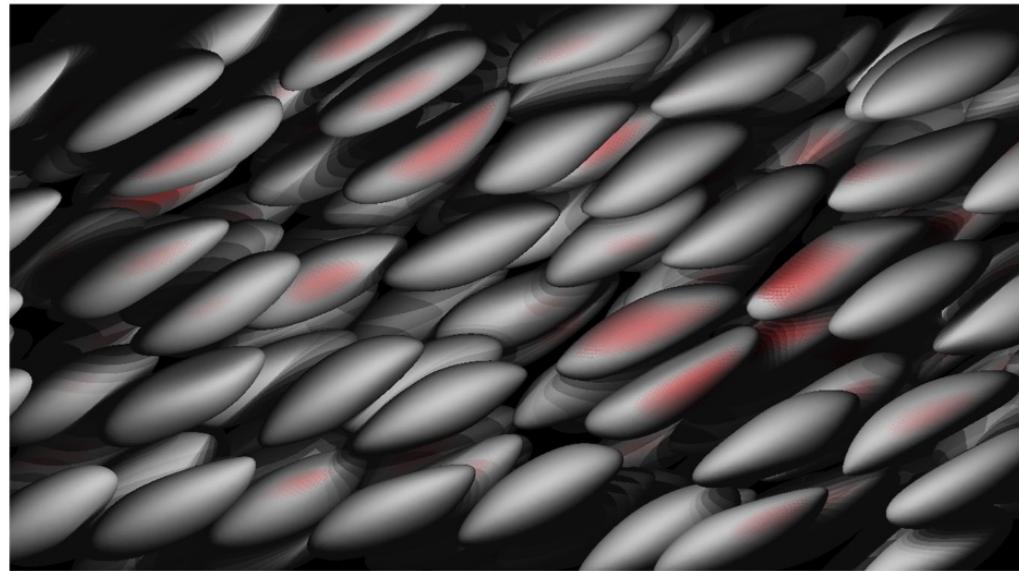


図 1.1 カプセル懸濁液の大規模粘性解析 (カプセル固体数 256, カプセルのメッシュ総数 130 万) [3].

GPU の運用には CUDA(その他には, AMD stream, OpenCL など) を使います。CUDA は Compute Unified Device Architecture の略で, NVIDIA 社 (GPU ベンダーの一つ) が無料で提供している GPGPU 向けの統合開発環境 (GPU を動作させるためのプログラミング・モデルおよびプログラミング言語, コンパイラ, ライブライアリなど) です。CUDA のインストール方法は付録 A(p.42) を参照して下さい。



メモ (マシンの性能評価)

マシンのパワーを見る指標として, GEMM(汎用行列-行列積) と GEMV(汎用行列-ベクトル積) があります。単精度 (single precision) ならば, SGEMM と SGEMV, 倍精度 (double precision) ならば, DGEMM, DGEMV になります。GEMM は “ $C \leftarrow \alpha AB + \beta C$ ” の計算を行い, このときのデータ量は $O(N^2)$, 演算量は $O(N^3)$, Bytes/flop は $O(1/n)$ なので大きな次元でほぼ 0 となります。一方, GEMV は “ $y \leftarrow \alpha Ax + \beta y$ ” の計算を行い, データ量は $O(N^2)$, 演算量は $O(N^2)$, Bytes/flop は $O(1)$ となります。したがって, 演算性能の指標としては GEMV を, メモリバンド域の指標には GEMV を使います。より詳しい説明は, 本資料の本旨からずれますので興味ある方は各自調べてみて下さい。

1.2 GPU architecture

■ GPU の構造

NVIDIA 社の GPU architecture を Fig.1.2 に示します。ビデオ・メモリと GPU とはメモリ・インターフェイスで接続されています。CPU とメイン・メモリ間はせいぜい 64bit(8Byte) 幅で接続されていますが、GPU とビデオ・メモリ間は、ハイエンドだと 512bit(64Byte) と、数倍大きなメモリ バンド幅で接続されています。

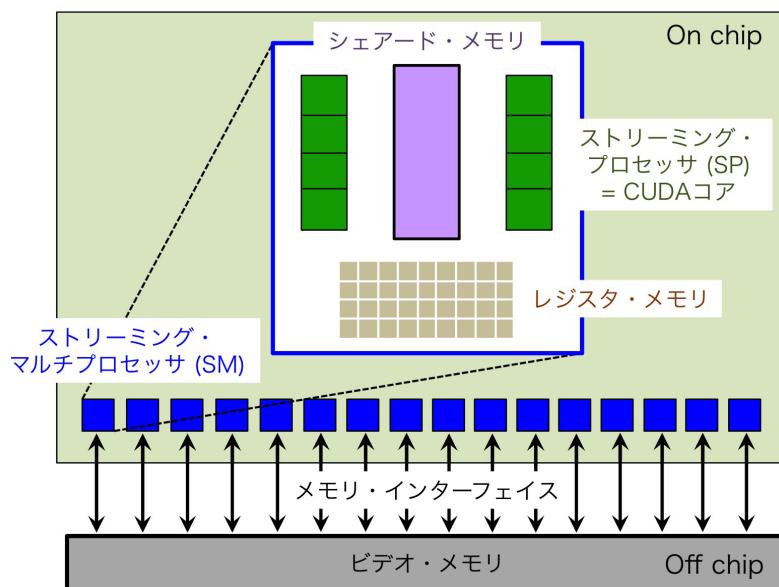


図 1.2 NVIDIA 製 GPU architecture.

GPU チップ内部には、ストリーミング・マルチプロセッサ (SM) が多数入っています (Tesla M2090 では 16 個搭載)。さらに SM の内部には、ストリーミング・プロセッサ (SP)(別名 : CUDA コア) と呼ばれる演算装置と、シェアード・メモリやレジスタ・メモリといったメモリなどで構成されています。SP は計算を行う最小単位の演算処理ユニットと言えます。

SM 中の SP の数は GPU の世代によって異なりますが、上で挙げた Tesla M2090 では 32 個の SP が入っています。したがって、Tesla M2090 には $32 \times 16 = 512$ (個) の SP が搭載されていることになります。Intel の Core i7 でも CPU コアが 4 個であることを考えると、これは非常に大きな数です。



メモ (NVIDIA 製品の GPU シリーズ)

□ GeForce シリーズ

いわゆるコンシューマ向けで、DirectX に最適化されているため 3D ゲームなどに適している。GPU のチップ開発・製造は NVIDIA 社が行い、それを載せるグラフィックボードは各 OEM によって生産されている。そのため、同じチップを載せていてもボード自体の仕様が異なることがある。

□ Quadro シリーズ

OpenGL に最適化されており、3DCG 作成や CAD などを利用するのに適している。性能は GeForce と変わらないが、ゲームは苦手。NVIDIA が認定した企業のみグラフィックボードの製造が許されており、また製造会社が限定されているため、NVIDIA のサポートが注力されている。

□ Tesla シリーズ

グラフィックボードから映像出力機能を除いたもので、まさに GPU コンピューティングのために開発された GPU。倍精度演算に強い。NVIDIA 社が全ての Tesla の動作確認を行っているため信頼性が高い。また、GeForce に比べてより多くのメモリが積まれている。Tesla にはファンがあるものと無いものがある。基本的にはファンのあるものはデスクサイドマシンで、ファンのないものはラックマウントケースで使用する。

■ GPU の性能

Intel の Core i7 はメモリに DDR3 を採用し、メモリ・コントローラをチップに内蔵したため、メモリ・バンド帯域を 36GB/sec まで太くすることが出来ました。一方、GeForce GTX 285(2009 年発売)では、メモリ・クロック数が 2.48GHz、メモリ・インターフェイスが 512bit(64Byte)なので、 $2.48\text{GHz} \times 64\text{Byte} \approx 159\text{GB/sec}$ のメモリ・バンド幅となります。最新の GeForce シリーズである GeForce GTX TITAN X(2016 年発売)では、3584 基の CUDA コア、メモリ・バンド帯域 480GB/sec、GDDR5X メモリを 12GB 搭載し、単精度浮動小数点演算性能は 11TFlops という性能です。

2009年4月から稼働している地球シミュレータ2に採用されているSX-9の1プロセッサあたりの演算性能が102GFlops、メモリバンド幅256GB/secなので、まさにスパコン並みの性能をGPUは持っていることがわかります。

第 2 章

CUDA プログラミング

2.1 CUDA プログラミングの概要

GPU はパソコンに装着して利用することを前提としており、GPU 単体では動作しません。したがって、CUDA のプログラミングも GPU を動かすコードだけでは成り立たません。CUDA プログラムの構成は、① CPU を動作させるホスト・コードと② GPU を動作させるデバイス・コードから成ります。ホスト・コードで行うことは次の 4 つです。

1) デバイス・メモリの確保

Kernel 関数内で使用するメモリ (= GPU のデバイス・メモリ) 上に配列を確保します (詳細 p.7).

2) CPU→GPU へのデータ転送

デバイス・メモリとホスト・メモリ (CPU 側のメモリ) 間のデータを通信します (詳細 p.7).

3) カーネル (kernel) 関数の呼び出し

GPU で計算させるためには、kernel 関数を呼び出す必要があります (詳細 p.10).

4) GPU→CPU のデータ転送

Kernel 関数で計算した結果を CPU 側に渡し、結果を出力させます (詳細 p.7).

一方、デバイス・コードで行うことは、単に “kernel 関数での計算” ということになります。実際は、ホスト・コードもデバイス・コードも統一のテキスト・ファイル内に記述します。このテキスト・ファイル (CUDA プログラム) は、「program.cu」など「.cu」という拡張子を付ける約束になっています。この拡張子がついたテキスト・ファイルをコンパイルするのが nvcc というコンパイラーです。nvcc は CUDA のインストール時に使えるようになります。CUDA プログラミングとは、C(あ

るいは C++, Fortran) からの単純な拡張であるとことを強調しておきます。

2.2 メモリの確保

上述の通り、GPU 計算は kernel 関数内の計算と言っても過言ではありません。この kernel 関数で使用するための準備（メモリの確保）と手続き（dim3 宣言）をこれから説明していきます。デバイス・メモリ（例；`xD` という配列）の確保はプログラム上で次のように記述します（プログラム先頭で “#include<stdlib.h>” を記述しておきましょう）。

```
CUDA_SAFE_CALL( cudaMalloc( (void**) &x, sizeof(float)*n));
```

これで、デバイス・コード内で `float` 型の `x[n]` という配列を使用することができます。“CUDA_SAFE_CALL” はあってもなくても良いです。ちなみに、これは CUDA4.9 までの書き方で、CUDA5.0 以降は次のように記述します。

```
checkCudaErrors( cudaMalloc( (void**) &x, sizeof(float)*n));
```

CUDA のバージョンによって書き方が変わるために、その度にプログラムを修正する必要があります。CUDA5.0 から 8.0(2017 年 6 月現在) までについては記述に関して大きな変更はありません。ホスト・メモリ（例；`xH` という配列）の確保は C 言語での通常の書き方です。

```
*xH = (float *) malloc (sizeof (float)*n);
```

先頭の “*” はポインタ指定を表します。C 言語の詳しい記述の仕方は本資料では割愛しますが、サンプル・プログラムを見れば雰囲気がそれとなくわかると思います。ちなみに、インデント（スペース）には意味はなく、単に見やすさのために入れています。

2.3 データの転送

プレプロセス（計算準備）の段階で確保したホスト・メモリにデータを格納したならば、デバイス・メモリにデータを転送（CPU→GPU）します。

```
(CPU→GPU)
CUDA_SAFE_CALL (cudaMemcpy(xD, xH, sizeof(float)*n, cudaMemcpyHostToDevice));
```

```
(GPU→CPU)
CUDA_SAFE_CALL (cudaMemcpy(xH, xD, sizeof(float)*n, cudaMemcpyDeviceToHost));
```

これも、CUDA4.9 以前までの書き方で、CUDA5.0 以降は次のように書き換えられます。

```
(CPU→GPU)
checkCudaErrors (cudaMemcpy(xD, xH, sizeof(float)*n, cudaMemcpyHostToDevice));
```

```
(GPU→CPU)
checkCudaErrors (cudaMemcpy(xH, xD, sizeof(float)*n, cudaMemcpyDeviceToHost));
```

「データの確保」と「データの転送」の仕組みについてより詳しく知りたい方は、「CUDA ランタイム API」というキーワードで調べてみて下さい。

これで、デバイス上にデータの配列を送れたので、いよいよ kernel 関数を呼び出して GPU で計算させるだけです。しかしその前に、GPU でのメモリ(デバイス・メモリ)の構成について言及したいと思います。なぜなら、kernel 関数を呼び出す宣言の仕方は GPU のメモリ構成と大きく関わっているからです。次から、「グリッド」、「ブロック」、「スレッド」という GPU 計算において重要な概念を説明します。

2.4 グリッドとブロックとスレッド

デバイス・メモリは、グリッド(grid), ブロック(block), そしてスレッド(thread)という3つの階層に分けて管理されています。GPU 計算をする際は、このメモリの階層を意識しておくことが重要です。デバイス・メモリの階層性の概念図を Fig.2.1 に示します。

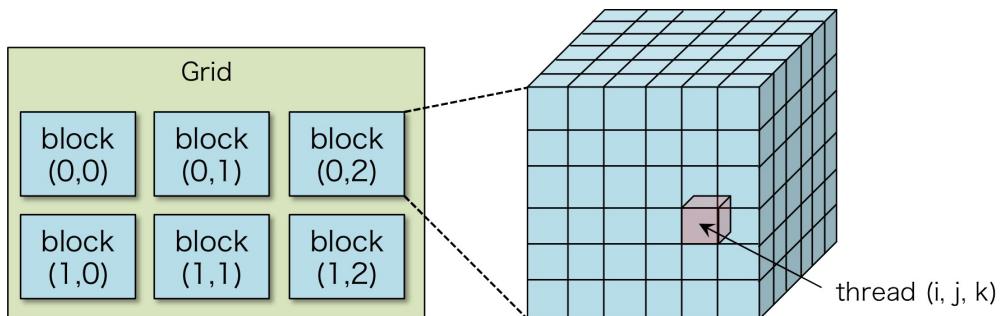


図 2.1 デバイス・メモリの階層性。

メモリの最も大きな階層がグリッドです。グリッドの中にはブロックという階層が構成されています。このブロックは x, y, z 方向の 3 次元的に配置され管理されますが、現時点の使用では z 方向のブロック数は 1 でなければならないので、実質的に 2 次元配置となります。ブロック数の x, y 方向の最大値は GPU の世代で異なります。これを超えて配置すると実行時にエラーになります。

ブロックの中にはスレッドという階層が構成され、3 次元的に管理されています。全てのスレッドにおいて次に説明するカーネル関数が実行されます。グリッド内のブロック数やそのブロック中のスレッド数は、ホスト・プログラムでカーネル関数を呼び出す際に指定します。1 ブロック当たりの最大スレッド数も GPU の世代で異なりますが、GForce GTX TITAN X では 1024 です。それ以前のものでは 512 が多いです。

2.5 CUDA の階層的メモリ・モデル

CUDA のプログラミングを行う際は、ハードのことを頭の片隅に置くようにしましょう。メモリをどのように使うかが GPU の性能を引き出すカギになります。CUDA の階層的なメモリ・モデルを Fig.2.2 に示します。メモリの 3 つの大きな分類を以下に示します。

□ スレッド固有メモリ：

レジスタ、ローカル・メモリ

□ ブロック内共有メモリ：

シェアード・メモリ

□ グリッド内(全スレッド)共有メモリ：

グローバル・メモリ、コンスタント・メモリ、テクスチャ・メモリ

上記のメモリの特徴を、簡単に以下にまとめます。

■ レジスタ

カーネル関数内で使われる変数の値が保持されます。1 クロックで値の読み書きができるほど、高速アクセスが可能です。

■ ローカル・メモリ

GPU のチップの外に置かれ、レジスタに比べると 100 倍以上もアクセスが低速です。読み出す場合も、一度レジスタ条に読み込んでから使われます。

■ シェアード・メモリ

レジスタと同等に高速アクセスが可能なメモリで、各 SM の中に 16,384byte 備わっています。ブロック内のスレッドは同一 SM で実行され、ブロック内の全てのスレッドからシェアード・メモリを共有することができます。

■ グローバル・メモリ

GPU のチップの外に置かれ、レジスタやシェアード・メモリに比べると 100 倍以上もアクセスが低速です。ハードとしてはビデオ・メモリ上に置かれているので、グローバル・メモリとして使える量は、ビデオカードのビデオ・メモリの搭載量と言えます。

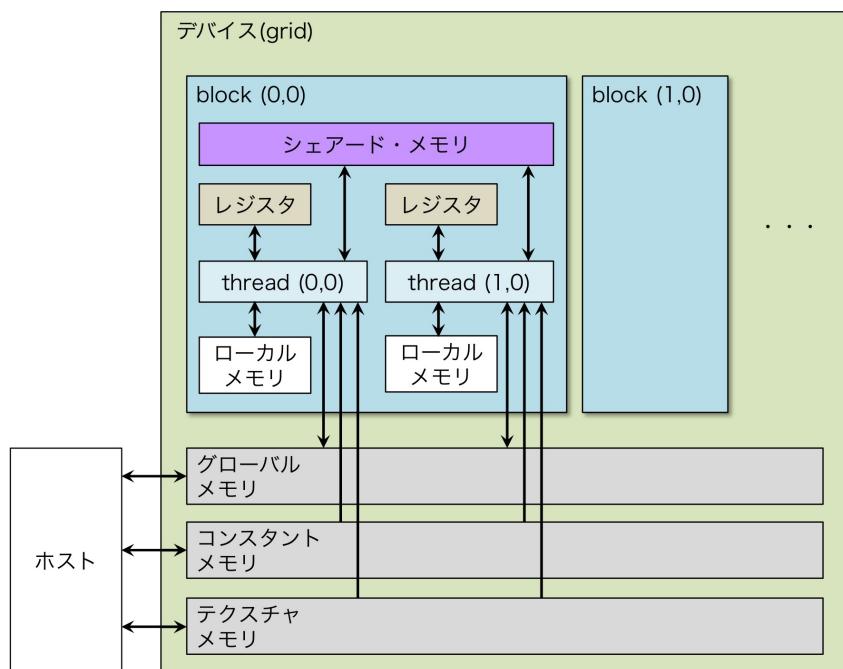


図 2.2 CUDA のメモリ・モデル。

2.6 カーネル (kernel) 関数

GPU に計算してもらいたい部分がカーネル関数です。カーネル関数内をどのように記述するかも重要ですが、カーネル関数の実行指示の仕方も重要です。CUDA では、どのようなスレッド数でカーネル関数を実行させるかを実行時に指定しなければなりません。

2.6.1 dim3 宣言

先に述べたブロックとスレッドは 3 次元で管理されています。CUDA ではその数を指定するためには、3 次元ベクトルの整数型の宣言である dim3 が使われます。これは、CUDA における C 言語の拡張部分になりますが、nvcc でコンパイルする限り他には特に何もすることなく使えます。例えば、

```
dim3    a(10, 10, 1), b(16, 8, 2);
```

などと宣言します。実質的には a と b は構造体として宣言され、メンバーとして “a.x, a.y, a.z” と “b.x, b.y, b.z” が以下のように自動的に決まります。

```
a.x = 10; a.y = 10; a.z = 1;  
b.x = 16; b.y = 8; b.z = 2;
```

また、

```
dim3    a, b;
```

とだけ宣言した場合は、

```
a.x = 1; a.y = 1; a.z = 1;  
b.x = 1; b.y = 1; b.z = 1;
```

と同じです。宣言した後に、

```
dim3    a, b;  
  
a.x = 10; a.y = 10; a.z = 1;  
b.x = 16; b.y = 8; b.z = 2;
```

のようにメンバーの値を書き換えることができます。

```
dim3    a(2, 3), b(4);
```

とした場合は、

```
a.x = 2; a.y = 3; a.z = 1;  
b.x = 4; b.y = 1; b.z = 1;
```

と同じです。この dim3 で指定した変数を用いてカーネル関数の実行パラメータを指定します。

2.6.2 カーネル関数の実行指示

CPU で実行させる通常の C 言語の場合、プログラム中に

```
function_cpu (aH, bH, cH);
```

と書くだけです。“aH, bH, cH” は関数の引数です。一方、CUDA のカーネル関数では、

```
function_gpu <<< Dg, Db, Ns, S >>> (aD, bD, cD);
```

となります。“<<< >>>” の部分は CUDA プログラミングの拡張部分です。“aD, bD, cD” はカーネル関数の引数です。“<<< >>>” 内のそれぞれの引数の意味は以下の通りです。

第一引数 “Dg”

グリッドのサイズ (= グリッド中のブロックの個数) を 3 次元的に指定します。ただし、Dg.z = 1 に限定されています。dim3 型で宣言します。省略はできません。

第二引数 “Db”

ブロックのサイズ (= ブロックの中のスレッドの個数) を 3 次元的に指定します。スレッドの総数は 512 に制限されています (GeForce GTX TITAN X では 1024 です)。dim3 型の変数で指定します。省略はできません。

第三引数 “Ns”

シェアード・メモリのサイズを指定します。整数または整数型で指定します。省略可能で、省略した際は 0 が入ります。

第四引数 “S” (本資料では詳しく解説はしません)

実行のストリーム番号を 0 以上の整数で指定します。省略可能で、省略した場合は、ストリーム番号 0 でそのカーネル関数が実行されます。ストリーム番号 0 は特別扱いされており、カーネル関数をストリーム番号 0 で実行した場合は、ストリーム番号 0 の処理が終了するまで次のストリーム番号 0 を実行することはできません。つまり、ストリーム番号 0 は同時に 1 つしか流れません。



メモ（ストリーム）

GPU もコンパイラも、タスク間の依存関係を自動的に判定する機能は持っていない。プログラマーがタスク間の依存関係を明示的に指定する必要があります。そこで用いられるのがストリーム (stream) です。ストリームの詳細な使い方については、参考文献 [8] の p.106 を参照して下さい。

例えば、カーネル関数の実行指示文として以下の記述があったとします。

```
dim3 Dg(5, 5, 1), Db(4, 4, 2);
func_gpu <<< Dg, Db >>> (aD, bD, cD);
```

もし、“Dg” や “Db” が 1 次元要素しか持っていない場合、つまり、

```
dim3 Dg(5, 1, 1), Db(4, 1, 1);
```

のときは、

```
func_gpu <<< 5, 4 >>> (aD, bD, cD);
```

だけですみます。ここまでで、カーネル関数を呼び出すところまできたので、最後にカーネル関数内の計算について見ていきます。

2.6.3 ビルトイン変数

カーネル関数内では宣言しなくても使うことのできる変数があります。これをビルトイン変数といいます。ただし、ビルトイン変数は値を読み出すだけで、書き込むことは出来ません。GPU では数十万という数のスレッドが実行されるため、各スレッドが自分自身のスレッドの ID を把握しておくことが重要となり、この ID の指定にビルトイン変数を用います。例えば、

```
dim3 a(10, 20, 1), b(16, 8, 4);
```

と宣言して、次のようなカーネル関数をホストから呼び出して実行したとします。

```
program_gpu <<< a, b >>> (xD, yD);
```

カーネル関数の中では、グリッドのサイズを指定するビルトイン変数 `gridDim` が使用でき、`x`, `y`, `z` 方向のそれぞれのグリッドサイズが次のように記述されます。

`gridDim.x = 10 (= a.x),`

`gridDim.y = 20 (= a.y),`

`gridDim.z = 1 (1 に限定).`

一方、ブロックサイズはビルトイン変数 `blockDim` を用いて、

`blockDim.x = 16 (= b.x),`

`blockDim.y = 8 (= b.y),`

`blockDim.z = 4 (= b.z),`

と記述されます。実行されているスレッドが所属する block の ID はビルトイン変数 `blockIdx` を用いて、

`blockIdx.x = 0 ~ 9 のどれかの値,`

`blockIdx.y = 0 ~ 19 のどれかの値,`

`blockIdx.z = 0,`

と記述されます。また、実行されているスレッドの ID はビルトイン変数 `threadIdx` を用いて、

`threadIdx.x = 0 ~ 15 のどれかの値,`

`threadIdx.y = 0 ~ 7 のどれかの値,`

`threadIdx.z = 0 ~ 3 のどれかの値,`

と記述されます。

2.6.4 メモリへのアクセス

カーネル関数を用いた簡単な計算例として、N次元のベクトル演算を行うことを考えます。3つの配列“A, B, C”を用いて、C言語のプログラムならば以下のように記述します。

```
for (i = 0; i < N; i++) C[ i ] = A[ i ] + B[ i ];
```

カーネル関数を用いて、一つのスレッドが一つの要素(i番目)の足し込みを担当させることにします。すると、全体でN個のスレッドが必要になります。とりあえず、1ブロックには最大512個のスレッドを割り当てることにしましょう。カーネル関数の呼び出しは以下のように記述されます。

```
dim3 Dg(N/512), Db(512);
vec_add <<< Dg, Db >>> (A_D, B_D, C_D);
```

呼び出されたカーネル関数“vec_add”以下のように記述されます。

```
--global__ void vec_add
//=====
(
    float *A, // array pointer of the global memory
    float *B, // array pointer of the global memory
    float *C // array pointer of the global memory
)
//=====
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    C[ i ] = A[ i ] + B[ i ];
}
```

ここで、“`--global__`”はデバイス上でのみ実行され、ホスト側からのみ呼び出されるカーネル関数を明示する関数修飾子です。int型で宣言した変数*i*によって、*i*番目のスレッドIDにアクセスします(Fig.2.3)。

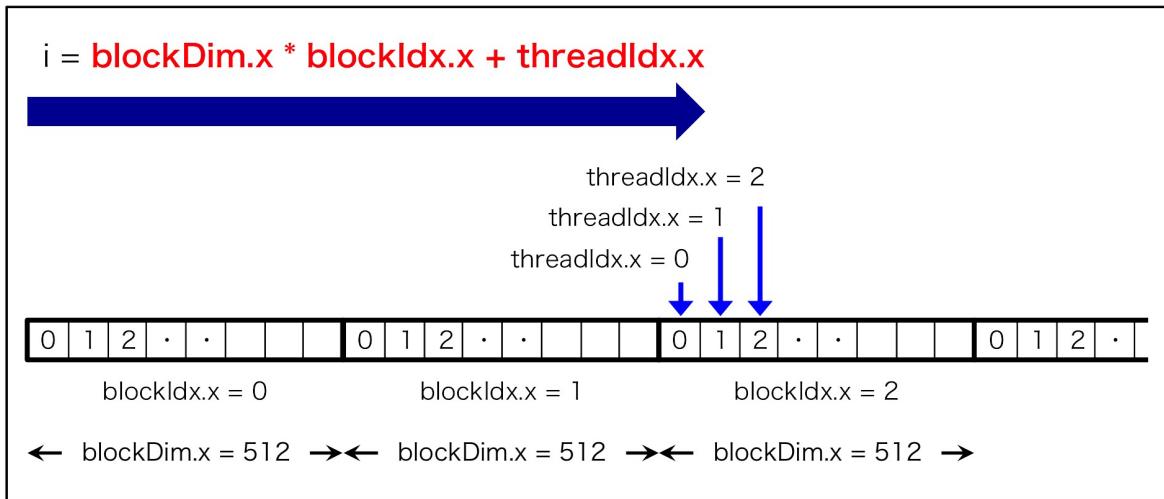
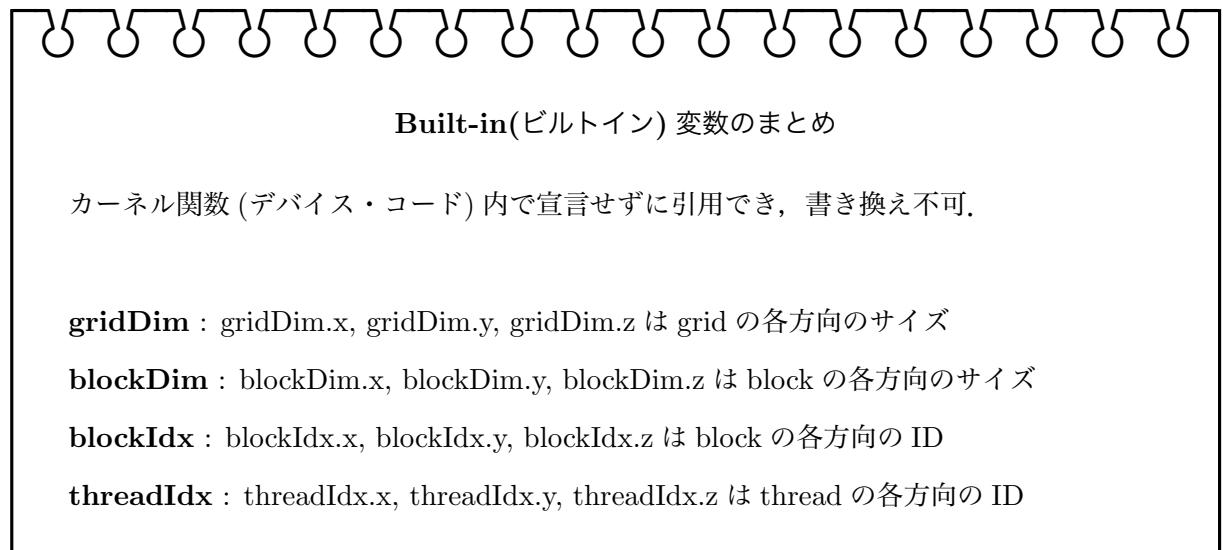


図 2.3 ビルトイン変数を使った 1 次元配列へのアクセス (概念図)。



第 3 章

GPU 計算

3.1 常微分方程式 (ルンゲ・クッタ法)

この章から本格的に GPU 実装とその計算を行っていきます。例題として、1万個以上の粒子の時間発展方程式を計算してみます。1万個以上の粒子は初期にある半径の円の中に分布しているとします。粒子は位置と時刻で決まる速度分布をもち、2次元平面内で運動をします。粒子番号を i とすると、 i 番目の粒子の位置 (x_i, y_i) の時間変化は次で表されます。

$$\frac{dx_i}{dt} = u(x_i, y_i, t), \quad \frac{dy_i}{dt} = v(x_i, y_i, t) \quad (3.1)$$

数値計算で時間に関する常微分方程式を解くには、本来は連続である時間をステップ状 (Δt という時間ステップ) に進むと近似します。本来は連続的な時間進行を Δt という有限な間隔で進むと近似することを離散化といいます。常微分方程式を数値計算で解く方法は色々ありますが（オイラー法、リープ・フロッグ法など）、ここでは、計算工学の分野で最も定番となっているルンゲ・クッタ法を使うことにします。ルンゲ・クッタ法は、「1段階解法」、「2段階解法」…と段数を選ぶことができ、段数に応じて計算精度が向上します。最も精度の低い1段のルンゲ・クッタ法（前進オイラー法）は次のような離散式になります。

$$x_i^{n+1} = x_i^n + u(x_i^n, y_i^n, t^n)\Delta t, \quad y_i^{n+1} = y_i^n + v(x_i^n, y_i^n, t^n)\Delta t \quad (3.2)$$

四段のルンゲ・クッタ法は以下のように1段ずつ計算を進めます。 p と q の下付き添字がルンゲ・クッタ法の段数を表します。

$$\begin{aligned}
 p_1 &= u(x_i^n, y_i^n, t^n), & q_1 &= v(x_i^n, y_i^n, t^n), \\
 p_2 &= u\left(x_i^n + \frac{1}{2}p_1\Delta t, y_i^n + \frac{1}{2}q_1\Delta t, t^n + \frac{1}{2}\Delta t\right), & q_2 &= v\left(x_i^n + \frac{1}{2}p_1\Delta t, y_i^n + \frac{1}{2}q_1\Delta t, t^n + \frac{1}{2}\Delta t\right), \\
 p_3 &= u\left(x_i^n + \frac{1}{2}p_2\Delta t, y_i^n + \frac{1}{2}q_2\Delta t, t^n + \frac{1}{2}\Delta t\right), & q_3 &= v\left(x_i^n + \frac{1}{2}p_2\Delta t, y_i^n + \frac{1}{2}q_2\Delta t, t^n + \frac{1}{2}\Delta t\right), \\
 p_4 &= u\left(x_i^n + \frac{1}{2}p_3\Delta t, y_i^n + \frac{1}{2}q_3\Delta t, t^n + \Delta t\right), & q_4 &= v\left(x_i^n + \frac{1}{2}p_3\Delta t, y_i^n + \frac{1}{2}q_3\Delta t, t^n + \Delta t\right), \\
 x_i^{n+1} &= x_i^n + \frac{p_1 + 2p_2 + 2p_3 + p_4}{6}\Delta t, & y_i^{n+1} &= y_i^n + \frac{q_1 + 2q_2 + 2q_3 + q_4}{6}\Delta t.
 \end{aligned} \tag{3.3}$$

3.1.1 メモリ確保

計算で必要となる配列は、現時刻 (t) での粒子の位置 (x, y) と、次の時刻 ($t + \Delta t$) での粒子の位置 (xn, yn) ですので、それぞれホスト側とデバイス側で、粒子数 (=N=2562) 分だけ配列を確保します。

(ホスト側)

```

com->xH = (VTYPE *) malloc(sizeof(VTYPE)*N); if (com->xH == NULL) error(0);
com->yH = (VTYPE *) malloc(sizeof(VTYPE)*N); if (com->yH == NULL) error(0);
com->xnH = (VTYPE *) malloc(sizeof(VTYPE)*N); if (com->xnH == NULL) error(0);
com->ynH = (VTYPE *) malloc(sizeof(VTYPE)*N); if (com->ynH == NULL) error(0);

```

(デバイス側)

```

checkCudaErrors (cudaMalloc((void**)&(com->xD ), sizeof(VTYPE)*N));
checkCudaErrors (cudaMalloc((void**)&(com->yD ), sizeof(VTYPE)*N));
checkCudaErrors (cudaMalloc((void**)&(com->xnD ), sizeof(VTYPE)*N));
checkCudaErrors (cudaMalloc((void**)&(com->ynD ), sizeof(VTYPE)*N));

```

ここで、“com->” は “computation” という構造体を表しています。サンプル・プログラミングでは、拡張性を考慮して構造体を使用しています。粒子数 N はヘッダー・ファイル (def.h) 内で定義しています。また、“VTYPE” は def.h 内で “float” として定義されています。

3.1.2 粒子の初期条件

(0.5(= GX), 0.25(= GY))を中心とした半径 0.24(= R) の円の内部に N 個の粒子をランダムに配置します。C 言語の現在時刻を使った乱数生成関数 (rand 関数) を用いて、次のように記述します

(プログラム先頭で, “#include<stdlib.h>” と “#include <time.h>” を記述しておきましょう).

```

strand ((unsigned int)time(NULL));
for (i = 0; i < N; i++) {
    do {
        xs = (float)rand() / RAND_MAX;
        ys = (float)rand() / RAND_MAX;
    } while ((xs - GX)*(xs - GX) + (ys - GY)*(ys - GY) > R*R);
    com->xH[i] = xs; com->yH[i] = ys;
    com->xnH[i] = xs; com->ynH[i] = ys;
}

```

ホスト側の配列に格納した値 “xH, yH” をデバイス側 “xD, yD” に転送します.

```

checkCudaErrors (cudaMemcpy(com->xD, com->xH, sizeof(VTYPE)*N, cudaMemcpyHostToDevice));
checkCudaErrors (cudaMemcpy(com->yD, com->yH, sizeof(VTYPE)*N, cudaMemcpyHostToDevice));

```

3.1.3 粒子位置の時間積分

粒子には次に示すような回転速度場を与えることにします.

$$u(x_i, y_i, t) = -2 \cos\left(\frac{\pi t}{T}\right) \sin^2(\pi t) \cos(\pi t) \sin(\pi t), \quad (3.4)$$

$$v(x_i, y_i, t) = 2 \cos\left(\frac{\pi t}{T}\right) \cos(\pi t) \cos(\pi t) \sin^2(\pi t). \quad (3.5)$$

ここで, x, y は粒子の位置, t は時刻です. π は円周率, T は速度場の回転周期, 今回は $T = 8$ という定数を設定します. これを C 言語の関数として呼び出してもいいのですが. オーバーヘッドの少ないマクロとして次のように記述します.

```

#define TAU 8.0
#define US(x, y, t) (-2.0*cos(M_PI*(t)/TAU)*sin(M_PI*(x))*sin(M_PI*(x))*cos(M_PI*(y))*sin(M_PI*(y)))
#define VS(x, y, t) ( 2.0*cos(M_PI*(t)/TAU)*cos(M_PI*(x))*sin(M_PI*(x))*sin(M_PI*(y))*sin(M_PI*(y)))

```

CPU で 1 段のルンゲ・クッタ法の計算をする関数 “cpu_ppush1()” の実行は, 各構造体の変数を引数にして, 次のように記述します.

```
cpu_ppush1(com->xH, com->yH, com->xnH, com->ynH, cdo->time, cdo->dt);
```

一方、GPU でカーネル関数を実行する際は以下のように記述します。

```
dim3 Dg(N/256,1,1), Db(256,1,1);
gpu_ppush1<<< Dg, Db >>> (com->xD, com->yD, com->xnD, com->ynD, cdo->time, cdo->dt);
```

1 段ルンゲ・クッタ法で時間積分する計算を CPU で行うと、次のように記述できます。

```
void cpu_ppush1
//=====
// program : Particle push by 1-stage Runge-Kutta time integration
(
    VTYPE *x, /* x-coordinate of the particles */
    VTYPE *y, /* y-coordinate of the particles */
    VTYPE *xn, /* updated x-coordinate of the particles */
    VTYPE *yn, /* updated y-coordinate of the particles */
    VTYPE time, /* time */
    VTYPE dt /* time step interval */
)
//=====
{
    int j;
    VTYPE xt, yt;

    for (j = 0; j < N; j++) {
        xt = US(x[j], y[j], time);    yt = VS(x[j], y[j], time);
        xn[j] = x[j] + xt*dt;         yn[j] = y[j] + yt*dt;
    }
}
```

これに対応する GPU のカーネル関数は、次のように記述されます。

```

__global__ void gpu_ppush1
//=====================================================================
// program : Particle push by 1-stage Runge-Kutta time integration
(
    VTYPE *x, /* x-coordinate of the particles */
    VTYPE *y, /* y-coordinate of the particles */
    VTYPE *xn, /* updated x-coordinate of the particles */
    VTYPE *yn, /* updated y-coordinate of the particles */
    VTYPE time, /* time */
    VTYPE dt /* time step interval */
)
//=====================================================================
{
    int j, jx, jy;
    VTYPE xg, yg, xtdt, ytdt;

    j = blockDim.x*blockIdx.x + threadIdx.x;
    xg = x[j]; yg = y[j];
    xtdt = US(xg, yg, time)*dt; ytdt = VS(xg, yg, time)*dt;
    xn[j] = xg + xtdt; yn[j] = yg + ytdt;
}

```

カーネル関数では、for文の代わりに配列のインデックス計算が行われるだけで、後はCPUプログラムとほとんど同じです。

カーネル関数が終了した後はスレッドの同期をとるために“**cudaThreadSynchronize()**”を追記します。

```

dim3 Dg(N/256,1,1), Db(256,1,1);
gpu_ppush1<<< Dg, Db >>> (com->xD, com->yD, com->xnD, com->ynD, cdo->time, cdo->dt);
cudaThreadSynchronize(); ← この一文を追記

```

“cudaThreadSynchronize()”を入れる理由は、カーネル関数の実行が CPU 計算に対して基本的に非同期だからです。つまり、CPU の計算 (時間計測 <p.18> もその一つ) はカーネル関数の実行命令に対して、その終了を問いません。“gpu_ppush1<<< Dg, Db >>> ()” は実行が終了しないうちに次の CPU の処理に移ります。ただし、カーネル関数が連続で続く際は、次のカーネル関数を呼び出す際に thread が自動的に同期されるため、“cudaThreadSynchronize()”を入れる必要はありません（ちなみに、“cudaMemcpy” も自動同期してくれます）。

例)

```
gpu.function_1<<< Dg, Db >>> ( );
cudaThreadSynchronize();
gpu.function_2<<< Dg, Db >>> ( );
cudaThreadSynchronize();
gpu.function_3<<< Dg, Db >>> ( );
cudaThreadSynchronize();
```

次のステップに進める (時間発展させる) ために、xn, yn の値を x, y に格納し直す必要があります。CPU での処理は、

```
for (j = 0; j < N; j++) {
    com->xH[j] = com->xnH[j];    com->yH[j] = com->ynH[j];
}
```

とすれば良いのですが、これは “N*4*4” バイトのデータをホスト・メモリ間またはグローバル・メモリ間で転送しなければならず、効率的ではありません。xn, yn の配列のデータをそのままにし、“xn→x”, “yn→y” と考えれば良いです。つまり、下記のように、“xn” と “x”, “yn” と “y” の配列ポインタのアドレスを交換 (スワップ) するだけで良いです。

```
VTYPE *tmp;
tmp = com->xH; com->xH = com->xnH; com->xnH = tmp;
tmp = com->yH; com->yH = com->ynH; com->ynH = tmp;
```

デバイス・メモリも同様に、

```

VTYPE *tmp;

tmp = com->xD; com->xD = com->xnD; com->xnD = tmp;
tmp = com->yD; com->yD = com->ynD; com->ynD = tmp;

```

4段ルンゲ・クッタ法の数値解法についても同様です。プログラムの詳細はサンプル・プログラム内を見ればわかりますので、本資料での説明は割愛します。

3.1.4 プログラム実行

ファイルのコンパイルは、サンプル・プログラムの make ファイルに記載されている通りです。コマンドライン上で “make” を打つことで “src” 以下のソース・プログラムがコンパイルされ、カレント・ディレクトリ上に “xxx.o” という実行ファイルが作成されます。次に、 “./run” を打てば計算が開始され、“result” 以下に計算結果が出力されます。“make clean” で実行ファイルが消去され、“make clean-all” で “result” 以下の出力ファイルも含めて消去されます。

3.1.5 ファイルの出力 (BMP ファイルについて)

サンプル・プログラムでのファイル出力は、8ビット (=256階調) の BMP(Microsoft Windows Bitmap Image) 形式の画像ファイルとしました。BMP ファイル作成の詳細は、本資料の主旨と離れますが、興味のある方は各自調べるか、参考文献 [8] の p.157-p.159 を参照して下さい。ここでは、BMP ファイルの簡単な説明に留めます。画像ファイルにはいろいろな種類がありますが(例えば、PNG, JPEG, GIF, etc)，だいたいは画像の左上を視点に右下に向かってデータが記憶されています。しかし、BMP はこれを上下反転した形で記憶されているのが標準です (Fig.3.1a)。

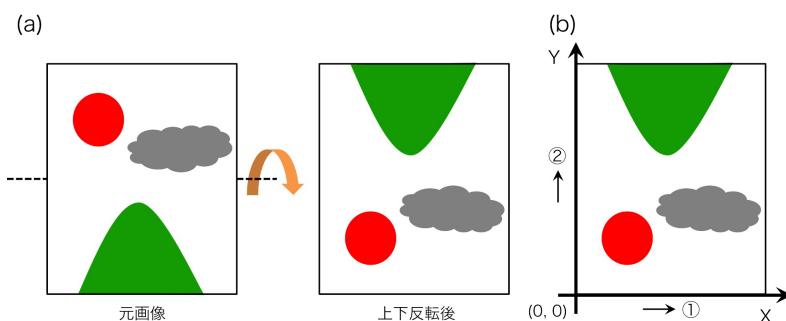


図 3.1 BMP ファイルのデータ記録形式。

XY 座標の始点 $(0, 0)$ は左下になります。反転後の画像の左下は元と画像の左上になっています。つまり、 $(0, 0)$ は元の画像の開始位置です。プログラムは $(0, 0)$ から描画を開始し、X 軸方向へデータを読み込んでいきます (Fig.3.1b の ① 方向)。X 軸方向にデータがなくなれば Y 軸方向に 1 つ進み (Fig.3.1b の ② 方向)，また X 軸方向へデータを読み込むのを繰り返します。上下反転した画像をこの順番で読み込めば、描画される画像が元の絵に戻ります。

3.1.6 演算性能

1 段ルンゲ・クッタ法で計算した結果を Fig.3.2a に示します。速度場を与える式は、数学的に可逆なので、time = 8.0 では、それぞれの粒子が完全に元の位置に戻らなくてはなりません。しかし、t = 8.0 の画像は t = 0.0 とは同じにならず、円からは程遠い形になっています (Fig.3.2a)。これは、ルンゲ・クッタ法の計算精度に原因があります。これは、離散化レベルの問題ですので、変数型を単精度 (float 型) から倍精度 (double) に変えたところで問題は解決しません。

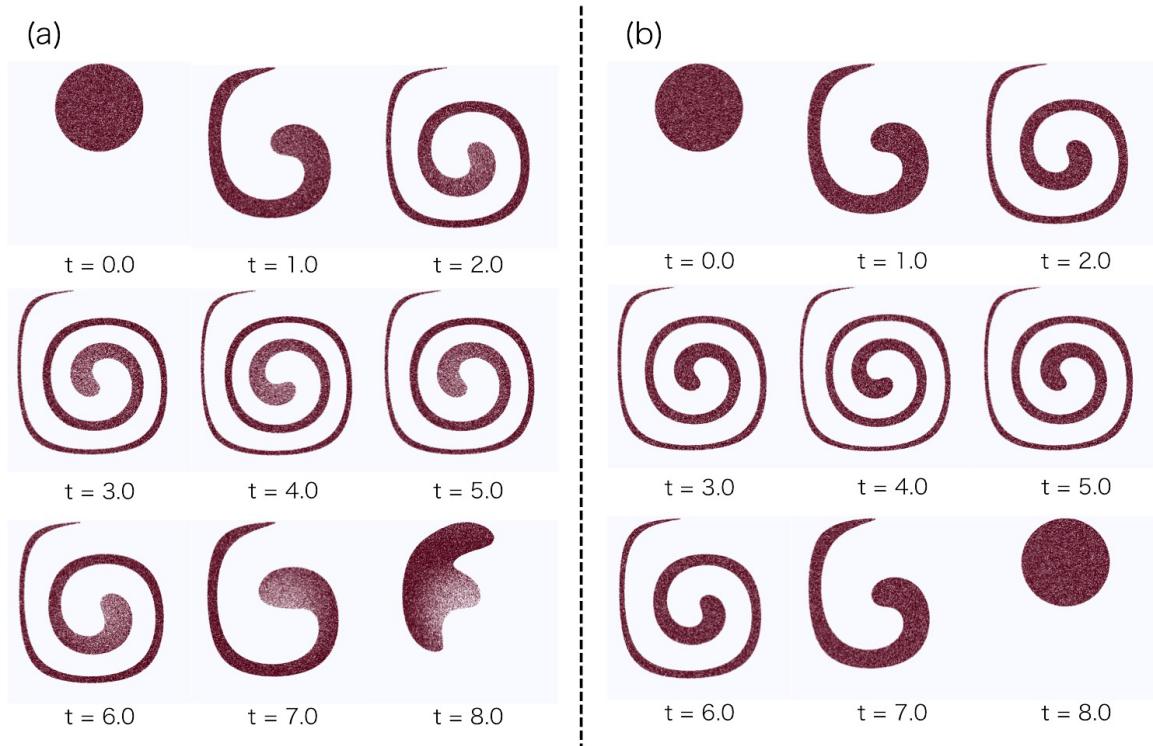


図 3.2 1 段ルンゲ・クッタ法で計算した粒子分布の時間変化。

サンプル・プログラムでは $dt = 0.01$ としていますが、もし、 $dt = 0.001$ として計算すれば、time = 8.0 まで計算するのに 10 倍の時間を要しますが、粒子分布は初期形状に非常に近くなります。

す。同じ $dt = 0.01$ でも、ルンゲ・クッタ法の段数を上げることで計算精度を上げることができます。サンプル・プログラム内で 1 段と 4 段の計算精度のスイッチは def.h で行います。“#define RUNGE_KUTTA4” の一文を active にするだけで OK です。同様に、CPU と GPU の計算スイッチも、def.h 内の “#GPU_COMP” と “#CPU_COMP” の一文を active にするか inactive(コメント文) にするだけで行えます。

4 段ルンゲ・クッタ法での計算結果は Fig.3.2b の通りです。Fig.3.2b では、 $t = 8.0$ でほぼ元の円形の分布に戻っています。計算精度が大切である理由がこれで理解出来るかと思います。

3.1.7 実行速度の比較

次に、GPU 計算と CPU 計算の演算性能を比較するために、 $t = 8.0$ まで計算するまでに必要な時間を測定してみます。計算時間の測定は、CUDA4.9 以前までは、“cutil.h” とういヘッダーファイルをインクルードして、以下のように記述していました。

```
(CUDA4.9 以前)

#include <cutil.h> ← Timer 関数を使用するためのヘッダーファイルをインクルード
int main (int argc, char *argv[])
{
    unsigned int timer; ← timer 変数を宣言
    CreateTimer(&timer);
    ResetTimer(&timer);
    StartTimer(&timer); ← 計測開始

    for () { main loop }

    cutStopTimer(timer); ← 計測修了
    VTYPE gpu_time = cutGetTimerValue(timer)*1.0e-03; ← 計測時間の単位を [sec] に
    するために 0.001 をかける
    printf("Elapsed Time= %9.3e [sec]\n", gpu_time);
    return 0;
}
```

CUDA5.0 以降は, “cutil.h” がなくなり, 代わりに “helper_cuda.h” と “helper_functions.h” をインクルードします。また, Timer 関数の記述も以下のように変更されます。

(CUDA5.0 以降)

```
#include <helper_cuda.h>
#include <helper_functions.h>
int main (int argc, char *argv[])
{
    StopWatchInterface *timer = NULL;
    sdkCreateTimer(&timer);
    sdkResetTimer(&timer);
    sdkStartTimer(&timer);

    for () { main loop }

    sdkStopTimer(timer);
    VTYPE gpu_time = sdkGetTimerValue(timer)*1.0e-03;
    printf("Elapsed Time= %9.3e [sec]\n", gpu_time);
    return 0;
}
```

CPU 計算に対する GPU の高速化倍率を計測時間を使って次のように定義します。速度計測する際は “main.cu” 内のファイル出力の一文と “printf()” の一文, さらに, ファイル出力の一文をコメント・アウトし, 純粋な演算時間だけを測定するようにしましょう。

$$\text{Acceleration rate} = 1.0 \left/ \frac{\text{Elapsed time by GPU [s]}}{\text{Elapsed time by CPU [s]}} \right. \quad (3.6)$$

1 段と 4 段ルンゲ・クッタ法での粒子に対する高速化倍率を結果を Fig.3.3 に示します。計算に使用したデバイスは GeForce GTX TITAN X, コンパイラーは CUDA-8.0 です。GPU の単純実装だけで, CPU に対して最大 40 倍の高速化が得られました。Fig.3.3 から, 粒子数が増加するほど CPU に対する高速化倍率が増加していることがわかります。これは, GPU のスレッドの使用率が限界近くまで達しているためです。

サンプル・プログラムでは、スレッド数を 256 と固定していますが、演算性能はスレッド数とブロック数に依存するので、GPU の特性を確認する意味でも各自でプログラム内の “Dg” と “Db” を変更して計算してみて下さい。ただし、1 ブロックあたりのスレッド数は **512** 以下とすることに留意して下さい（この制限はデバイスによって異なります）。ブロック内のスレッド数は多いほど演算性能は向上することを実感できるはずです。

スレッド数とブロック数に対する演算性能 (Flops) の変化については、参考文献 [8] の p.186, p.197-p.199, p.206, p.211 を参照して下さい。

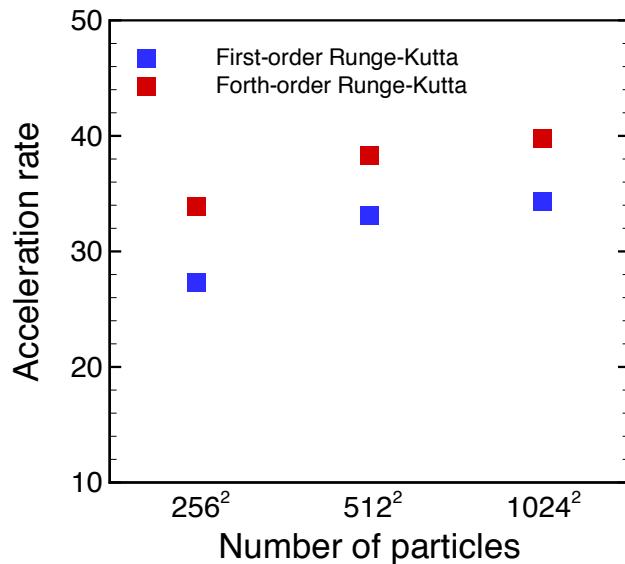


図 3.3 CPU に対する加速化倍率 (Single precision, Device: GeForce GTX TITAN X).

3.2 偏微分方程式 (拡散方程式)

次に、偏微分方程式の計算を考えます。偏微分方程式の例として、次式で示される拡散方程式を取り上げたいと思います。拡散現象が z 方向に一様であった場合、式中のラプラシアンは二次元の二階微分となります。関数 f は例えばコップの水に垂らしたインクの濃度などを表します。

$$\frac{\partial f}{\partial t} = \kappa \Delta f = \kappa \left(\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \right) \quad (3.7)$$

3.2.1 初期条件

二次元の計算領域は、簡単のため “ $0 \leq x \leq 1$ ”, “ $0 \leq y \leq 1$ ” とします。濃度分布 $f(x, y)$ の初期状態 ($t = 0$) を次式で定義します。

$$f(x, y) = \exp \left[-\alpha \left\{ (x - 0.5)^2 + (y - 0.5)^2 \right\} \right], \quad \alpha = 30.0 \quad (3.8)$$

3.2.2 計算手法

二次元の計算領域に対して、 x 方向に nx , y 方向に ny の格子数を設定します。 x 方向の格子番号を “ i ”, y 方向の格子番号を “ j ” としたとき、格子点 (i, j) の場所での濃度分布を “ $f_{i,j}$ ” と表記することにします。また、時間方向については、時間ステップを上付添字で表すことになります。時間と空間の離散化に基づいて拡散方程式を有限差分法で離散化すると、

$$\begin{aligned} f_{i,j}^{n+1} &= f_{i,j}^n + \kappa \left(\frac{f_{i+1,j}^n - 2f_{i,j}^n + f_{i-1,j}^n}{\Delta x^2} + \frac{f_{i,j+1}^n - 2f_{i,j}^n + f_{i,j-1}^n}{\Delta y^2} \right) \Delta t \\ &= (1 - 2c_0 - 2c_1) f_{i,j}^n + c_0 f_{i+1,j}^n + c_0 f_{i-1,j}^n + c_1 f_{i,j+1}^n + c_1 f_{i,j-1}^n \\ &= c_0 (f_{i+1,j}^n + f_{i-1,j}^n) + c_1 (f_{i,j+1}^n + c_1 f_{i,j-1}^n) + c_2 f_{i,j}^n \end{aligned} \quad (3.9)$$

$$\text{where, } c_0 = \frac{\kappa \Delta t}{\Delta x^2}, \quad c_1 = \frac{\kappa \Delta t}{\Delta y^2}, \quad c_2 = 1 - 2 \left(\frac{\kappa \Delta t}{\Delta x^2} + \frac{\kappa \Delta t}{\Delta y^2} \right) \quad (3.10)$$

となります。空間については二次の中心差分法を適用しています。

3.2.3 境界条件

領域の端はノイマン境界（境界に垂直方向に濃度勾配 0）を設定します。

$$\frac{\partial f}{\partial x} = 0, \quad \frac{\partial f}{\partial y} = 0.$$

$$\rightarrow f_{0,j}^n = f_{1,j}^n, \quad f_{nx-1,j}^n = f_{nx-2,j}^n, \quad f_{i,0}^n = f_{i,1}^n, \quad f_{i,ny-1}^n = f_{i,ny-2}^n. \quad (3.11)$$

3.2.4 GPU 計算のプログラム

前述までにまとめた離散式は 1 格子に対して計算されます。そこで、1 格子当たりの計算を 1 スレッドで行うことを考えます。常微分方程式の数値解法と同様に、はじめに 1 ブロックあたりのスレッド数を決定し、その後、ブロック数を決定します。カーネル関数では “ $nx * ny$ ” 個のスレッドが実行されることになります。

“n+1” ステップ目の濃度分布 (f^{n+1}) を変数を使ってデータを格納すれば ($\text{fcc} = f_{i,j}^n$, $\text{fce} = f_{i+1,j}^n$, $\text{fcw} = f_{i-1,j}^n$, $\text{fcn} = f_{i,j+1}^n$, $\text{fcs} = f_{i,j-1}^n$), 境界条件の分岐を入れるのが容易になります (Fig.3.4)。

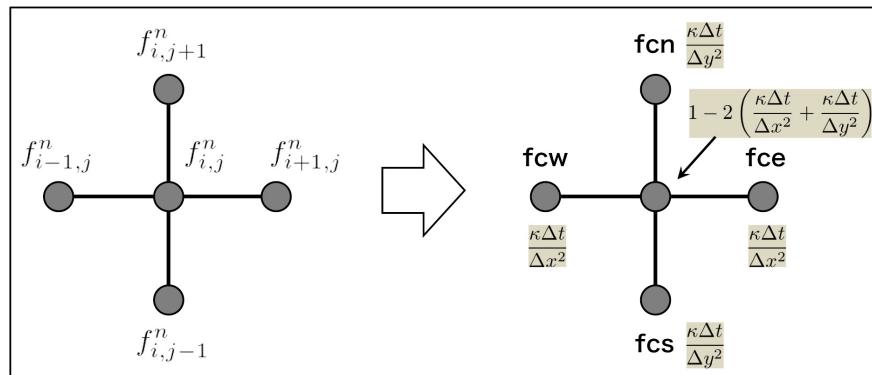


図 3.4 隣接格子点のデータを変数として格納。

上述までの計算は、サンプル・プログラム内では、“cuda_diffusion2d_0 <<< >>> ()” として記載されています。拡散方程式の計算した結果を Fig.3.5 の通りです (格子数は $nx * ny=2562$)。時間発展とともに濃度が拡散し、領域全体が均一になっていく様子がわかります。

3.2.5 演算性能

常微分方程式の計算同様、演算性能を計測してみます。今回は演算性能の指標として、単位時間あたりの浮動小数点演算回数である Flops(Floating-point operation per second) を導入します。プログラム内の関数 diffusion2d() の戻り値を n から n+1 ステップを計算するために行われた浮動小数点演算の回数とします。ここで、先ほど離散化した拡散方程式をもう一度見てみると、

$$\text{fn}[j] = c0 * (\text{fce} + \text{fcw}) + c1 * (\text{fcn} + \text{fcs}) + c2 * \text{fcc};$$

であり、1 格子点当たり 7 回の浮動小数点演算があるので、関数 “diffusion2d()” の戻り値は “ $7 * nx * ny$ ” となります。プログラム上での変数 “flops” は、実行されたステップ数 (N) までの浮動小数点演算回数の総和となります。したがって、演算性能 (単位 [GFlops]) は以下の式で表されます。

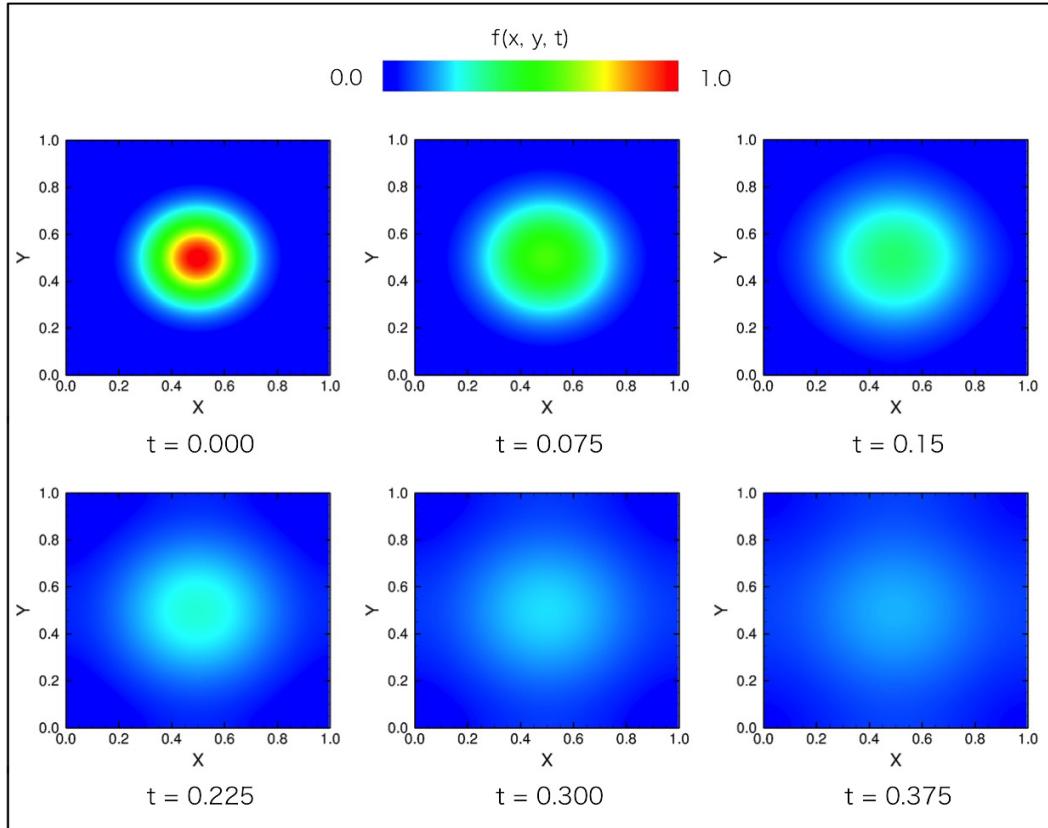


図 3.5 2 次元拡散方程式を計算した結果。

$$\text{Performance} = \frac{N \times (7 \times nx \times ny)}{\text{Elapsed time [s]}} \times 10^{-9} [\text{GFlops}] \quad (3.12)$$

CPU 計算と GPU 計算での格子点に対する GFlops の値を Fig.3.6 に示します。CPU 計算では約 3GFlops に対し、GPU 計算では最大 150GFlops を示し、CPU 計算に対しておよそ 50 倍の高速化を実現しました。また、計算に使うスレッドの数が増加するほど高い演算性能を発揮する傾向は、先ほどの常微分方程式の計算と同様です。

3.2.6 高速化の基本概念

ここで、GPU 計算の 1 スレッドが行う計算内容について考えてみます。1 格子での $n+1$ ステップの値 (f_n) を計算する司令文を見てみると、グローバル・メモリ上の配列要素にアクセスし、“ $fcc = f[j]$, $fcn = f[j+nx]$, $fcs = f[j-nx]$, $fce = f[j+1]$, $fcw = f[j-1]$ ” の 5 点のデータを読み込みます。その後、7 回の浮動小数点演算を行い、グローバル・メモリ “ $fn[j]$ ” に書き込みます。“ $(5+1)*4 = 24\text{Byte}$ ” のデータを読み込むのに対し、浮動小数点演算は 7 回しかありません。

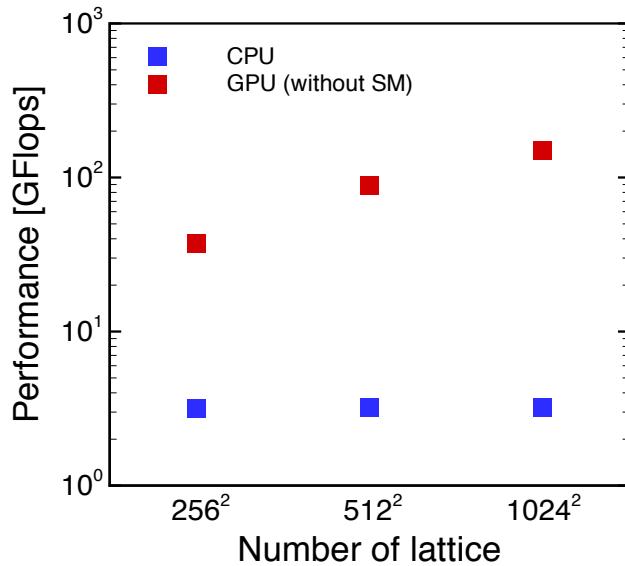


図 3.6 2 格子点に対する演算性能 [GFlops] (Single precision, Device: GeForce GTX TITAN X).

GPU は、第 1 章でも述べた通り、演算速度も速く、データ転送速度も速いのですが、**1Byte** のデータをグローバル・メモリに転送する時間は、1 回の小数点演算よりも 100 倍以上の時間がかかります。つまり、拡散方程式は、データ転送速度が全体の計算時間をほとんど決定する、と言っても過言ではありません。グローバル・メモリへのアクセスがコアレッシング (Fig.3.7) の条件を満たすかどうかによって、データ転送速度は大きく変わります。blockDim_x が 16 の倍数ならば、“ $fcc = f[j], fcn = f[j+nx], fcs = f[j-nx]$ ” の 3 回のアクセスは、コアレッシングになります。そのため、blockDim_x が 16 以上に指定した場合、速い計算速度が得られます。

コアレッシングされていないメモリアクセス

	各スレッドが1番目に読み込む要素	各スレッドが2番目に読み込む要素	各スレッドが3番目に読み込む要素	各スレッドが4番目に読み込む要素
スレッドが0が読み込む要素	(0,0)	(0,1)	(0,2)	(0,3)
スレッドが1が読み込む要素	(1,0)	(1,1)	(1,2)	(1,3)
スレッドが2が読み込む要素	(2,0)	(2,1)	(2,2)	(2,3)
スレッドが3が読み込む要素	(3,0)	(3,1)	(3,2)	(3,3)

各スレッドが同時に読み込む要素が隣り合っておらず、メモリアクセスに遅延が生じる。

コアレッシングされたメモリアクセス

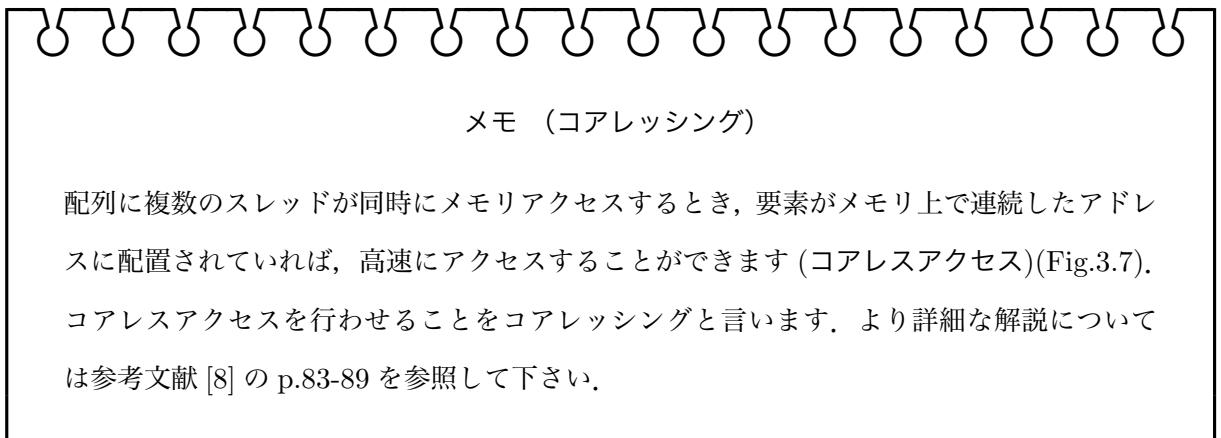
スレッドが0が読み込む要素	スレッドが1が読み込む要素	スレッドが2が読み込む要素	スレッドが3が読み込む要素	
(0,0)	(0,1)	(0,2)	(0,3)	各スレッドが1番目に読み込む要素
(1,0)	(1,1)	(1,2)	(1,3)	各スレッドが2番目に読み込む要素
(2,0)	(2,1)	(2,2)	(2,3)	各スレッドが2番目に読み込む要素
(3,0)	(3,1)	(3,2)	(3,3)	各スレッドが2番目に読み込む要素

各スレッドが同時に読み込む要素が隣り合っており、コアレスアクセスがされている。

図 3.7 (左) コアレッシングされていないメモリアクセスと、(右) コアレッシングされたメモリアクセス。

拡散方程式のような簡単な計算ではメモリ・アクセスが計算律速になっていると述べましたが、実は多くの計算でそうなっています。そのため、演算部分を高速化することよりも、データ転送の部分を効率化することが有効です。

データ・アクセスの高速化の条件として、グローバル・メモリへのアクセスをコアレッシングにすることは当然です。グローバル・メモリからデータ転送レートをさらに上げることは困難ですから、データ転送量を減らすことができるかどうかを考えます。次節から、データ転送量を減らす方法として、シェアード・メモリの利用をとりあげます。



3.3 シェアード・メモリの利用

拡散方程式の計算では、 j 点を計算するために、 $f[j], f[j+1], f[j-1], f[j+nx], f[j-nx]$ の 5 点のデータを読み込みます。次に、隣接する “ $j+1$ ” 点を計算するには、 $f[j+1], f[j+2], f[j], f[j+nx+1], f[j-nx+1]$ の 5 点を読み込みます。

“ j ” 点の計算と “ $j+1$ ” 点の計算では、同じ “ $f[j]$ ” の値が使われています。もし、“ j ” 点を計算するスレッドで読み込んだ “ $f[j]$ ” の値を “ $j+1$ ” の点を計算するスレッドで使うことが出来れば、“ $j+1$ ” 点を計算するスレッドでは、“ $f[j]$ ” の値をグローバル・メモリに取りに行かなくてもいいことになります。

“ $j+nx$ ” 点の値、 “ $j-nx$ ” 点の値についても同じことが言えます。何も工夫しない限り、“ j ” 点の “ $n+1$ ” の値 ($f[n]$) を計算するためには、5 回のデータの読み込みと 1 回のデータの書き込みで、計 6 回のグローバル・メモリへのアクセスが必要になりますが、読み込んだデータをスレッド間で共有出来れば、上で述べた理屈だとデータ転送量を三分の一にまで減らせ、高速化が期待できます。

ここで利用するのが、ストリーミング・マルチプロセッサ (SM) の中にある 16kByte のシェアード・メモリ (p.3 の Fig.1.2) です。ブロック内に配置されたスレッドは同一 SM 上で計算されるので、シェアード・メモリを利用し、スレッド間でデータを共有することができます。

3.3.1 シェアード・メモリの宣言とアクセス

まず、シェアード・メモリ上への配列の確保から行います。簡単のため、ブロック内のスレッドを次のように設定します。

```
dim3 Db(16, 16, 1);
```

つまり、全計算領域 “ $nx*ny$ ” の範囲の中の一部の “ $16*16$ ” という範囲を 1 つのブロックが担当し、その中には “ $16*16$ ” 個のスレッドが走ります。次に、カーネル関数の中にシェアード・メモリを使った配列の宣言を行います。

```
_shared_ float fs[16 + 2][16 + 2];
```

これで、静的にシェアード・メモリ上に “ $fs[16+2][16+2]$ ” というサイズの配列 ($18*18*4=1,296\text{Byte} < 16\text{kByte}$) が確保されます。この配列はブロックごとにそれぞれ確保され、スレッドごとに確保されるわけではありません。“ $16+2$ ” の “ $+2$ ” をつけた理由は、ブロックの境界格子の計算をやりやすくするためです (詳細は後述; p.35)。

シェアード・メモリのアクセス速度はレジスタと同程度と言われ、非常に高速です。グローバル・メモリ上の $nx*ny$ 格子点のデータの一部をそっくりブロック内のシェアード・メモリ上に持ってきてしまい、後は各スレッドからシェアード・メモリにアクセスするという方法を考えます。

グリッドの中のブロックも 2 次元的に配置しているとすると (Fig.3.8)，グローバル・メモリ上の j 点は，“ $threadIdx.x (= 0 \sim 15)$ ” と “ $threadIdx.y (= 0 \sim 15)$ ” を用いて次のように 1 次元配列指定できます。

```
jy = blockDim.y * blockIdx.y + threadIdx.y;
jx = blockDim.x * blockIdx.x + threadIdx.x;
j  = nx * jy + jx;
```

“ $threadIdx.x = 0$ ” と “ $threadIdx.y = 0$ ” のスレッドが読み込んだデータをあとで説明する “ $fs[1][1]$ ” に書き込みたいので、以下のように記述します。

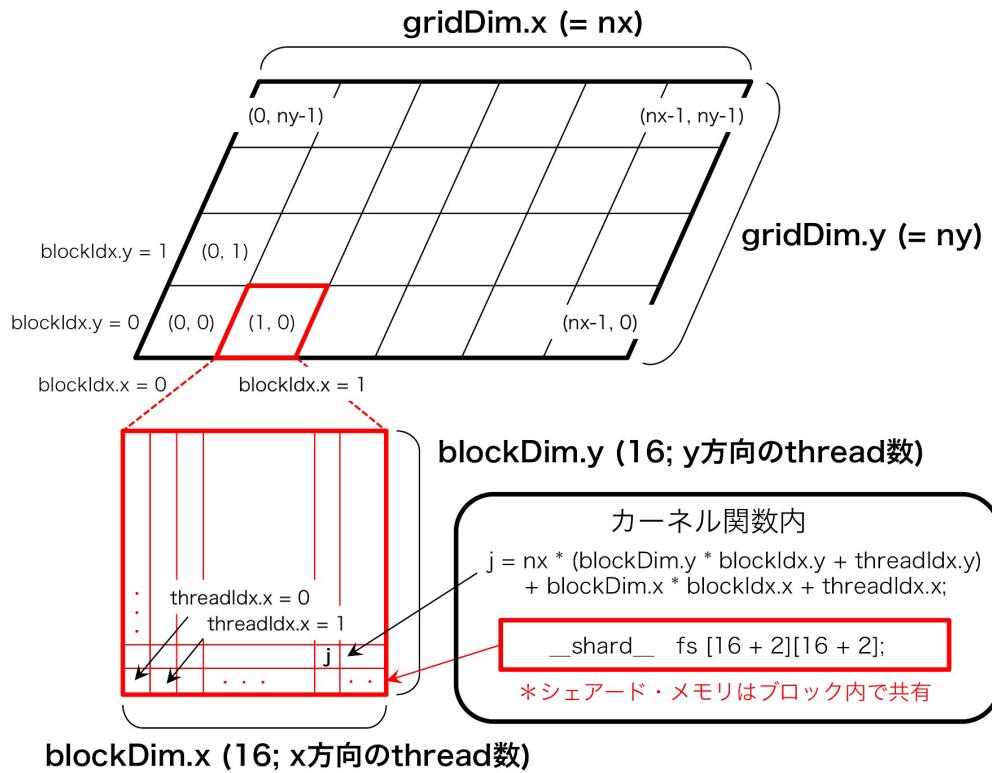


図 3.8 2 次元的に配置したブロックとその中のスレッド。

```
fcc = f[j];
fs[threadIdx.y + 1][threadIdx.x + 1] = fcc;
```

この行が実行されると，“ $16*16(= 256)$ ” のデータがシェアード・メモリ上の配列に書き込まれます。

“`fs[][]`” に読み込んでしまえば、

```
int j = blockDim.x * blockIdx.x + threadIdx.x;
fcc = f[j];
fce = f[j + 1];
fcw = f[j - 1];
fcn = f[j + nx];
fcs = f[j - nx];
```

としていたものを

```

fcc = fs[threadIdx.y + 1][threadIdx.x + 1];
fce = fs[threadIdx.y + 1][threadIdx.x + 2];
fcw = fs[threadIdx.y + 1][threadIdx.x + 0];
fcn = fs[threadIdx.y + 2][threadIdx.x + 1];
fcs = fs[threadIdx.y + 0][threadIdx.x + 1];

```

とするだけで、

```
fn[ j ] = c0 * (fce + fcw) + c1 * (fcn + fcs) + c2 * fcc;
```

が計算できます。

3.3.2 ブロックの境界格子の計算

ブロック内で “16*16” の計算領域の中央部分を計算する場合は問題ありませんが、例えば左端の格子点は “fs[j-1]” が計算領域の外に出てしまいます。計算領域全体に対する境界条件と類似のことをブロック内の計算に対しても考えなくてはいけません。

シェアード・メモリを使う場合は、敢えて “fs[16+2][16+2]” として “16*16” の上下左右に 1 格子点分のデータを格納するための袖領域も付け加えます。これにより、 “fs[][]” へのアクセスには境界領域かどうかの条件分岐が不要になります。

まず、左端の袖領域にグローバル・メモリ上の “f[]” からデータを読み込みます。“threadIdx.x = 0” の 16 個のスレッド (“threadIdx.y = 0~15”) に左端の配列要素への読み込みも行わせます。ここで、

```
int    jx = threadIdx.x + 1,    jy = threadIdx.y + 1;
```

と置くことにします。そのブロックがグリッドの中で左端にある (blockIdx.x = 0) の場合、そこは壁境界になるため、グローバル・メモリに読みに行く必要がなく、自分自身の値 (fcc) を隣にコピーすればいいことになります。つまり、

```

if (threadIdx.x == 0) {
    if (blockIdx.x == 0) fs[jy][0] = fcc;
    else                 fs[jy][0] = f[j - 1];
}

```

となります。右側の壁境界の処理も同様に、

```

if (threadIdx.x == blockDim.x - 1) {
    if (blockIdx.x == gridDim.x - 1) fs[jy][blockDim.x + 1] = fcc;
    else                           fs[jy][blockDim.x + 1] = f[j + 1];
}

```

です。下側の壁境界の処理は、

```

if (threadIdx.y == 0) {
    if (blockIdx.y == 0) fs[0][jx] = fcc;
    else                 fs[0][jx] = f[j - nx];
}

```

となります。最後に上側の壁境界の処理は、

```

if (threadIdx.y == blockDim.y - 1) {
    if (blockIdx.y == gridDim.y - 1) fs[blockDim.y + 1][jx] = fcc;
    else                           fs[blockDim.y + 1][jx] = f[j + nx];
}

```

となります。ここで重要なのは、“n+1”ステップのデータ ($fn[j]$) を書き込む前に、ブロック内のスレッドの同期を取ることです。この理由は、スレッドのインストラクションが処理される順序は全く決まっていないからです。したがって、以下のようにカーネル関数内でスレッドの同期をとります。

`--syncthreads();` ← 書き込む前に必ず同期をとる。

$fn[j] = c0 * (fce + fcw) + c1 * (fcn + fcs) + c2 * fcc;$

シェアード・メモリを利用してグローバル・メモリへのアクセス回数を三分の一に減らしたカーネル関数は、サンプル・コードの “cuda_diffusion2_1<<< >>> ()” の通りです。

3.3.3 シェアード・メモリの節約

これまでの計算では、1つのスレッドが1格子点の計算を担当していました。この制約は強すぎるるので、1つのスレッドが m 個の格子点を計算することを考えます。もし、ブロック内に 256 のスレッドを設けたならば、約 $\text{mkByte} = 256 * 4 * m = 1024 * m$ Byte のシェアード・メモリを使うことになります。16kByte まで使うことができるとすれば、 m は 16 より少なくななければなりません。さらに変数がたくさんある場合や 3 次元計算のことを考えると、シェアード・メモリを節約しなければなりません。

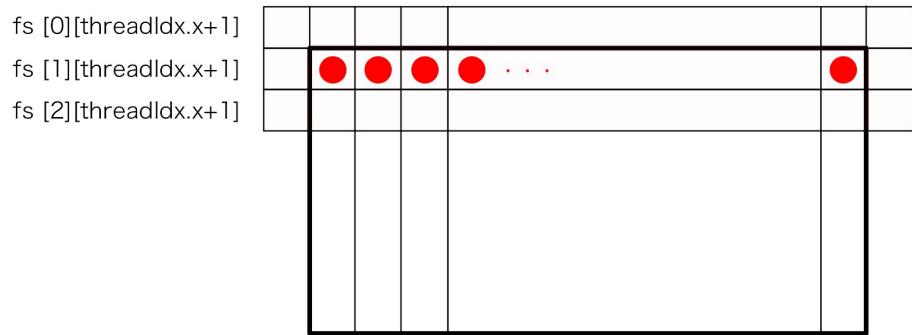
そこで、1スレッドに複数の格子点を計算させることを考えます。ブロック内に 1 列だけ “blockDim_x” 個のスレッドを配置します。1スレッドが計算する格子点を y 方向に “blockDim_y” 個とすると、1 ブロックが担当する格子点数は “blockDim_x * blockDim_y” 個となります (変数 “blockDim_x” と “blockDim_y” は、サンプル・プログラムでは、ヘッダー・ファイル “def.h” の中で定義されています)。つまり、 “for (jy = 0; jy < blockDim_y; jy++)” というループがスレッドの中に入ります。

拡散方程式の隣接格子点へのアクセスは、上下左右に 1 点しか参照しません。今、x 方向にはスレッドを並べていて同時に実行されるので、シェアード・メモリの x 方向のサイズは “blockDim_x+2” となります。y 方向については、for 文で順に計算しているので、 “(jx, jy)” 番目の格子点を計算しているとき、y 方向の参照格子は “(jx, jy+1)” と “(jx, jy-1)” だけです。シェアード・メモリの y 方向のサイズは 3 列分あればいいので、

```
_shard_    float    fs[3][blockDim_x + 2];
```

と静的に確保し、この “fs” を有効に使いまわします。“jy = 0” の 1 列の格子点を計算する際、グローバル・メモリからシェアード・メモリに値お読み込む部分は Fig.3.9 の赤色の 3 列です。「●」は計算しているスレッドを表しています。

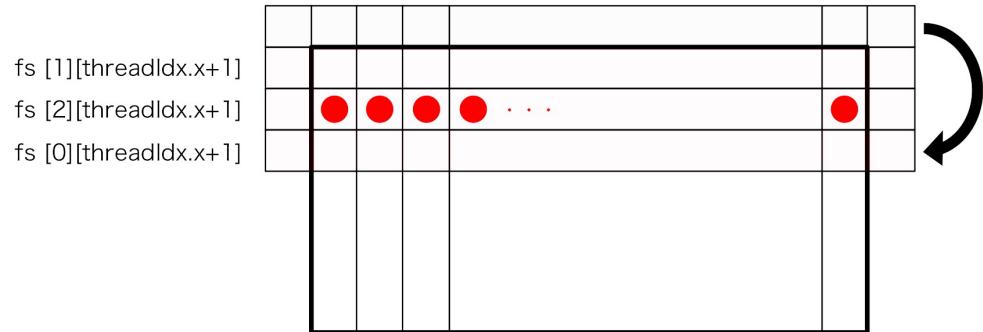
“jy = 0” の 1 列の格子点を計算した次は、“jy = 1” の 1 列の計算を行います。y 方向へのアクセスは “jy = 0, jy=1, jy=2” になりますが、“jy = 0” と “jy = 1” の f の値は “fs” に読み込んでいるので、“jy = 2” のデータだけを新たにグローバル・メモリから読み込めばいいことになります。ここで、“jy = 2” の値を読み込むためのシェアード・メモリの配列を用意するのではなく、一つ前の段

図 3.9 $j_y = 0$ の 1 列の格子点を計算する際のシェアード・メモリへのデータの読み込み。

の計算に使った配列データ,

`fs[0][blockDim_x + 2];`

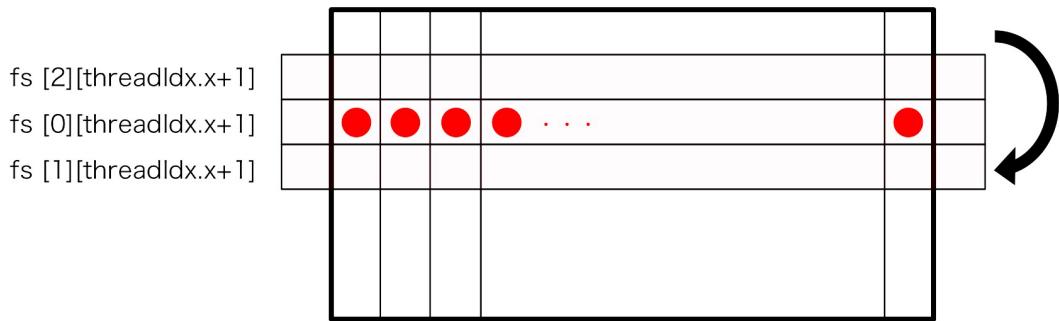
はもう必要なくなったので、この配列を “ $j_y = 2$ ” のデータを入れるために使うことにします (Fig.3.10).

図 3.10 $j_y = 1$ の 1 格子点を計算。

“ $j_y = 2$ ” を計算するときにも同じように不要となった “`fs[1][blockDim_x + 2]`” を “ $j_y = 3$ ” のデータを読み込むために使います (Fig.3.11). このように、シェアード・メモリを使いまわすことによって、使用量を大幅に削減することができます。この計算は、サンプル・プログラムの “`cuda_diffusion2d_2 <<< >>> ()`” として記述されています。

3.3.4 変数 (レジスタ) の利用

前述の計算では、各スレッドを x 方向に 1 列に並べ、 y 方向に順次計算することによってシェアード・メモリを 3 列だけで済ませることができました。しかし、よく考えてみると、シェード・メモリに入れたデータの中で、各スレッドが担当する格子を計算するとき、他のスレッドがグローバル・メ

図 3.11 $jy=2$ の 1 格子点を計算.

モリから読み込んできた値を使うのは x 方向の隣接格子の値を参照するときだけです。

そこで、 y 方向の隣接格子を参照する際に y 方向の 1 つ上の格子で使った値を再利用するために、スレッド内でローカルに 3 つの変数を宣言し、それを使うことにします (Fig.3.12)。この変数のメモリは、レジスタ (Fig.1.2, Fig.2.2) に確保されます。

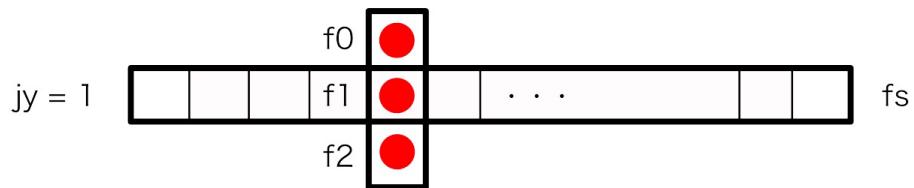


図 3.12 シェアード・メモリとレジスタを使用した場合の隣接格子へのアクセス。

シェアード・メモリとしては、隣接格子がグローバル・メモリから読み込んだ値を使うのに必要なサイズである横一行分を確保すれば十分です。したがって、次のようにローカル変数とシェアード・メモリを使った配列を宣言します。

```
float f0, f1, f2;
__shard__ float fs[blockDim_x + 2];
```

“blockDim_x + 2” と “+2” としてあるのは、同じく左右のブロック境界の袖領域のためです。

```
fs[threadIdx.x + 1] = f1;
```

とし、さらに、“fs[threadIdx.x]” には左隣の格子の値、“fs[threadIdx.x+2]” には右隣の格子の値が入っています。この状態ならば $n+1$ ステップの値を計算を以下のようにまとめることができます。

```

fn[j] = c0 * (fs[threadIdx.x] + fs[threadIdx.x + 2])
+ c1 * (f0 + f2)
+ c2 * f;

```

“ $jy+1$ ” の格子点の “ $n+1$ ” の値を計算するためには “ $jy+2$ ” の値が必要になり、それをグローバル・メモリから変数 “ $f3$ ” に読み込みたいところです。しかし、“ $f0$ ” は必要なくなったので、上述のシェアード・メモリの使い回しを考えます (Fig.3.13)。

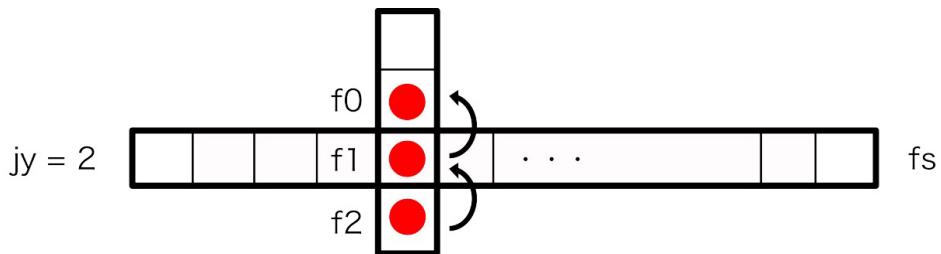


図 3.13 レジスタの再利用。

変数の値はレジスタに保存されているので、書き換えや値の交換は非常に高速に行うことが出来ます。“ $f1 \rightarrow f0$ ”, “ $f2 \rightarrow f1$ ” として値を読み込みます。こうすることによって、再び “ $fs[threadIdx.x + 1] = f1$ ” とするだけで、隣接格子の値にアクセスすることができるようになり、“ $jy+1$ ” の格子の “ $n+1$ ” の値を計算することができます。

計算領域の分割やブロックの切り方、その中のスレッドの配置などは前述の内容と全く同じです。シェアード・メモリとレジスタを使用した計算は “`cuda_diffusion_3 <<< >>> ()`” としてサンプル・プログラム内に記述されています。

3.3.5 演算性能の比較

前述までに示した 4 つのカーネル関数 (`cuda_diffusion2d_0`, `cuda_diffusion2d_1`, `cuda_diffusion2d_2`, `cuda_diffusion2d_3`) の演算性能を比較してみます。Fig.3.14 で示したように、それぞれのカーネル関数での格子点数に対する GFlops を Fig.3.14 に示します。

1 ブロックあたりのスレッドの数が多いほど演算性能が向上するというこれまでの知識を踏まえ、“`blockDim.x = 128`”, “`blockDim.y = 16`” (合計 512 thread/block) とします。Fig.3.14 に示す通り、格子点数の数が多いほどシェアード・メモリを使った恩恵がよく表れてきます。また、計算に使用したデバイスは GeForce GTX TITAN X で、もともとメモリ・バンド域が広く (480GB/sec),

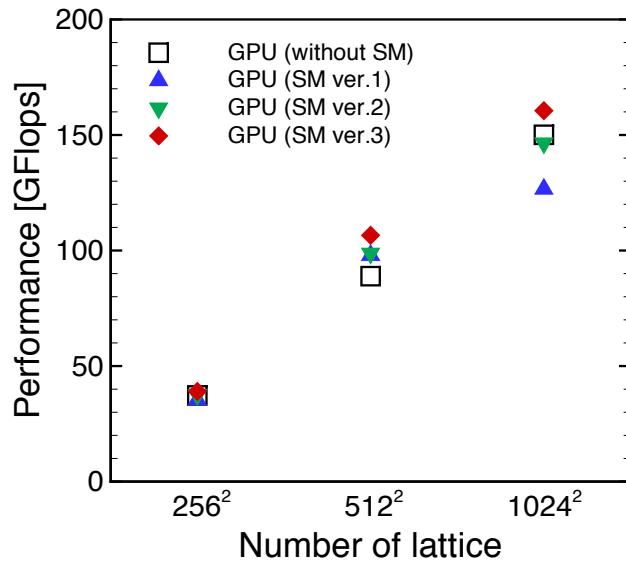


図 3.14 シェアード・メモリを使用したカーネル関数での拡散計算の演算性能 [GFlops] (Single precision, Device: GeForce GTX TITAN X).

演算律速よりもメモリ律速になる拡散方程式のような計算では、このメモリ・バンド帯域に演算性能が引っ張られてしまいます。シェアード・メモリを使ってもそれほど高速化がされていないのはそのためです。

ところで、シェアード・メモリの容量は GPU の世代によって異なります。例えば、GeForce GTX TITAN X のシェアード・メモリの最大容量は 49.152kByte です (デバイス上のシェード・メモリの容量は “./deviceQuery” というコマンドで確認することができます; p.46)。最新のデバイスほど使用できるシェアード・メモリも大きくなっていますので、デバイスの特徴を意識した CUDA プログラミングが重要です。

付録

A. CUDA インストール方法

A.1. CUDA8.0 のインストール

NVIDIA のサイト (<https://developer.nvidia.com/cuda-downloads>) から CUDA Toolkit をダウンロードし (Fig.15), 各マシンでインストール作業を行って下さい。最新バージョン (2017年6月現在) としては、CUDA8.0 がリリースされています。CUDA をインストールするにあたり、C コンパイラ (gcc) もあわせてインストールする必要があります。本資料では、C コンパイラのインストール方法は割愛します。

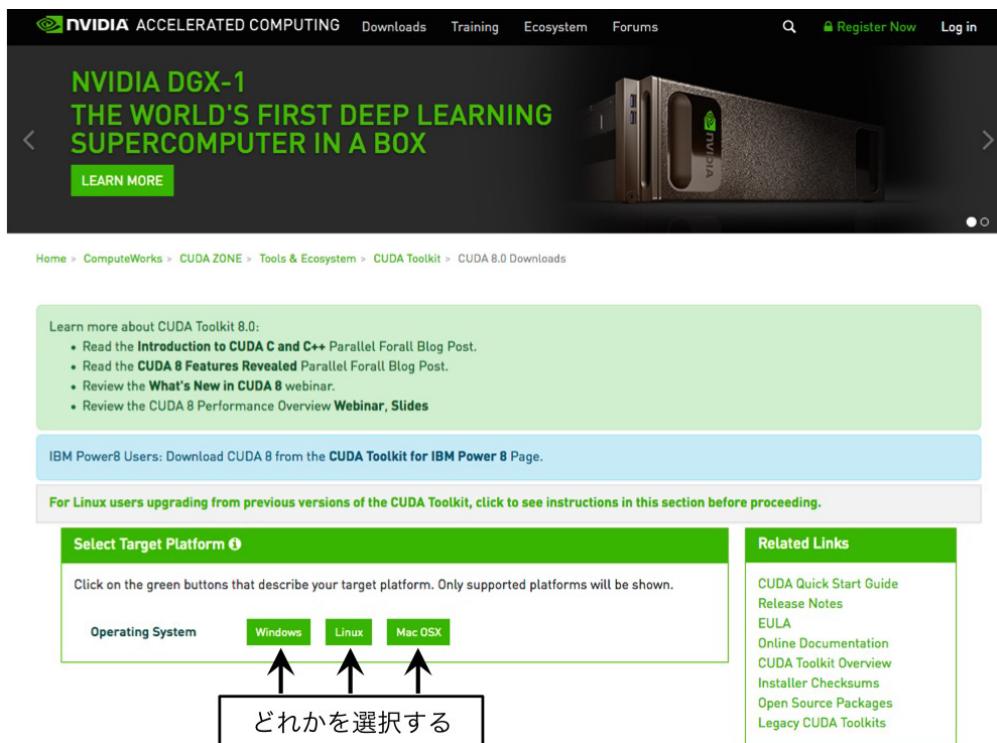


図 15 NVIDIA の web ページ (<https://developer.nvidia.com/cuda-downloads>)。

A.2. Linux(CentOS) での CUDA8.0 のインストール

Linux(CentOS 6 または 7) におけるインストール方法を以下に説明します。インストール作業は全て root 権限で行います。

CUDA7.0 から、ネットワーク・インストール用の rpm ファイルとローカル・インストール用の rpm パッケージの二種類が提供されています。ネットワーク・インストールは手軽ですが、途中、ネットワーク回線の混雑の影響でインストールが失敗する場合があります。基本的には、ローカル・インストール用(約 1.4GB)のファイルをダウンロードし(Fig.16), rpm でのインストールを行った方が安全です。本資料でも、基本的に “rpm(local)” によるインストール方法を取り上げます。

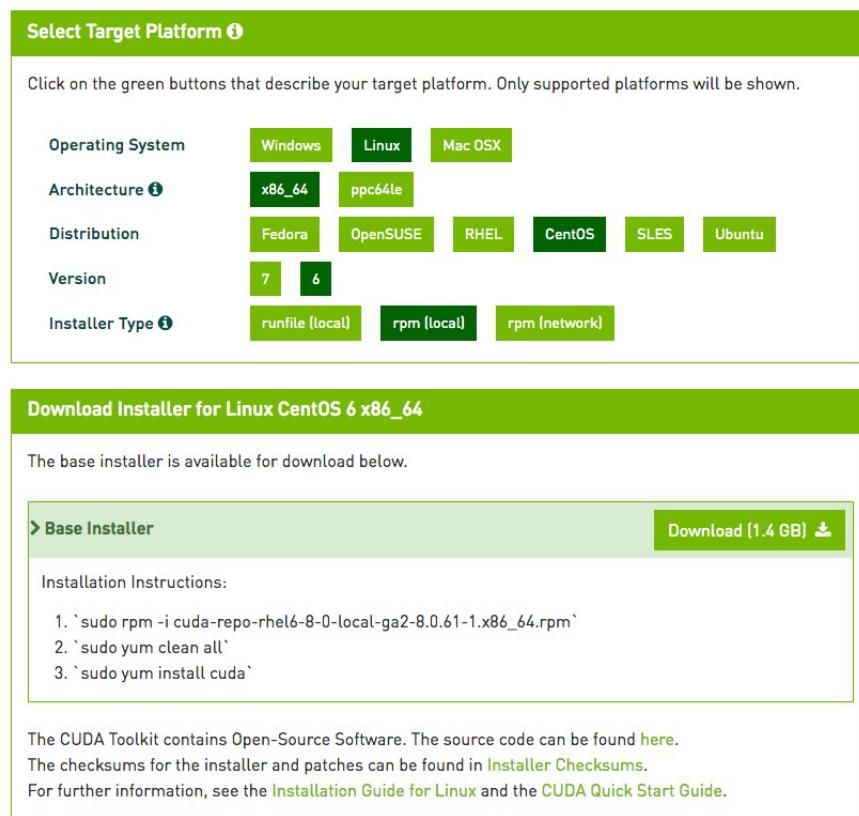


図 16 CUDA Toolkit (rpm) のダウンロード。

まず、GPU がデスクトップのマザーボード上の PCI Express に正しく挿入されているかを確認します。

```
$ lspci | grep -i nvidia;
```

付録

次に、Linux のバージョンを確認します。CUDA8.0 は CentOS 6.x と 7.x に対応しています。

```
$ uname -m && cat /etc/*release
```

CUDA8.0 は gcc 4.8.2 以降、kernel 3.10 以降が必要とされておりますが、gcc 4.4.7、kernel 2.6 でもインストールすることができました。それぞれのバージョンを確認するコマンドは次の通りです。

```
$ gcc --version
```

```
$ uname -r
```

CUDA ドライバーインストールには、カーネルと同じバージョンの kernel headers と development packages が必要です。今の例ですと、“2.6.32-358.14.1.el6”がインストールされていなければなりません (Fig.17)。確認方法は以下の通りです。

```
[takeishi@blood] % uname -r  
2.6.32-358.14.1.el6.x86_64  
[takeishi@blood] % yum list installed | grep kernel  
abrt-addon-kerneloops.x86_64  
dracut-kernel.noarch      004-303.el6      @base  
kernel.x86_64            2.6.32-220.el6    @anaconda-CentOS-201112091719.x86_64/6.2  
kernel.x86_64            2.6.32-358.14.1.el6  
kernel-devel.x86_64       2.6.32-220.el6    @anaconda-CentOS-201112091719.x86_64/6.2  
kernel-devel.x86_64       2.6.32-358.14.1.el6  
kernel-firmware.noarch   2.6.32-358.14.1.el6  
kernel-headers.x86_64     2.6.32-358.14.1.el6  
libreport-plugin-kerneloops.x86_64  
[takeishi@blood] %
```

図 17 kernel headers と development packages の確認。

```
$ yum list installed | grep kernel
```

インストールされていない場合は、以下のようにインストールして下さい。

```
$ yum install kernel-devel-$(uname -r) kernel-headers-$(uname -r)
```

RedHat 系の場合、DKMS/libvdpau のような他の外部パッケージに依存するため、EPEL 等のサードパーティのリポジトリのセットが必要となります。以下のコマンドにより、EPEL のためのセットアップを行います。yum を用いてリポジトリのセットを行います。既に、リポジトリがセットされている場合は、実施する必要はありません。(* run ファイルを実行 “sh cuda_8.0...” しても、“The driver installation is unable to locate the kernel source.” というエラーがでて、cuda がインストールされない時がある。その場合は、以下の赤字を実行してから、再度インストールをコマンドを叩く。)

その後，“./deviceQuery”を叩いても GPU が認識されない場合があるため、すかさずドライバーの run ファイル “./NVIDIA-Linux-...” を実行し、再度 “./deviceQuery” を叩く。）

```
$ yum install epel-release  
$ yum install --enablerepo=epel dkms  
$ yum search libvdpau  
(libvdpau が実装されているかを確認し、なければ install します↓)  
$ yum install libvpau  
$ yum update
```

dkms や libvdpau が yum でインストールできるようになったかは、以下のようにして確認します。

```
$ yum info dkms  
$ yum info libvdpau
```

CUDA のサンプルプログラムのコンパイルには、glut ライブラリが必要となります。これは、あらかじめ、以下のように yum でインストールしておきます。

```
$ yum install freeglut  
$ yum install freeglut-devel
```

もし、新規ではなく、以前の CUDA から最新の CUDA にバージョンアップするときは、

```
$ yum remove cuda-* ← yum で cuda パッケージを全て消去する  
$ rm /etc/yum.repos.d/cudas-7-0-local.repo ← 過去の cuda7.0 のリポジトリを消去  
$ yum clean expire-cudas
```

インストール準備はこれで完了したので、CUDA Toolkit を NVIDIA のサイトからダウンロードします (Fig.22)。今回は “rpm(local)” である “cuda-repo-rhel6-8-0-local-ga2-8.0.61-1.x86_64.rpm” をダウンロードします (Fig.23)。

```
$ yum install kernel-devel-$(uname -r) kernel-headers-$(uname -r)
```

CUDA ドライバーは基本的には X display のドライバでもあるので、これをインストールするときは、X が動作していない状態で行う必要があります。Linux システムの起動時は、デフォルトで run level が 5 になっているので、これを一旦、run level 3 に戻してから CUDA のドライバのインストー

付録

ルを行います。

```
$ init 3
```

ダウンロードした rpm を “/tmp” に配置します。以降の作業は全て “/tmp” 配下で行います。

```
$ rpm -install cuda-repo-rhel6-8-0-local-ga2-8.0.61-1.x86_64.rpm  
$ yum clean expire-cache  
$ yum install cuda
```

これで、インストールは完了です。もし、rpm(local) からインストールできない場合は、runfile(local) を CUDA のサイトからダウンロードし（多少時間がかかりますが）、以下の操作を行って下さい。

```
$ init 3  
(run ファイルに実行権限がない場合は以下↓)  
$ chmod 744 cuda_8.0.61_375.26_linux.run  
$ sh cuda_8.0.61_375.26_linux.run
```

最後に、使用可能なデバイスを以下のように確認します。

```
$ cd /usr/local/cuda-8.0/samples/1_Utils/deviceQuery  
$ make  
$ ./deviceQuery
```

デスクトップに 2 つのデバイスを挿した状態で “./deviceQuery” を打った時の出力画面を Fig.18 に示します。使用可能なデバイスが、“Device 0” と “Device 1” として、その詳細なスペックを確認することができます (Fig.18)。次からデバイスの仕様を見たいときは make をする必要はなく、 “./deviceQuery” だけを打てば OK です。GPU を増設する度に “./deviceQuery” を行って下さい。使用するデバイスのスイッチは、サンプル・プログラム中ではヘッダーファイル “def.h” 内で行っています。 “#define DEV” の値を “0” にするか “1” にするかで、使用するマシンを決定しています。

インストール作業が全て終了したならば、run level を 5 に戻し、reboot しましょう。

```
$ init 5  
$ reboot
```

reboot 後は、root アカウントで “/etc/ld.so.conf” に以下の行を追記し、“ldconfig” をたたきます。

```

Device 0: "GeForce GTX TITAN X"
  CUDA Driver Version / Runtime Version      8.0 / 8.0
  CUDA Capability Major/Minor version number: 5.2
  Total amount of global memory:             12207 MBytes (12800163840 bytes)
  (24) Multiprocessors, (128) CUDA Cores/MP:
    GPU Max Clock rate:                    1076 MHz (1.08 GHz)
    Memory Clock rate:                   3500 MHz
    Memory Bus Width:                     384-bit
    L2 Cache Size:                       3145728 bytes
  Maximum Texture Dimension Size (x,y,z):   1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers: 1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers: 2D=(16384, 16384), 2048 layers
  Total amount of constant memory:          65536 bytes
  Total amount of shared memory per block:  49152 bytes
  Total number of registers available per block: 65536
  Warp size:                            32
  Maximum number of threads per multiprocessor: 2048
  Maximum number of threads per block:     1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size   (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                  2147483647 bytes
  Texture alignment:                    512 bytes
  Concurrent copy and kernel execution: Yes with 2 copy engine(s)
  Run time limit on kernels:           No
  Integrated GPU sharing Host Memory:  No
  Support host page-locked memory mapping: Yes
  Alignment requirement for Surfaces:  Yes
  Device has ECC support:            Disabled
  Device supports Unified Addressing (UVA): Yes
  Device PCI Domain ID / Bus ID / location ID: 0 / 2 / 0
  Compute Mode:
    < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

Device 1: "GeForce GTX TITAN"
  CUDA Driver Version / Runtime Version      8.0 / 8.0
  CUDA Capability Major/Minor version number: 3.5
  Total amount of global memory:             6082 MBytes (6377373696 bytes)
  (14) Multiprocessors, (192) CUDA Cores/MP:
    GPU Max Clock rate:                    876 MHz (0.88 GHz)
    Memory Clock rate:                   3004 MHz
    Memory Bus Width:                     384-bit
    L2 Cache Size:                       1572864 bytes
  Maximum Texture Dimension Size (x,y,z):   1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers: 1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers: 2D=(16384, 16384), 2048 layers
  Total amount of constant memory:          65536 bytes
  Total amount of shared memory per block:  49152 bytes
  Total number of registers available per block: 65536
  Warp size:                            32
  Maximum number of threads per multiprocessor: 2048
  Maximum number of threads per block:     1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size   (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                  2147483647 bytes
  Texture alignment:                    512 bytes
  Concurrent copy and kernel execution: Yes with 1 copy engine(s)
  Run time limit on kernels:           Yes
  Integrated GPU sharing Host Memory:  No
  Support host page-locked memory mapping: Yes
  Alignment requirement for Surfaces:  Yes
  Device has ECC support:            Disabled
  Device supports Unified Addressing (UVA): Yes
  Device PCI Domain ID / Bus ID / location ID: 0 / 1 / 0
  Compute Mode:
    < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
> Peer access from GeForce GTX TITAN X (GPU0) -> GeForce GTX TITAN (GPU1) : No
> Peer access from GeForce GTX TITAN (GPU1) -> GeForce GTX TITAN X (GPU0) : No

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 8.0, CUDA Runtime Version = 8.0, NumDevs = 2, Device0 = GeForce GTX TITAN X, Device1 = GeForce GTX TITAN
Result = PASS
[root@Blood deviceQuery]#

```

Device 0

Device 1

Device 0

Device 1

図 18 deviceQuery による挿入デバイス (GPU) のスペックの確認。

```

$ vi /etc/ld.so.conf
/usr/local/cuda8.0/lib64
/usr/local/cuda8.0/lib
$ ldconfig

```

CUDA をインストールするとサンプルプログラムもインストールされます。 “/usr/local/cuda-8.0/samples/” 内には色々なサンプル・コードが入っています (データ並列アルゴリズム, 線形代数, 物理シミュレーションなど)。是非参考にしてみて下さい。サンプルプログラムのコンパイルには、環境変数の設定が必要となるので、root アカウントで “./bashrc” 内に以下の文を追記します。

```
$ vi .bashrc  
  
export PATH=/usr/local/cuda/bin:$PATH  
  
export LD_LIBRARY_PATH=/usr/local/cuda/lib64:/lib:$LD_LIBRARY_PATH  
  
$ source .bashrc ← .bashrc の反映
```

■ 備忘録

再起動させると、モニターに何も写らない状況になる時があります。これは、カーネルが更新されたためで、新しくなったカーネルに対してもう一度ドライバーをインストールすることで解決できます。下記のサイトから、自身が使用する GPU の規格にあった NVIDIA Driver をダウンロードします。

<http://www.nvidia.co.jp/Download/index.aspx?lang=jp>

ダウンロードしたドライバー（ここでは、“NVIDIA-Linux-x86_64-375.66.run”）をホーム上にコピーし、実行します。

```
$ chmod 744 NVIDIA-Linux-x86_64-375.66.run ← 実行権限を付加  
  
$ ./NVIDIA-Linux-x86_64-375.66.run'
```

今後、カーネルの自動更新を避けるために、root で “/etc/yum.conf” に次の二文を追記すると良いでしょう。

```
$ vi /etc/yum.conf  
  
exclude = kernel*
```

最後に reboot します。reboot 後に画面が表示されるまでには少し時間がかかります。

```
$ reboot
```

B. GPU プログラミングの応用例

B.1. レーリー・テーラー不安定性の成長

圧縮性流体計算の例として、レーリー・テーラー不安定性の成長を取り上げます。油のように比重の軽い流体の上に水のような重い流体が置かれる状態は不安定であり、界面の乱れがレーリー・テーラー不安定性として急速に発達することが知られています (Fig.19a)。レーザー核融合や超新星爆発にも現れる現象で、特に高周波数の擾乱の成長率は大きく、線形成長を過ぎて free fall と呼ばれる段階まで計算するには多数の格子点が必要であり、長時間の計算が必要とされています [2, 5].

$$\frac{\partial \mathbf{Q}}{\partial t} + \frac{\partial \mathbf{E}}{\partial x} + \frac{\partial \mathbf{F}}{\partial y} = 0, \quad (13)$$

$$\text{where, } \mathbf{Q} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ e \end{bmatrix}, \quad \mathbf{E} = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho u v \\ e u + p u \end{bmatrix}, \quad \mathbf{F} = \begin{bmatrix} \rho v \\ \rho u v \\ \rho v^2 + p \\ e v + p v \end{bmatrix}. \quad (14)$$

B.2. AL-Si 合金の樹枝状凝固成長

金属材料の機械的強度や特性はミクロな組織的構造に基づくため、より高性能な材料を得るためにミクロなダイナミクスの解明が必要です。近年、材料の相転移や相分離などの解明に非平衡統計物理学から導出されるフェーズフィールド・モデルが注目されています。導出される方程式は、時間空間の偏微分方程式であり、空間の離散化によりメモリへのステンシル計算となります。秩序変数 ϕ (フェーズフィールド変数) を導入し、高層部分に $\phi = 1$ を、液相部分に $\phi = 0$ を設定します。界面を含む領域では ϕ が 0 から 1 へと急峻かつ滑らかに変化する拡散界面として扱われ、 $\phi = 0.5$ が界面として扱われます。二元合金のデンドライト凝固成長では、フェーズフィールド・モデルから導出される界面エネルギーの異方性を考慮した Allen-Cahn 方程式と溶質濃度についての時間発展方程式(拡散方程式)を解きます (Fig.19b) [6].

$$\frac{\partial \phi}{\partial t} = M_\phi \left[\nabla \cdot (a^2 \nabla \phi) + \frac{\partial}{\partial x} \left(a \frac{\partial a}{\partial \phi_x} |\nabla \phi|^2 \right) + \frac{\partial}{\partial y} \left(a \frac{\partial a}{\partial \phi_y} |\nabla \phi|^2 \right) + \frac{\partial}{\partial z} \left(a \frac{\partial a}{\partial \phi_z} |\nabla \phi|^2 \right) - \Delta S \Delta T \frac{dp(\phi)}{d\phi} - W \frac{dq(\phi)}{d\phi} \right], \quad (15)$$

$$\frac{\partial c}{\partial t} = \nabla \cdot [D_S \phi \nabla c_S + D_L (1 - \phi) \nabla C_L]. \quad (16)$$

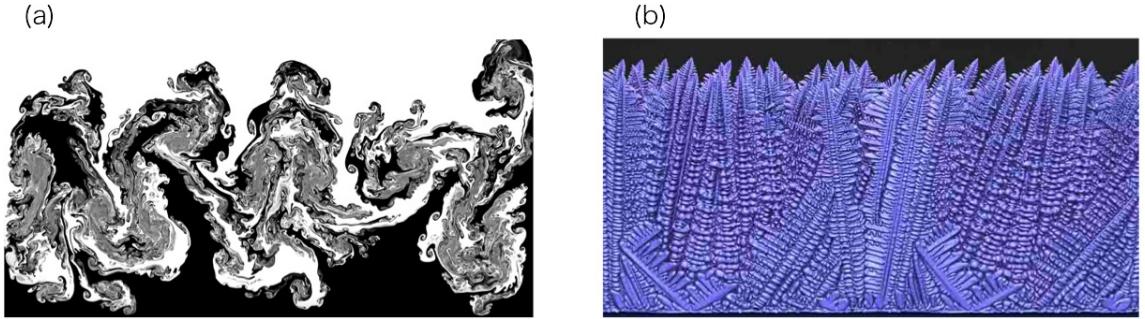


図 19 (a) レーリー・テラー不安定性の成長 (YouTube title; “2D High-resolution Rayleigh-Taylor Instability in UHD (4K, 2160p)”. (b) TSUBAME2.0 の 512GPU を利用し, $4096 \times 1024 \times 4096$ 格子で計算した Al-Si 合金の樹枝状凝固過程 (TSUBAME ESJ, Vol.5, 2012).

参考文献

- [1] John Cheng, Max Grossman, and Ty McKercher. CUDA C プロフェッショナル プログラミング. インプレス, 2015.
- [2] Y. Imai, T. Aoki, and K. Takizawa. Conservative form of interpolated differential operator scheme for compressible and incompressible fluid dynamics. J Comput Phys., Vol. 227, pp. 2263–2285, 2008.
- [3] D. Matsunaga, Y. Imai, T. Omori, T. Ishikawa, and T. Yamaguchi. A full gpu implementation of a numerical method for simulating capsule suspensions. J Biomech Sci Engng., Vol. 14, p. 00039, 2014.
- [4] N. Takeishi, Y. Imai, T. Yamaguchi, and T. Ishikawa. Flow of a circulating tumor cell and red blood cells in microvessels. Phys Rev E., Vol. 92, p. 063011, 2015.
- [5] M. S. Davies Wykes and S. B. Dalziel. Efficient mixing in stratified flows: experimental study of a Rayleigh-Taylor unstable interface within an otherwise stable stratification. J Fluid Mech., Vol. 756, pp. 1027–1057, 2014.
- [6] A. Yamaoka, T. Aoki, S. Ogawa, and T. Takaki. Gpu-accelerated phase-field simulation of dendritic solidification in a binary alloy. J Crystal Growth., Vol. 318, pp. 40–45, 2011.
- [7] 伊藤智義. GPU プログラミング入門 - CUDA5 による実装. 2013.
- [8] 青木尊之, 額田彰. はじめての CUDA プログラミング. 工学社, 2009.