GPU による
流体シミュレーションの高速化

千葉大学理学部物理学科 4 年 山本瑶祐

2010 年 8 月 28 日

1　始めに

GPU は本来グラフィクス処理専用のハードウェアである。

著しい進歩を遂げてきた CPU であるが、元々あまり画像処理に向かないアーキテクチャである上、基礎的な画像処理すら賄えない場合が多かった。その上、例え可能であっても、CPU が、頻繁に行われる画像処理にのみかかりっきりになるのは好ましいことではない。このため、専用ハードウェアを設け、画像処理を CPU から分離する流れが出てくるのは自然のことであった。こうして GPU は誕生した。

GPU は高い処理能力を必要とする画像処理へ対応するため、あまり高い性能を持たない小さな演算装置を大量に並べることで解決した。これはポラックの法則、すなわち、プロセッサの能力は、そのダイサイズの平方根に比例する、という経験則に従うと極めて合理的である。すなわち、プロセッサの能力が平方根でしか伸びないのであれば、プロセッサの数を増やせば良いのである。こうすれば、並列性さえ良ければ、プロセッサの数を N 倍することで、性能も N 倍になり、ポラックの法則を打ち破ることができる。

対して、CPU は過去の並列化を考慮しないソフトウェアや、そもそも現在のソフトウェア技術を持ってしても並列化できないソフトウェア (そしてそれらのソフトウェアは現在でも多数派である) を高速に動作させるために、プロセッサの数を増やすことはできず、ひたすらに 1 プロセッサあたり性能が求められた。

こうして GPU と CPU の性能差は広がっていったが、一方で GPU 側は、当初 2D のみの対応から、3D への対応や、画像処理の多様化に伴い、CPU ほどでないにせよ、汎用的な処理が要求され、またその要求を満たし続けてきた。

斯様な情勢であって、GPU の演算能力を画像処理以外に用いよう、と考える人間が出てくるのは寧ろ当然と言える。こうして登場したのが、GPU の演算能力を画像処理以外に用いる GPGPU である。

また、GPGPU が盛んになってきたために、常に貪欲に計算リソースを渇望し続けてきた、流体シミュレーションで用いよう、と考える者が出てくるのも、また必然である。

2　研究目的

研究目標は、後述するように、nVidia 製 GPU と CUDA を用いて、流体シミュレーションは高速化するのかを調べる。また、どのような流体シミュレーションが高速になるのかも調べる。

3　GPU について

3.1　特徴

GPU は以下の特徴を持つ

- 小型の Core を大量に搭載
- 高い単精度小数点演算性能
- 非常に低いデコード能力

- 要求される高い並列度
- 比較的高いバンド幅
- 低用量だが高速な Register & Shared Memory
- 明示的なメモリ管理 (キャッシュ無し)
- CPU 用の既存言語は使えない

具体的に CPU(Intel Core i7 965) と GPU(Tesla C1060) を性能で対比すると以下のようになる。

	CPU	GPU
論理演算性能	51.20Gflops[*1]	933Gflops
Core 数	4	240
バンド幅	38.4GB/s[*2]	102GB/s
キャッシュ	あり (L2,L3)	なし

3.2 言語

GPU を用いるにあたって、使える言語としては大きく CUDA と OpenCL が挙げられる。この 2 つは、基本的には同じハードウェアを対象としているので、似ている部分もあるが、異なる部分も多い。

CUDA　nVidia 社が開発した言語で、C++ をベースに拡張をしたものになる。開発元が開発元故に、nVidia 製 GPU でしか使えない。

OpenCL　多くの企業 (CPU 企業含む) によって開発された言語で、こちらも C がベースとなっている。GPGPU に対応した GPU であれば全て対応するのみならず、CPU で並列処理を行うことも可能である。ただし、現実問題として、性能を出し切るためには、特定のハードウェアに特化したプログラムである必要があると言われている。

今回は、開発環境が出揃っており、技術的にも成熟している CUDA を用いてプログラミングを行った。以降は特に明示しない限り、CUDA と nVidia 製 GPU に限った話である。

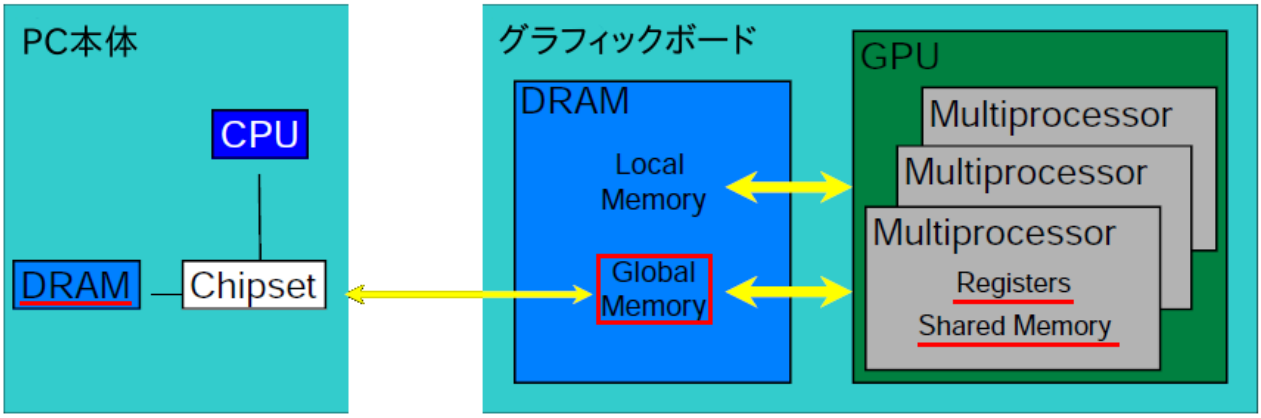
3.3 メモリ構造

計算機には、演算部だけでなく、演算部へデータを渡す装置が必要である。論理的には、足し算であれば、元のデータが 2 つに結果が 1、足し算命令 1 つの、合計 4 つが必要となるのが自明である。

確かに現代の演算装置の性能は非常に早く進化しているが、その演算装置にデータを流すメモリの性能は、必ずしもそれに追い付いてきたとは言えない。それどころか、その差は開くばかりである。

そのため、現代の計算機は、メモリ構造を階層化することで、その問題に応えている。演算部に近いほどメモリは高速だが低容量に、遠いほど低速だが大容量になる。GPU も例外では無いどころか、論理演算性能が高いだけに、むしろ CPU より逼迫しているとすら言える。

その結果として、GPU のメモリは次のような階層構造を形作っている。



最も GPU に近い Register&Shared Memory から、そこから離れた Global Memory、Local Memory が、グラフィックボードと呼ばれる拡張ボード内には存在する。その外側には、CPU が使うホストメモリが存在する。GPU における演算は、Register や Shared Memory にデータがヒットする内は、ほぼ No Wait で計算することが可能である。つまり、論理演算性能に近い演算速度を出すためには、できる限り Register や Shared Memory に計算データを置く必要がある。ただし、その容量は Register で 16384 本 (64KB)、Shared Memory では 16KB と大きくない。

一方、GPU のチップの外に配置される Global Memory は、CPU が搭載するメインメモリよりは性能が高いものの、その速度は Register や Shared Memory には大きく及ばず、大体 Register や Shared Memory の 100 分の 1 ぐらいの性能しか出ない。つまり、常に Global Memory にデータを置いて計算した場合、他の条件が良くても、最低で 100 分の 1 まで性能が落ちることを意味する。

3.4 Processor 構造

GPU で実際に計算をするのは、単純な計算を行う Streaming Processor(SP)、倍精度演算を行う DP、複雑な演算を行う SFU に分かれる。SP が 8 と DP が 1、SFU が 2 と Shared Memory を併せて Streaming Multiprocessor(SM) を形成する。

SP が積和算 (2FLOPs)、SFU が 4 つの積算を行うことで 1SM あたり 1Cycle で 24FLOPs、1 つの GPU でこれが複数個 (Tesla C1060 の場合 30) あり、これが特定周波数 (Tesla で 1.3GHz) で駆動する。結果、Tesla C1060 で 936GFLOPs が達成される。

倍精度の場合は DP が積和算を行った場合で、1Cycle で 2FLOPs となり、論理演算性能は単精度の 12 分の 1 となる。

なお、基本的に、全ての SP は同じ命令を実行する。2 つの SFU も SP とは違っていても良いが、SFU 同士は同じ命令を実行する。これは CPU に比べて命令をデコードする能力が非常に低いからである。

というよりも、同じ計算を並列で動かすことで性能を達成する GPU においては、消費電力が大きくてトランジスタ消費の多いデコーダーは荷物でしかないので、最小限しか搭載されていない、という方が正しい。

ちなみに、SP も SFU も 4cycle の間、同じ命令を実行するので、SP に於いては実質 32、同じ計算を行うことになる。この 32 を 1Warp と呼ぶ。

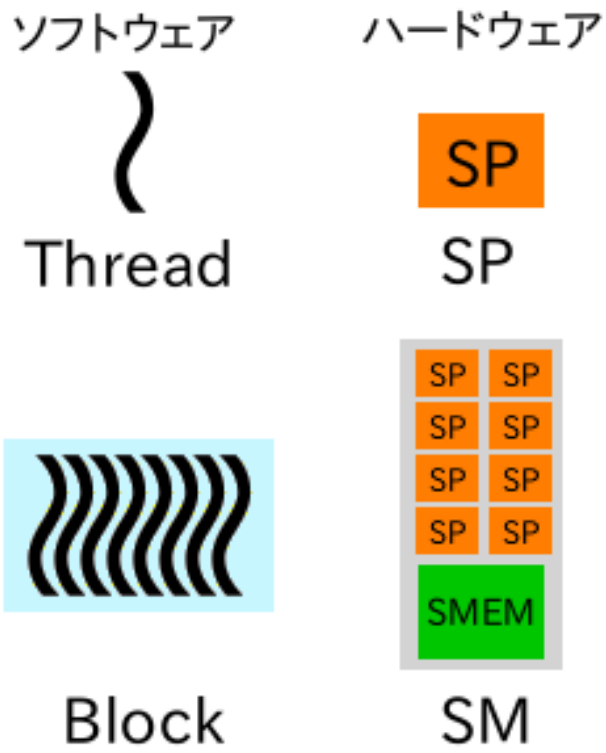
3.5　プログラム構造

CUDA では、命令発行を Thread、Block、Grid と呼ばれる単位で行う。この構造は実際の Processor 構造と密接に絡みあっている。

最も最小の単位は Thread だが、これは 1SP に相当する。同じ Thread の計算は同じ SP で行われるので、Register によって変数が共有される。

この Thread を複数束ねることで Block を形成する。1Block は必ず 1SM で処理される。また、Block の内部では Shared Memory が共有される。

Block を複数束ねると Grid となり、一般に GPU への命令は 1Grid で行われる。Grid を例えると C 言語における 1 関数に近い。複数の Grid を GPU へ投げることで、目的の計算を達成することもある。



4　CUDA 高速化

ここでは CUDA の高速化について扱う。厳密には、計算機はピーク性能が示されており、それ以上は絶対に伸びない筈で、つまるところは低速化させない為の技術である。

特に GPU は、CPU と違い、どのような場合でもある程度の性能が出るように注意深く設計された演算装置では無いので、様々なことに留意する必要がある。

4.1　分岐

GPU は本来分岐命令には強くない。というか、分岐という概念はほとんど無い。1Warp が同じ動作をするのだから、ある意味当然である。唯一、SP は、わざと命令を実行しないことによってのみ、分岐を達成できる。

つまり、1Warp の 32Thread が True と False に分岐したら、先ず True の処理が実行され、その際 False の Thread は命令を実行しない。次に False の処理が実行され、True の Thread は命令を実行しない。結果として、分岐によって、最低でも倍の実行時間がかかることになる。

もしこの分岐が n 階にネストしていた場合、GPU は最低でも 2 の n 乗に比例して遅くなるのが容易に推測されるであろう。これを divergent branch と呼ぶ。

4.2　Global Memory

Global Memory へのアクセスは 32byte、64byte、128byte 単位で、しかもそれぞれ 32、64、128 の倍数の Address を先頭にしてアクセスする必要がある。

さて、この場合、単精度小数点は 4byte であるので、Thread が Global Memory へ個々にアクセスした場合、盛大な無駄が生じる。よって GPU は Thread の Global Memory へのアクセス要求を 16Thread 毎に纏める。これを Coalescing と呼ぶ。基本的に、Coalescing しない限り、グラボの高速なメモリの恩恵は受けられない。

Coalescing が起こる条件は、16Thread が連続した Global Memory の Address 領域にアクセスすることである。古いグラボでは、更に先頭の Thread が 32、64、128 の倍数の Address にアクセスする必要がある。

4.3 Shared Memory

Shared Memory へのアクセスは 16Thread 単位で行われる。そして Shared Memory は 16Bank に分かれている。この Bank は Shared Memory を 4byte 毎に管理している。つまり、Shared Memory の Bank0 は、Shared Memory Address の 0、64、128…を管理していることになる。ここで重要なのは、Shared Memory が Bank1 つ当たり 1 アクセスしか処理できないことである。

つまり、Thread が 4byte のデータを 2 つおきアクセスしようとした場合、偶数番目の Bank には 2 つのアクセス要求が来るが、奇数番目の Bank にはアクセス要求が無い状況になる。(又はその逆) この状況になると、Shared Memory へのアクセスは普段の倍かかることになる。これを Shared Memory bank conflict と呼ぶ。

Global Memory の場合と異なり、とにかく 16Thread が異なる Bank にアクセスすれば良いので、連続である必要は無い。

Shared Memory bank conflict とならない唯一の例外は、全ての Thread が同じデータにアクセスしようとした場合のみである。

4.4 Register

Register にも Bank conflict が存在する。nVidia の資料によると、Thread が 64 の倍数であれば、コンパイラが自動で Register Memory bank conflict を回避することになっている。

また、Register が不足した場合、Local Memory へデータが行くことになっている。Local Memory の速度は Global Memory とほぼ同じで、相当に遅い。できる限り Register が溢れない程度に有効活用する必要がある。

なお、初期設定では、レジスタは 1Thread 当たり 60 本に制限されている。これよりも多くのレジスタを用いる場合は、-maxrregcount オプションで指定する必要がある。

4.5 同時実行

1SM は常に 1 つの Block のみを実行しているわけでは無い。Global Memory へのアクセスが生じた場合など、処理が行われなくなった場合は、別の Block の処理を実行する。

このように、同時に複数の Block を実行することによって、あたかも Global Memory へのアクセスが隠蔽されたかのように見える。

1SM が実行できる最大の Block は複数の要因が絡みあって決まる。まず、1 つの SM で実行できる Block 数は最大 8 である。次に、1 つの SM で実行できる Thread 数は最大 1024 である。また、実行している Block の数だけ Register、Shared Memory がそれぞれ必要になる。

以上を整理すると、

- 8
- 1024 / Thread per Blcok
- 16384byte / Shared Memory per Block
- 16384 / (Register per Thread × Thread per Block)

のうち、最も小さい値が同時実行可能な最大 Block 数ということになる。

4.5.1 Occupancy

上記で見てきたように、同時実行は他の要因が無ければ、最大で 1SM 当たり 1024Thread まで実行される。1SM で同時に 1024 並列行った場合の Occupancy を 1.0 とし、効率的に実行できているか否かの数値を表すのが Occupancy である。

なお、2 次元の場合、Register の制限で 4 並列までしか実行できず、1Block 当たり 64Thread が実行されるので、

1SM で同時に 256Thread が実行される。従って Occupancy は 0.25 となる。

4.6 Thread

1 つの SM で処理できる最大の Thread は 768or1024 であるので、先ず Thread は 768 の約数であることが望ましい。次に、Register Memory bank conflict を考慮すると、64 の倍数であることが望ましい。最後に、1SM の上限 1024Thread に達するためには、1SM の最大実行可能 Block 数の 8 から考えて、128 以上であることが望ましい。

以上から考えると、128、192、256 がベスト、ということになる。nVidia の Reference Manual に依ると、192、256 が良い、ということになっている。数が多い方が良いということであろうか。

実際には、Thread 当たり計算量が大きい場合や、Shared Memory、Register が多く消費される場合等では、1SM 当たり 1024Thread は多過ぎるため、Block 当たり 64Thread でも充分な性能が出る場合が多い。

4.7 Block

1 つの SM 当たりで実行する Block 数は、最高 8 まで増える。また、現在最も高性能な GPU では、1 枚 30SM が存在する。従って、240Block 程度投げると、理論上において GPU を使い切ることができる。

実際には、複数 GPU を搭載したものもあり、Reference Manual には 1000Block 程度で性能がスケールする、と書かれている。

もし複数枚の GPU を用いて、更に高速化を図るのであれば、もっと Block 数が必要となる。

4.8 まとめ

以上を踏まえると、数万 Thread 程度を生成しないと、GPU は速度を出せないことが分かる。

また、容量の限られた Shared Memory と Register をうまく利用しないと、やはり速度は出ない。ある程度、データアクセスの局所性が必要、ということになる。

幸い、流体シミュレーションはある程度上記の特性を満たしており、比較的 GPU によって速度が出やすいと思われる。

5 GPU による流体計算

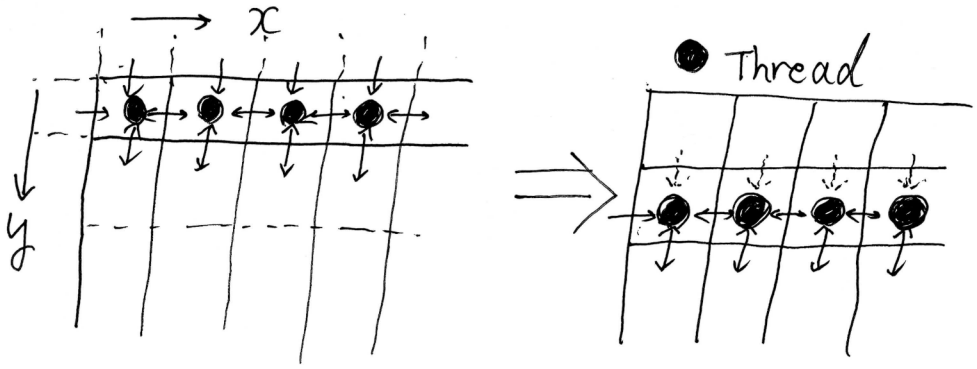
以上を踏まえて、GPU による流体シミュレーションのコードを作ることにする。

5.1 拡散方程式

流体シミュレーションにおいては、1 次精度である限り、ほとんどのスキームで、ある格子点の $\boldsymbol{t} + \boldsymbol{\Delta t}$ の物理量を求めるには、隣接する格子点 (2 次元では 4 つ) の全ての物理量が必要である。

これは拡散方程式も同様で、拡散方程式によって速度の出たアルゴリズムは、そのまま実際の流体シミュレーションに適用できることになる。

拡散方程式において速度を出すために、以下のようなアルゴリズムを考えた。



まず、計算領域全体を、いくらかの Block に分ける。今回は計算領域を仮に 1024×1024 とし、Block サイズを 128×8 とする。この場合、Block 数は 1024 となり、先程の条件を満たすことになる。

この Block 全てに Thread を割り当てる訳ではない。この Block の 1 列にのみ Thread を割り当て、この Thread を y 方向に動かすことで、Block 全体の計算を行う。つまり、上記の例では、Thread 数は 128 であり、Thread1 つが 8 つの格子点の計算を行うことになる。

具体的には、まず Thread 自身が存在する格子点と、その上下の格子点のデータを Global Memory から読み出し、Register に格納する。次に Thread 自身が存在する格子点のデータを Shared Memory に保存する。そして上下の移流を Register から読み出して計算、左右の移流を Shared Memory から読み出して計算し、計算結果を Global Memory に書き出す。

次の格子点を計算する際には、更に 1 つ下のデータを読み出して Register に置き、Swap する。Shared Memory も全て 1 つ下のデータへ置き換える。これを繰り返す。

このように Thread をすべての格子点で割り当てないアルゴリズムを利用するのは Register と Shared Memory の問題に起因する。もし、Block 全てに Thread を割り当てるようなアルゴリズムの場合、全てのデータを Shared Memory に置かなくてはならない。Register が使われず、非効率である。また、拡散方程式では問題にならないが、実際の流体では Shared Memory が不足する恐れがある。

一方で、Thread1 つが 8 つの格子点を計算するこの方式だと、Shared Memory は Thread が並んだ横方向の物理量のみを保存すれば良く、また、上下の物理量は Register のみが保存すれば良く、Shared Memory が節約できる。

実際にこの方式を用いて拡散方程式を解くと、CPU の約 50 倍の性能が出た。ただし、CPU 側のチューニングがされていないので、実際にはかなりその差は縮まるであろう。

5.2 ロー法

次に、このアルゴリズムを実際に流体シミュレーションへ適用してみる。次元は 2 次元のものと 3 次元のものを作った。

そのスキームとして、今回はロー法を用いたが、ロー法で行った特別な理由は無い。たまたま手元の教科書にロー法の解説が詳しく掲載されており、手っ取り早いかと思ったからである。(実際には、ロー法はかなり難しい部類に入ると思われる)

ロー法の特徴としては、計算量は比較的多いものの、安定したスキームで、数値振動が起こりにくい、という特徴がある。

なお、$\boldsymbol{\Delta t}$ の値は、CFL 条件を満たすような最大の値を取るように、変化させるのが一般的であるが、今回は簡略化のために省いた。また、CIP 法など他にも流体シミュレーションを扱う上で便利な手法があるが、全て省いた。

格子点に関しても、等間隔を前提にプログラムをした。ただ、dx 可変でも速度をあまり損なうことなく処理することが可能であると予想される。

拡散方程式と比べると、Flow の計算量が著しく増加しているため、拡散方程式では隣接格子点両方で計算していた Flow であるが、流体シミュレーションにおいては、一度計算したものを、x 方向は Shared Memory へ、y 方向は Register に保存し、二度手間を省いた。

3 次元のものは、Thread を x、y 方向に並べ、z 方向へ Thread を移す、という手法を用いた。よって、基本的には 2 次元の場合と大差が無いが、2 次元と遜色無い速度を出すために、かなり工夫が必要であった。後程述べる。

5.3 Lax-Wendroff

別のスキームとして、Lax-Wendroff のものを作成した。Lax-Wendroff は、非常に少ない計算量で、2 次精度を達成できるスキームであるのが特徴である。また、状態方程式をそのまま代入する形で表すことができるので、Roe 法に比べ拡張が容易である。

ただし、純粋な Lax-Wendroff では衝撃波から数値振動が発生し、まともに計算するのは不可能である。従って、人工粘性の項を導入し、数値振動を抑えることとする。

素の Lax-Wendroff は、基本的に Half-Step と Full-Step の 2 段階に分けて行われる。

2D の場合、Half-Stpe では $(\boldsymbol{i+1/2, j+1/2})$ での Flux を求める。その際には $(\boldsymbol{i,j})$、$(\boldsymbol{i+1,j})$、$(\boldsymbol{i,j+1})$、$(\boldsymbol{i+1,j+1})$ の物理量を用いる。

次の Full-Step では、最終的に求めたい $\mathbf{d}\boldsymbol{t}$ 後の $(\boldsymbol{i,j})$ の値を求める為に、$(\boldsymbol{i-1/2, j-1/2})$、$(\boldsymbol{i+1/2, j-1/2})$、$(\boldsymbol{i-1/2, j+1/2})$、$(\boldsymbol{i+1/2, j+1/2})$ の Half-Step で得られた Flux を用いる。

以上のように、1 次元ロー法とは異なり、1 点の $\mathbf{d}\boldsymbol{t}$ 後の物理量を求める際には、5 点の物理量だけではなく、9 点の物理量が必要であり、必要なメモリ量やその帯域はかなり増加する。

もし全く工夫無しにロー法を拡張したのでは、特に Shared Memory が著しく肥大化してしまう。従って、Shared Memory は全て一時メモリとして使い回し、Register を有効活用することで、この問題をある程度解決することに成功した。

また、それとは別に人工粘性を導出する必要がある。こちらは、拡散方程式やロー法と同じように、5 点の物理量が必要である。

これは Lax-Wendroff 部分とは異なる領域のデータが必要で、かなりの Register を消費した。こちらの節約はできていない。

5.4 CFL 条件

Lax-Wendroff スキームにおいて、初めて CFL 条件による $\mathbf{d}\boldsymbol{t}$ の算出を導入した。この算出にも GPU を用いており、実行時間のうちの 10% を占める。

この処理は各々の格子点において CFL 条件を満たすような $\mathbf{d}\boldsymbol{t}$ を求め、その値が処理する全体において最小となる値を、実際の $\mathbf{d}\boldsymbol{t}$ となるような処理を行う。

この最小の $\mathbf{d}\boldsymbol{t}$ を求める処理がやっかいで、並列化を行いつつも、Shared Memory bank conflict や divergent branch を防ぐ必要がある。

実際のプログラムにおいては、1SM 当たり 256 格子点、256Thread を割り当て、各格子点の $\mathbf{d}\boldsymbol{t}$ を計算した後、128Thread が 2 格子点のうち小さい方を Shared Memory に代入する、ということを、1Thread になるまで繰り返すようにした。

5.5 CPU と GPU の比較

以上を踏まえ、実際に動かしてみて、速度を計測した結果が以下である。

なお、CPU のものは、OpenMP を用いて for 文並列化を行っている。1 並列に対し、4 並列は 3.9 倍、8 並列で 4.1 倍となり、並列化効率はそれぞれ 97%、103% となっている。簡易な並列化だが、十分な並列化効率が達成できた。

※ただし若干の問題点があり、2CPU のシステムでは、バンド幅がネックとなって処理速度が上がらなくなる。従って、2CPU では行っていない。for 並列では不足で、MPI できちんと並列化するか、MPI 並列化を行う必要がある。

実行環境としては、CPU と GPU1 に関しては以下の構成のマシンを用いた。GPU2 は以下の構成となっている。

CPU Core i7 920(42.56GFLOPs)

Memory　3GB、25.6GB/s(DDR3-1066 Triple Channel)
GPU　GeForce GTX 260(875GFLOPs)
GPU Bandwidth　118GB/s
OS　Ubuntu Desktop 10.04
CC　gcc 4.3.4 + OpenMP
NVCC　CUDA 3.0 + gcc 4.3.4

そして、計算としては衝撃波管問題を、以下のようサイズで解いた

Grid　2D 1024x1024、3D 120x126x32
Step　LW 828、Roe 1024

以上を踏まえ、CPU に比べて GPU がどの程度高速かを調べる。まずはロー法で、

	CPU	GPU
2D 時間	64(s)	1.44(s)
3D 時間	76(s)	1.98(s)
実効性能 2D	5.59GFLOPs	248GFLOPs
実効性能 3D	4.21GFLOPs	181GFLOPs
実効効率 2D	13.1%	28.3%
実効効率 3D	9.9%	20.6%

次は Lax-Wendroff で、

	CPU	GPU
時間	50.7(s)	2.24(s)
実効性能	4.43GFLOPs	100GFLOPs
実効効率	10.4%	11.4%

CPU があまりチューニングされてないとはいえ、相当に高速である。GPU が流体シミュレーションで有用だ、と言うには充分であろう。

また、特筆すべき点として、全般的に GPU の方が実効効率が良くなっている。無論、CPU のチューニング不足という問題はあるであろうが、一方でキャッシュの問題が考えられる。

CPU は GPU の Shared Memory のような超高速だが、プログラマが明示的に保存の指示を行うメモリを持たない。代わりに、自動でデータを保持するキャッシュを持つ。

キャッシュ容量は Shared Memory よりは多いものの、その性能は低く、そもそも流体でキャッシュのヒット率は低いと言われており、元々 CPU にとって不利である。その結果がこの実効効率に繋がった可能性がある。

なお、1 格子点 1Step 当たり計算量は、Roe 法 2D が 357、3D が 661、Lax-Wendroff が 278 である。Lax-Wendroff は 2 次精度である一方、Roe 法は 1 次精度であるので、それも考慮すると、Lax-Wendroff の計算量はかなり少ない。

5.6　メモリ

この流体シミュレーションを行う際に必要となる物理量であるが、格子点毎に密度、速度 (x,y,z)、圧力の、2 次元で 4 つ、3 次元では 5 つが必要となる。

この場合、実際に割り振られるメモリは次のようになっている。(全て 64Thread の場合)

	実容量	LW	Roe2D	Roe3D
Register	65536Byte	18688Byte	12288Byte	20480Byte
Shared	16384Byte	3692Byte	2416Byte	5672Byte
同時実行数	8	3	4	2

まずは Shared Memory について考える。2 次元の場合、Thread 数が 64（この値が常に早いことは確かめられている）であると、両端の部分を含めると、単精度小数点の容量は 4byte より、物理量を保存するのに、Shared Memory が 1056byte 必要である。

実際には、ロー法において、計算途中の値である flow も保存するため、その倍必要になる。この値は 16384byte に比べ充分に小さい。

Lax-Wendroff では、物理量の他に 2 つの Flow を一時的に保存する領域が必要で、3 倍が必要となる。

3 次元のロー法の場合は、x-Thread が 16、y-Thread が 4 とする (常にこの場合最速) とすると、必要な Shared Memory は 2160byte とほぼ倍になる。Flow の量 1700byte を追加すると結構な量になる。

Register に関しては、様々な値が保存されているので、簡単には示すことができないが、上記のような値になる。Register が多い分には、Shared Memory へ逃すことも容易なので、Shared Memory ほど問題にはならない。

なお、Global Memory に関しては、現状全格子点の物理量が入力と出力に分かれて保存されている。その量は 1024×1024 の場合で約 34MB で、格子点を 10 倍にしてもかなり余裕がある。Tesla C1060 のような科学計算用の GPU の場合、Global Memory は 4GB あり、あまり問題とならないだろう。もし容量が必要なら、枚数を増やして解決する手もある。

5.7 精度

実は GPU の単精度小数点演算の計算結果は微妙に異なる。

まず、一般的な四則演算の場合、最下位 1bit の値が不定である。この程度であれば大きな問題にはなり難いとは思われる。

次に、三角関数等の超越関数等の計算を行った場合、10 進数単精度小数点の場合で最下位桁の値が不定となる程度にずれる。この点は無視できないほどには大きいが、そもそもそのような計算はロー法・Lax-Wendroff においては極めて限られるので、実質考慮する必要は無い。

6 小改良

非常に些末な改良点をここで述べる。特に必要が無ければ読み飛ばしても問題無い。

6.1 Thread と Flow の一致

拡散方程式の計算においては、Thread の数は変化を適用する格子点の数とした。そして、Thread がその回り全ての Flow を計算し、格子点における変化量を算出した。

一方、ロー法においては、Flow の計算量がかなり大きいので、Flow を 2 度計算するのは無駄が多い。従って、Flow 計算結果は Register と Shared Memory に保存し、2 度計算するのを省いた。

この場合、拡散方程式と全く同じ方法で Thread を配置した場合、if 文を用いて 1 つの Thread が 2 つの Flow を計算する必要が出てくる。

これは無駄であるので、Thread の数を Flow の数にした。こうすると、if 文が要らず、全ての Thread が 1 つの Flow を計算するようになり、都合が良い。

この場合、Thread は適用する格子点の数よりも 1 少ない値になる。これが少々やっかいで、Thread を無駄なしに配置しようと思うと、x 方向の全体の格子点数が、x 方向の Block サイズから 1 を引いたものの倍数にする必要がある。

2 次元の場合、この工夫を取り入れても大きく高速化しないため、このアルゴリズムを導入していない一方、3 次元の場合は大きく高速化が達成されため、導入した。3 次元の計測における格子点数が変な値になっているのは、これが原因である。

7 nVidiaGPU の世代

nVidiaGPU の世代の違いにおける相違点と、プログラム上の注意に関して述べる。特に必要が無ければ読み飛ばしても問題は無い。

7.1 GT8X,GT9X

最初に CUDA へと対応したのが GT8X、GT9X 世代であるが、最新の GT200 世代に比べていくらかの制約がある。

- 1SM 当たり Thread 数の上限が 768
- 1SM 当たり Register 数が 8192 本
- Coalescing の挙動の違い

Coalescing の挙動については、Thread0 の Global Memory へのアクセスが 32 や 64、128 の倍数でないと、Coalescing が全くなされず、アクセス速度が 10 分の 1 程度まで落ちる。G200 世代では 2 つに分かれて Coalescing され、速度は 1.5 分の 1 程度にしか落ちない。

7.2 GT200

一方で、今回の研究で主に用いた GT200 は以下の特徴を持つ。

- 1SM 当たり 8SP + 1DP + 2SFU
- 1SM 当たり Thread 数の上限は 1024
- 1SM 当たり Register 数は 16384 本

7.3 GF100

nVidia の最新の GPU は、Tesla C20XX に搭載される GF100 である。
GF100 は、GT200 に比べて、大きく変化している。

- 1SM 当たり 32CUDA Core + 4SFU(DP は無い)
- SP が倍精度演算に対応 (2Cycle)(Tesla のみ)
- L1 キャッシュを搭載 (Shared と共用)
- L2 キャッシュを搭載
- GDDR5 をサポート
- ECC メモリをサポート (Tesla のみ)
- アドレス空間が 64bit に

1SM での処理は、前に述べたように 32Thread 単位であるので、GT200 のように 8SP を 4Cycle 回して演算するよりも、32CUDA Core の 1Cycle で回した方が合理的である。(ただし、後述するが、Shared Memory の問題がある) なお、CUDA Core は SP の名前が変更されただけ、と考えて良い。

また、倍精度専用のハードウェア (DP) が廃止された代わりに、CUDA Core を 2Cycle 回すことで倍精度演算を行うように変更された。結果的に、倍精度演算性能は、単精度に比べて 12 分の 1 から 2 分の 1 まで上昇したことになる。

また、L1 キャッシュが搭載された。Shared Memory と併せて 64KB が確保され、16KB 又は 48KB を選べる。GT200 向けに最適化したプログラムでは L1 を多めに確保することで、性能を少し上げられるし、GF100 向けに Shared Memory を 48KB 確保してギリギリまで速度を追求することもできる。

実際問題として、Shared Memory の容量は、1CUDA Core 当たり、という観点から見ると、48KB だとしても、むしろ減っている。ただし、Core が 4 倍になったことで、16Thread を 2Cycle で処理していた GT200 に比べ、32Thread

を 1Cycle で処理する GF100 の Shared Memory は、確実に何らかの形で帯域幅が向上している筈である。

L2 キャッシュも搭載された。容量は 768KB で、速度は L1 や Shared Memory、Register よりも大分遅くなるが、Global Memory よりは余程マシ程度には早い筈である。

Global Memory も大幅に強化され、GDDR5 サポートで高速化が期待されるし、ECC メモリのサポートで、non ECC で懸念される、宇宙線によるメモリのデータ破壊が回避でき、結果、長時間計算が現実的になってきた。またアドレス空間が 64bit になることで、Tesla C2070 の 6GB メモリモデルが登場した。

なお、GT200 までは、Tesla と GeForce に値段以上の差は無かったが、GF100 は ECC や倍精度演算が Tesla のみ対応するなど、差が生じている。従って、安価な市中の GPU が使えた GT200 に比べると、コストパフォーマンスは寧ろ低下したと言えるかもしれない。その値段差は無視できないので、倍精度や ECC メモリが不要であれば、GeForce を使うという選択肢もある。

8 結論

今回の研究において、流体シミュレーションを GPU で行った場合、かなりの高速化が望めることが分かった。

スキームは、Lax-Wendroff のような 2 次精度だが単純なスキームよりも、Roe 法のように高度なスキームの方が得意である。1 次精度と 2 次精度が与える効果は、おそらく 2 次精度でバンド幅という側面で不利が予想される。

また、MHD や 2 次精度への移行、倍精度計算などによって、Shared Memory と Register の使用量が増大し、同時実行数が減少する。アルゴリズム等の工夫が必要となるであろう。ただし、GF100 によって幾分かは緩和できる。

以上から、GPU が有利な計算は、計算量は多いけど、必要となるデータの量が少ないようなスキームで、大規模な計算であればなお良い、という結論となろう。