GPU 並列計算環境の構築とスレッド階層構造

伊藤 誠

1　はじめに

大規模な数値計算を行う場合、単一のノード (単一の CPU) を用いて行うのではなく、多数のノード (複数の CPU や複数の計算機) を並列にして計算を行うのが一般的になって久しい。日本を代表する並列計算機である「京」は約 88,000 個の CPU を並列で動かし高速に計算をするための巨大な計算機システムである。「京」はポートアイランドにある理化学研究所の研究施設に設置されている。他にも大型の並列計算機システムは大学レベル、大きな研究室レベルで設置されている。その他の大型並列計算機として、GPU(Graphics Processing Unit) を用いた並列計算機システムもある。GPU 単体でも数千コアによる並列計算が可能であり、さらにそれらを複数個用いた並列計算機システムもある。東京工業大学の TSUBAME(つばめ) はこの種の代表的な計算機システムである。

本稿では GPU を用いた並列計算環境を、1 つの GPU を用いて個人レベルで実際に構築し、GPU 並列計算の基本的特性、特にワープ・スレッド数 (ブロックサイズ) と計算時間の関係や、GPU の世代間の相違を明確にするために行った計算結果の報告を行う。

2　並列化の手法

拙著「数値計算の最適化について」[1] では、OpenMP 等を用いた並列化について議論した。OpenMP は C や Fortran で記述されたプログラム内に、ディレクティブ (指示行) と呼ばれるコメントを挿入することにより、単一 CPU 内にある複数のコア (マルチコア) を用いて並列計算を行う。つまり OpenMP による並列化はコンパイラの機能で実現されている。従って OpenMP 等に対応しているコンパイラがあれば、並列計算を行うことができる。gcc や gfortran、また Intel、PGI 等の商用コンパイラは全て OpenMP 等に対応している。

OpenMP による並列化以外に、複数のノードを用いて並列化を行う MPI(Message Passing Interface) という方法がある。MPI は通常のプログラムに並列化のための関数やサブルーチンの呼び出しを付加することによって複数のノードを用いた並列化を行う。従って MPI は OpenMP と異なりライブラリを用いて並列化を行うものであり、コンパイラの機能で実現するものではない。「京」などはこの部類に入る。

上記以外に GPU 並列計算機がある。GPU を用いた計算機の並列化は、ここ 10 数年で急激に普及した並列化の手法である。GPU はもともとコンピュータの描画を高速で行うために開発されたデバイス (ハードウェア) であるが、その超並列性・高速性を数値計算に利用したいという人たちが、描画用のプログラミング言語 OpenGL を用いて数値計算を行ったのが始まりである。ただし OpenGL はあくまでも描画用に特化したプログラミング言語であり、必ずしも数値計算に適した言語ではない。ところが GPU メーカである NVIDIA が、新たな市場を求めて数値計算分野に対し、CUDA(Compute Unified Device Architecture:クーダ) と呼ばれる汎用並列コンピューティングプラットフォームを提供するようになった。それに含まれる C/C++コンパイラ (`nvcc`) を用いて通常の C/C++プログラムに GPU 用の関数を付加する形でプログラミングが可能となり、GPU を用いた数値計算が急速に普及するに至ったのである。GPU 並列計算機は OpenMP や MPI と異

なり、専用ハードウェア+専用コンパイラを用いた並列化と捉えることができる。今では描画用の出力機能を持たない数値計算専用の GPU も販売されており、現在計算速度 TOP500 のうち GPU を用いた計算機がかなりの割合を占めるようになってきたようである。先に示した TUBAME(つばめ) はこの部類の計算機である。

3　並列計算環境

大型の並列計算機が設置されている時代に、そもそも、並列計算環境を個人レベルで構築する必要はあるであろうか。現代のいわゆるスーパーコンピュータ (スパコン) は、万のオーダーのノードを持った計算機であり、何十〜何百もの GPU を備えた計算機である。したがって個人レベルで数個のノードや 1 つの GPU で並列計算環境を整えたところで意味はあるのであろうか。「所詮おもちゃの計算機」「本物のスパコンには遠く及ばない」と陰口を言われるだけであろうか。まず個人レベルで並列計算環境を構築する意味について考察してみよう。

世界のトップレベルのスパコンは、我々が使っているパソコン (PC) とは比べることはできないほどの性能を持っている。近年の個人用 PC の性能は目を見張るものがあるが、メモリを大量に使う計算や、非常に時間がかかる計算を PC で行うことはできない。従って、トップレベルの計算機を使いたい研究者は大勢いるであろう。にもかかわらずそのような計算機は世界的に見ても利用可能なものはそれほど多数あるわけではない。つまり世界最高性能の計算機を 1 人で独占することはできない。計算機資源を大勢の人で同時に分け合うことになる。するとどんなに性能がいい計算機であっても、計算を依頼してから結果が出るまでに反って時間がかかったり、計算結果を解析するために、データを自分の計算機に転送するための手間がかかったり、お試し計算をすることができなかったり等々、結構制約が多のである。真にそのようなスパコンを使わなければならない計算であれば、スパコンを利用するしかないが、プログラムが正しく動くことを検証したり、細かい最適化を行ったり、パラメータを様々に変えて計算を実行したり、といったテスト計算をするだけのためにスパコンを利用する必要はないであろう。

それよりも使いたいときに自由に使えて、すぐに結果が得られ、すぐにプログラムや結果の検証が行える計算環境が身近にある方が良い場合もあるだろう。従って将来スパコンへと結びつくような並列計算環境を個人的レベルで構築することも意味があると考えられる。

4　環境の構築

では実際どのようにして計算環境を構築すればよいであろうか。

近年の PC の CPU はマルチコアであるのが一般的なので、OpenMP に関しては、対応したコンパイラさえあれば、すぐに並列計算環境を構築することができる。上でも述べたが、gcc や gfortran は OpenMP に対応しているので、Linux 上で OpenMP 環境を整えるのは容易である。実際 Linux 自身も含め、gcc さらに gfortran も無料であるので、PC が 1 台あれば OpenMP 並列計算環境はすぐに構築できる。

MPI に関しては、複数ノード (複数の CPU、複数台の PC) と MPI 用のライブラリと、コンパイラを用意することができればやはり並列計算環境を整えることが可能となる。単一ノード (1 台の PC) でも CPU がマルチコアであるため、MPI 環境は実現可能である。しかしノード間通信等の検証も重要になるので、複数台の PC を用いたほうが単一 CPU では分からない様々な問題点を

見出すことができる。従って、MPI 並列計算環境を整えるには、複数台の PC を用意した方が良いため OpenMP の場合よりやや敷居が高くなる。

GPU 並列計算環境についてであるが、MPI のような同質 (homegeneous) の並列計算環境に比べ、GPU 並列環境はいわゆる異質 (heterogeneous) な計算環境になるため敷居が高いと思っていた。PC に加え別途 CUDA に対応した GPU が少なくとも 1 つ必要であるが、実のところ環境構築自体はあっけないほど容易である。実際の構築の手順としては、Linux をインストールした PC に、

1. GPU を 1 枚 PCI-Express × 16 のスロットに挿す。
2. GPU のデバイスドライバと、CUDA Toolkit をインストールする。[1]

だけである。CUDA Toolkit にはコンパイラ (nvcc)、デバッガ、プロファイラ、ドキュメント等が含まれている。したがって PC1 台と GPU が 1 つあれば、並列計算機の環境とプログラミングの環境が揃うわけである。ちなみにデバイスドライバも CUDA Toolkit も無料である。そこで今回 GPU を用いた並列計算環境を手持ちの Linux マシン上に構築してみた。今回構築した環境は以下の通りである。

CPU:	Intel Core i7 7700K
メモリ:	64GiB
OS:	Ubuntu 18.04
GPU:	Geforce GTX 1060 6GB
CUDA:	Ver 9.1

上記の計算環境を用いて、基本的な並列計算を行った結果を以下で報告する。

5　GPU 並列プログラミング

さて、GPU 並列計算機環境は容易に構築できたが、実際その環境を使ってプログラミングを行うにはまだ少し高いハードルがある。CUDA 独自のプログラミングと GPU のハードウェアの理解は必須である。

CUDA 付属のコンパイラ nvcc は ANSI C を拡張したコンパイラである。CUDA Fortran[2]コンパイラもあるが、これは別途用意しなければならないので、とりあえず CUDA C で書かれたプログラムを実行してみることにする。ここではまず、参考文献 [2] に掲載されている例題を参照することにした。さらに GPU に関する正確な知識・概念が全くない状態では CUDA C プログラミングを理解することはできない。そこで GPU のハード等が詳しく書かれた参考文献 [3] も参照した。

5.1　スレッド、ブロック、グリッド

「GPU は多数の演算器が数百～数千あり、それらが並列で計算を行う」ということは理解していても、それだけでは GPU を用いたプログラミングを行うことはできない。まず CUDA はスレッドと呼ばれる単位で GPU に計算をさせるということを理解しないといけない。そして各スレッド

[1] ドライバも CUDA Toolkit も Linux(Ubuntu) のリポジトリにあるため、マウスでクリックするだけでインストール可能である。CUDA の最新バージョンは 9.2(本稿執筆時) で、リポジトリにあるのは 9.1 という違いはあるが、特に問題はない。

[2] PGI(Portland Group International) Fortran 等。PGI は 2013 年に NVIDIA に買収された。

はカーネルと呼ばれる関数で記述され、GPU の演算器で並列に実行される。さらにスレッドが集まってブロックという単位を作り、ブロックが集まってグリッドを構成する。ブロック内のスレッドは 1〜3 次元の配置が可能であり、グリッド内のブロックも同様に 1〜3 次元の配置が可能である。CUDA のプログラムはこのようなスレッド–ブロック–グリッドの階層構造を形成している。これらは通常の C 言語にはない概念である。では「なぜこんな概念が必要なのだろうか」それには GPU のハード (アーキテクチャ) の知識が必要になる。

5.2　コア、SM

GPU の演算器の最小単位はコアと呼ばれる。GPU 上にはコアが数 100〜数 1,000 個あり、それらを用いて並列計算を行う。個々のスレッドの計算がコアで実行される。このコアが 16、32、64･･･個集まったものが SM(Streaming Multiprocessor) と呼ばれるものである (個数はハードウェアに依存)。この SM が数個から数 10 個集まって GPU を構成している。計算するときは個々のブロックが SM に割り当てられることになる。GPU の構造を簡単にまとめると以上のようになる。

コア数、SM 数というハード的な制約がスレッド数、ブロック数を制限しているかと言うと、そういうわけではない。CUDA 側から見た論理的なスレッド、ブロック数は GPU のコア数、SM 数とは必ずしも一致しない。もっとも GPU の種類によってブロック、スレッド数の総数にはそれぞれ異なる上限値がある。

では、ここまで理解したところでプログラムを実行してみることにする。

6　プログラムの実行と計算時間の計測

まず、1 番簡単なスレッド・ブロックを 1 次元に配置した場合のプログラムを実行することにする。用いたプログラムは、参考文献 [2] の「リスト 2-5」にある `sumArrayOnGPU-timer.cu` である[3]。

このプログラムは 2 つの長大ベクトル (要素数:$2^{24} = 16,777,216$) の和を求めるプログラムである。和を求めるためのカーネル関数は以下の通りである。変数の意味等は参考文献 [2] を参照してほしい。

```
__global__ void sumArraysOnGPU(float *A, float *B, float *C, const int N)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N) C[i] = A[i] + B[i];
}
```

1 つのコア (スレッド) でベクトルの 1 つの成分の和を計算するわけである。また、カーネル関数の呼び出し部分は

```
sumArraysOnGPU<<<grid, block>>>(d_A, d_B, d_C, nElem);
```

である。`<<<grid, block>>>`の部分が通常の C の関数呼び出しと異なる部分で、「`grid=`グリッドサイズ=グリッド内の**ブロック数**」、及び「`block=`ブロックサイズ=ブロック内の**スレッド数**」を

[3]CUDA C プログラムの拡張子は `.cu` である。

指定している。このカーネル関数の 1 回の呼び出しで `grid`× `block`= 2^{24} 個のスレッドが起動し、GPU で並列に処理される。本稿では用語として主に**ブロック数**、**スレッド数**を用いる。

参考文献 [2] では 1 ブロック当たりのスレッド数を変えて GPU の計算時間を計測している。スレッド数が 1,024 の時の計算時間が `0.002456sec`、512 の時が `0.002058sec` で 1.19 倍パフォーマンスが向上したとある。スレッド数 256 の場合は、ブロック数が当時の GPU の制限値を超えたため測定できていない。スレッドの総数は 2^{24} で変化しないので、当然スレッド数に連動してブロック数も変化する。参考文献 [2] ではブロック数が多い (ブロック当たりのスレッド数が小さい) 方がパフォーマンスが良くなると、たった 2 つの計測結果から結論づけている。では今回構築した計算環境ではどうであろう。以下は 100 回同じ計算をした場合の平均計算時間である。測定誤差はあとで述べるスレッド数間のばらつきよりも小さかったのでここでは問題にしない。

GTX 1060　スレッド数	計算時間 (sec)
256	0.001263
512	0.001276
1,024	0.001287

参考文献 [2] にあるような顕著な違い (スレッド数が小さいほうが約 20%程度向上) は見出すことができなかった。参考文献 [2] のように 2 つの計測から、本稿では 3 つの計測から結論づけるのも強引な気もするが、使用した GPU が異なるので、計算時間を測定した GPU の違いのために、差を見出すことができなかったのかも知れない。実際今回用いた GPU(GTX 1060) と参考文献 [2] で用いた GPU(Tesla M2070) とでは GPU のアーキテクチャが異なっている。[4]

NVIDIA の GPU は GPU のハードウェアバージョン (世代) を表すために Compute Capability(CC) という用語 (番号) を用いている。また CC の違いにより著名な科学者の名前 (クラス名) がつけられている。

Compute Capability(CC)	クラス名
2.x	Fermi
3.x	Kepler
5.x	Maxwell
6.x	Pascal

参考文献 [2] で用いられた Tesla M2070 は Fermi 世代の GPU であり、Geforce GTX 1060 が Pascal 世代の GPU である。

Fermi の 1 次元グリッドサイズ (グリッドあたりのブロック数) は $2^{16} - 1 = 65,535$ が上限であるが、Kepler 以降は大幅に制限が緩和され $2^{31} - 1 = 2,147,483,647$ である。Fermi と Kepler 以降はかなり大きなアーキテクチャの違いがあるようで、実際 CUDA Toolkit 9.1 では Fermi 世代の GPU は対象外となっている。

そこでまず、Maxell 世代の GPU である GTX 750Ti を用いて計算時間を計測してみた。以下がその結果である。

[4]参考文献 [2] で使用された Tesla M2070 は数値計算用の GPU であり、今回用いた Geforce GTX 1060 はコンシューマ向けの (数値計算も可能な) ゲーム用 GPU であるという点も異なっている。

GTX 750Ti　スレッド数	計算時間 (sec)
256	0.002733
512	0.002722
1,024	0.002723

この結果から分かるように、Maxwell 世代の GPU を用いても、スレッド数の違いによる明確な計算時間の違いは見いだすことはできない。

では、Kepler 世代の GPU ではどうであろうか。幸い Kepler 世代の GPU(Geforce GTX 660) が手元にあるのでそれを用いて同様な計測を行った。その結果は

GTX 660　スレッド数	計算時間 (sec)
256	0.001840
512	0.001866
1,024	0.001895

である。やはりスレッド数 (ブロック数) の違いによる計算時間の差はない。ちなみに GTX 660 より新しい GPU である GTX 750Ti の方が計算時間がかかっているが、これは使用した GTX 750Ti が、同シリーズの GPU の中でローエンドの部類であり、一世代前の GTX 660 より性能が低いのが原因である。

Fermi 世代の GPU でも試してみたいがもはや手に入れることはできないし、Toolkit も対応していないので諦めるしかない。

ここまでスレッド数を 3 つ (参考文献 [2] では 2 つ) だけ選んで計算時間を計測したが、これだけでスレッド数と計算時間の関係を結論づけるには不安が残る。特に選ばれたスレッド数は計算機にとって特別な意味のある 2 の冪乗であるため、何か特殊な影響が現れているのかも知れない。そこで、もう少し詳しくスレッド数依存性を調べるために、スレッド数を 1〜1,024 まで 1 つづつ変えて計算時間を測定してみた[5]。結果をプロットしたのが図 1 である。上記と同様にそれぞれ 100 回の測定値の平均である。縦軸が長くなってしまって、特徴がよく分からないので、縦軸の範囲を制限して表示したのが図 2 である。

この図を見ても左端 (スレッド数が小さい場合) を除いて、参考文献 [2] のようなブロック数の違い (スレッド数が小さいほどブロック数が多い) による計算時間の違いを見て取ることはできない。参考文献 [2] の原著が出版されたのが 2014 年でそれほど古いものではない (邦訳の出版年は 2015 年。2018 年には 2 刷が出版されている)。にもかかわらず、参考文献に書いてある内容と最近のハードウェア GTX 1060 とは随分異なっている。参考文献 [2] の結果と異なるのは明白である。

原因は Fermi 世代 (Tesla M2070) と、今回用いた Pascal 世代の GPU(GTX 1060) の違いによるものであろうか。そこで同様の測定を Maxwll 世代 (GTX 750Ti) と Kepler 世代 (GTX 660) についても行った。その結果を図 3、4 に示す。

GTX 750Ti に関しては、GTX 1060 とそれほど大きな違いは見られないが、GTX 660 に関しては他の GPU には見られない特徴的なスレッド数依存性が見られる。まず完全とは言えないが周期的な振る舞いをしている。またスレッド数の増加に伴いスレッド数 1000 の付近では 1 割ほど計算時間が長くなっているのが見て取れる (`0.00205sec` 程度)。つまり参考文献 [2] と程度の差はあるが、同様の傾向が見られる。しかし、図からはよく分からないが、スレッド数 1,024 の計算時間

[5] スレッド数 1,024 はハードウエアの上限である。

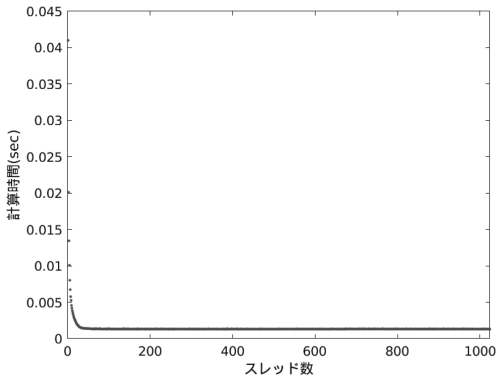

図 1: GTX 1060

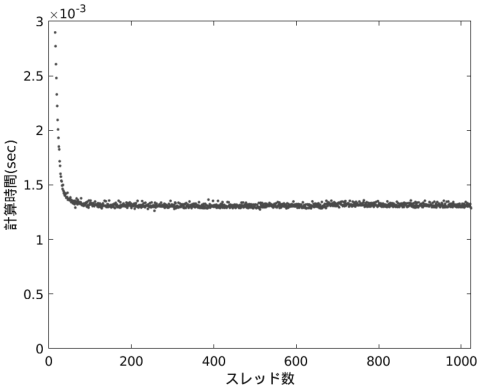

図 2: GTX 1060: 図 1 の拡大図

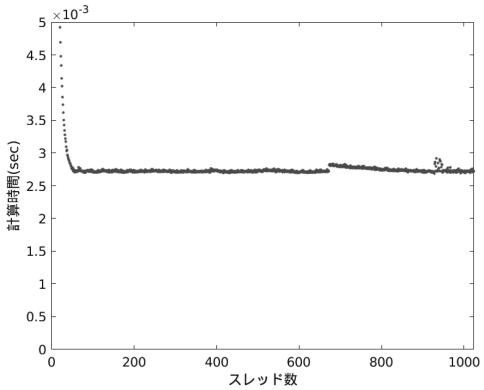

図 3: GTX 750Ti

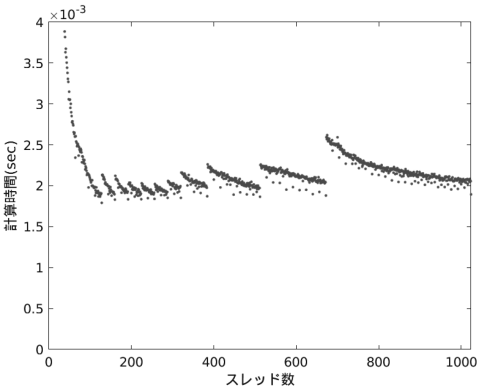

図 4: GTX 660

だけが `0.001895sec` と短くなっている。このためスレッド数 256、512、1,024 を選んで測定したとき計算時間の増加を見ることができなかったのである。

このようにピンポイントで 3 つだけ (あるいは 2 つだけ) スレッド数を選んで測定し、結論を導くことがいかに危険であるかが分かるであろう。

またもう一つ GTX 660 の特徴として、スレッド数 700 弱のところで異常に計算時間が長くかかっているのが分かる (計算時間の飛び)。

これらの計算時間の振る舞いの原因を探るために、まず図 1〜4 の左端に見える計算時間の著しい増加と、図 4 に見られる準周期的な振る舞いについて考えてみることにする。

7　ワープ

各ブロックにスレッドをいくつ割り当てようが、スレッドの総数は 2^{24} で変わらないのだから、全てのコアがまんべんなく動けば計算時間に差は現れないはずである。図 2、3 は左端を除けばもっ

ともらしい結果である。しかしそれでは左端のような振る舞いや、図 4 のような準周期的な振る舞いが現れる理由がない。ではこれらの振る舞いは何を意味しているのだろうか。実は NVIDA の GPU では同一ブロック内のスレッドはスレッド数にかかわらずワープと呼ばれるまとまった単位で同時に実行されるのである。ワープサイズは NVIDIA の全ての GPU に対して、32 に固定されている。

例えば 1 ブロック内のスレッド数が 1 であるとすると、32 スレッドで 1 ワープを構成するので、1 ワープに割り当てられた 32 個のコアの内、31 個のコアは計算をせずに遊んでしまうことになる。従ってスレッド数 1〜32 までの間の計算時間は、スレッド数に反比例することになる。

$$\text{計算時間} = \frac{A}{\text{スレッド数}} \tag{1}$$

ここで A は規格化の定数である。このように考えると図 2〜4 の左端の振る舞いはもっともらしく思われる。しかしワープの影響は当然スレッド数 33 以上でも現れると考えられるが、図 2、3 からは左端以外、ワープの影響が見られない。32 の周期性が見られないのである。図 4 も周期的とは言え完全に周期 32 とはなっていない。それはなぜだろうか。

まず図 2、3 に対して考えられるのは、カーネル関数として用いた 2 つの実数の和を取るという計算が簡単すぎて 33 スレッド以上では計算時間のスレッド数間のばらつきが大きいので、その中に埋もれてしまっているということである。図 4 に見られる計算時間のスレッド数依存性の原因は不明のままであるが、まず、計算式を少し複雑にしてみて (掛け算や割り算を適当に加えてみて) 計算時間を計測して見た。具体的には、先に挙げたプログラムの和の計算部分 `C[i] = A[i] + B[i];` を

```
C[i] = A[i] * B[i] + A[i] / 2.0 + B[i] / 3.0 + A[i] - B[i];
```

に変更した。その結果が図 5、6 である。

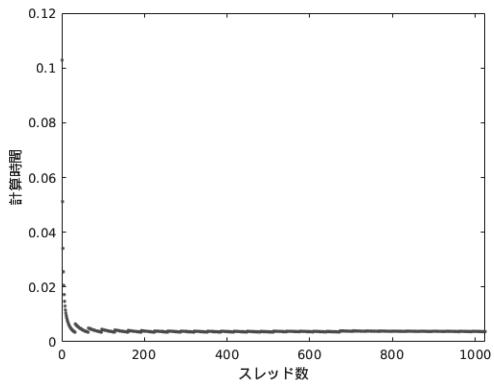

図 5: GTX 1060

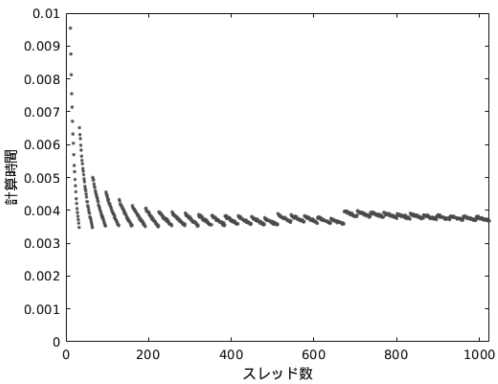

図 6: GTX 1060: 図 5 の拡大図

同様に GTX 750Ti と GTX 660 についても図 7、8 に示す。この図から全ての GPU に対して周期性が現れているのが分かるであろう。GTX 660 に関しても図 4 と異なり、明らかな周期性が現れていることが分かる。また、スレッド数の増加に伴い若干の計算時間の増加が見られる。

図 6〜8 を詳しく見てみるとスレッド数が比較的小さい部分ではスレッド数 32 周期で計算時間に変化が現れていて、ワープサイズ 32 の影響が現れていると考えられる。つまりワープに 32 個ず

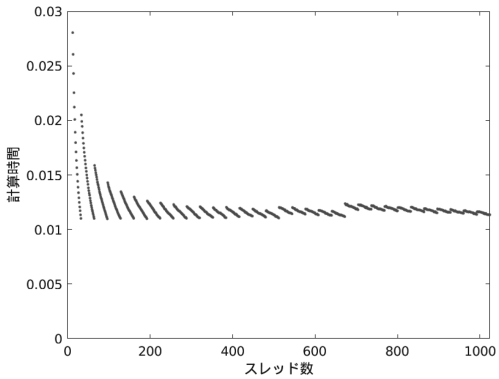

図 7: GTX 750Ti

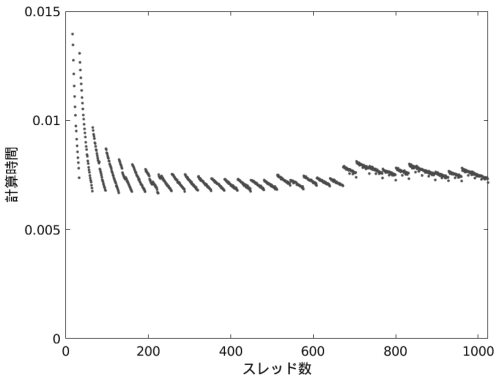

図 8: GTX 660

つスレッドを分割していくと、各ブロックに割り当てるスレッド数が 32 の倍数になっていなければ、最後のワープにだけ遊んでしまうコアができてしまい、計算時間に影響する。

単純に考えると、1 ブロック当たり丁度 32 の倍数個だけスレッドを割り当てればワープの影響は現れないはずであるから、計算時間は次の式に従うと考えられる。

$$\text{計算時間} = \frac{A}{\text{スレッド数}} \times \text{ワープ数} \tag{2}$$

例えばブロック内のスレッド数が 33〜64 であればそのブロックに必要なワープ数は 2 である。GTX 1060 に対して、スレッド数 1〜128 の場合の計算時間をプロットしたのが図 9 である。実線が式 (2) のグラフである。規格化定数 A はスレッド数 32 の時の計算時間から求めたものを使った。式 (2)

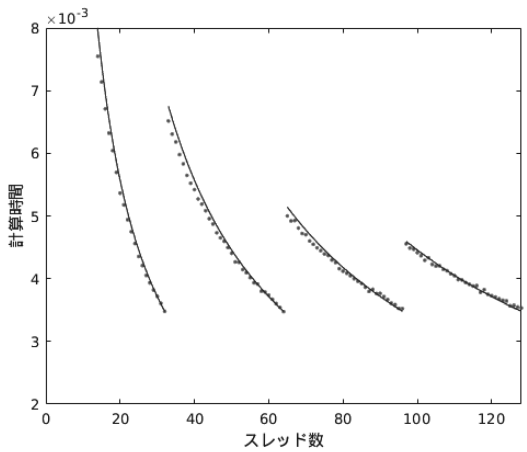

図 9: GTX 1060

との一致は見事なものである。特にスレッド数 97〜128 の間は実線と実測値の点がほぼ完璧に一致している。このように、スレッド数が比較的小さい場合はスレッド数と計算時間の関係は式 (2)

でよく一致することが明らかとなった。参考文献 [2] で測定したスレッド数 (512 と 1,024) は 32 の倍数であり、計算時間の差はワープの影響ではない。

このように計算時間の周期性はワープサイズに原因を求めることができた。しかしスレッド数が大きくなるに連れ、特にスレッド数 700 付近での計算時間の「飛び (突然計算時間が長くなる)」現象が見られることは意外な結果である。これらの原因は何であろうか。これはブロック数が影響しているのであろうか。そこで原因を探るために、更にカーネル関数に組み込み関数 (sin, cos) の項をつけ加え

```
C[i] = A[i] * B[i] + A[i] / 2.0 + B[i] / 3.0
                   + A[i] - B[i] + sin(A[i]) + cos(B[i]);
```

のように変更して計算時間を計測した。図 10 にその結果を示す。

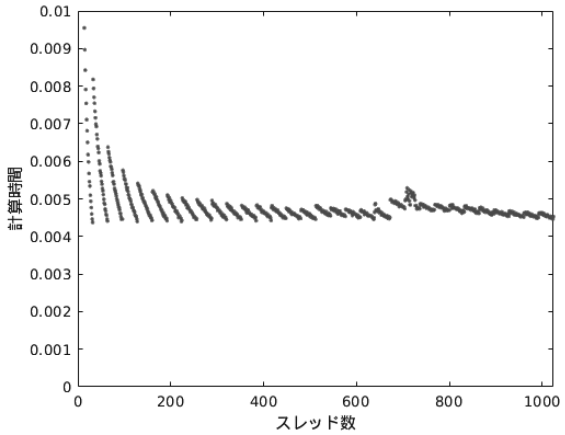

図 10: GTX 1060

この図を見ると、スレッド数 670 前後の挙動が図 6 よりも怪しくなっている。つまり計算の複雑さの問題ではないようである。「飛び」の出現箇所には特に規則性もないので、ブロック数が問題になるとも思えない。すると今回使用した GPU 固有の問題なのか、あるいは GPU 一般に言えることなのであろうか。

そこで同じ計算を先ほどと同様に GTX 750Ti と GTX 660 を用いて計算時間を計測してみた。結果を図 11、12 に示す。やはりスレッド数 670 付近で計算時間に「飛び」が見られる。この現象は少なくともこの 3 種類の GPU に共通の問題である。

図 2、3 で明確に現れなかった 32 周期が計算を少し複雑にすることで現れたが、このことが本当に計算の複雑さにあるのかどうか、また図 4 の様に完全ではない周期的な振る舞いや、「飛び」が現れるのは何故か、と言った原因は残念ながら本研究からは明らかにすることはできない。実際の計算ではカーネル関数で単に 2 つの実数の和を計算させることはありえないし、「飛び」の原因がハードウェアにあるとすればもはや原因を追求する術はないので、ここではこれ以上追求することはしない。

今までの結果から導くことのできる結論として、各ブロックに 32 の倍数でスレッドを割り当てた場合、参考文献 [2] のような 20%もの著しいな計算時間の差は見られなかった。計算パフォーマンスを求めるのであれば、ブロック数は多い (スレッド数は小さい) 方が良く、スレッド数は 32 の

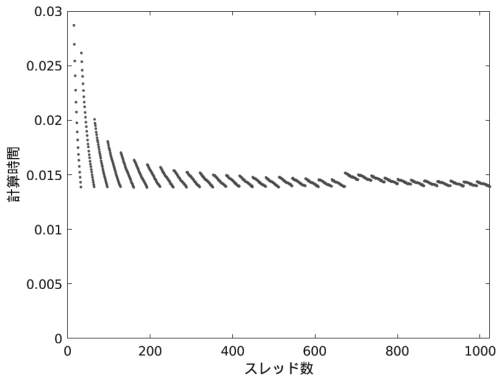

図 11: GTX 750Ti

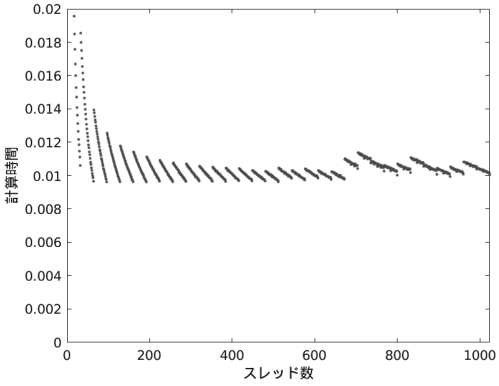

図 12: GTX 660

倍数にするべきであるのは明らかである。さらに「飛び」が現れるスレッド数は避けるべきである。特に Kepler 世代の GTX 660 ではスレッド数 600 以上で理解不明の挙動が現れるのでやはりスレッド数は小さく取るべきである。

本稿で調べたような、可能なスレッド数に対して全て計算時間を計測したり、複数の世代の GPU に取り替えてその特性を調べたりすることは、とても大型の並列計算機を利用してできることではない。当然プログラミングの基礎を習得するのに大型並列計算機を用いることはできない。個人的に並列計算環境を持つ有意さがこの当たりに現れている。

8　まとめと展望

今回個人レベルで GPU 並列計算環境を構築し、その特性について調べた。その結果個人レベルで GPU 並列計算環境を構築するのは極めて容易であることが分かった。プログラミングは通常の C/C++と異なっており、CUDA C 独特の関数の使い方等を新たに習得する必要がある。また GPU のアーキテクチャの特徴をしっかり捉えないと計算のパフォーマンスを向上させることができないことが明らかとなった。つまりスレッド階層構造の中でスレッド数が計算時間に影響を与えることが明らかとなった。特に NVIDIA の GPU の仕様であるスレッドが 32 個集まってワープを構成することをしっかり理解することが重要である。また GPU の世代間の違いも大きいため、各世代の特徴をしっかり把握することも重要であることが明らかとなった。

本稿で示したように、計算時間とワープの関係を詳細に図示したものはない。さらに計算時間とワープの関係を式 (2) ではっきり示すことができたのも本稿の重要な結論である。これは、1〜1,024 スレッド全てに渡って計算時間を計測し、図示して詳細に考察を行ったからである。スレッド数対計算時間を詳細に調べたことと、式 (2) を示したことは本稿で初めてであると思われる。これらの結果を参考に効率的なプログラミングが可能となるであろう。とりあえずブロック数は可能な限り多めにとり、ブロック内のスレッド数は必ず 32(ワープサイズ) の倍数になるように取るのがよい。

実は今回の研究で、GTX 1060 を用いると CPU だけで計算するよりも 8 倍程度高速化されることが明らかとなった。ただ GPU を用いてスレッド数に注意すればどのような計算でも高速になる

かと言うとそう簡単ではない。今回の稿では言及しなかったが、CPU 側から GPU 側へのデータ転送のオーバーヘッドも実は計算時間に多大な影響を及ぼす。また GPU 側のメモリの階層構造もプログラミングをする時に気をつけなければならない重要な点である。同様に今回はスレッドの階層構造としてブロック 1 次元、グリッド 1 次元だけに焦点を当てたが、2、3 次元にした場合メモリアクセスと合わせてパフォーマンスにどのような影響を及ぼすのかを確認することも大切である。これらについてはまた稿を改めて報告する機会があればよいと思う。

さらに GPU 並列プログラミングのコンパイラは CUDA 付属の `nvcc` や CUDA Fortran だけではない。OpenCL は NVIDIA 以外の GPU にも対応した GPU 対応コンパイラの規格である。また OpenMP Ver4.x もディレクティブを用いて GPU プログラミングを可能とする規格である。同様に OpenACC もディレクティブを用いたコンパイラの規格となっている。これらの規格に対応したコンパイラがあれば様々な角度から、並列計算の可能性を探ることができる。

近年の GPU 並列計算機の普及は凄まじいものがあり、GPU の性能の進化も著しい。CC が 7.x である Volta 世代の GPU も発売されている。GPU の個々の特性や世代間の相違を見極めることは、計算パフォーマンスを追求する上で重要なことである。スレッド数依存性を詳細に考察したり、複数の世代の異なる GPU を取り替えて世代間依存を調べることができたのも、個人的 GPU 計算環境があればこそであろう。

CUDA C プログラミングを実行するにあたり参考文献 [2][3] を参考にしたが、特に参考文献 [2] のように、必ずしも正しいことが書かれているわけではない。教科書の結果を鵜呑みにするのではなく、自らの手で調べることは重要である。

今回の研究でワープと言う概念が、NVIDIA GPU を利用する上でまず第一に重要なことであることが分かったが、上述したようにメモリ階層構造等並列計算のパフォーマンスを向上させるためには知って置かなければならないことも多数ある。しかし理屈を追いかけるよりもまず実際の CUDA API を理解し、具体的なプログラムを多数実行してみて、実際の数値計算に応用することも次の重要な目標になるであろう。

参考文献

[1] 伊藤 誠 (2012) 数値計算の最適化について (大阪産業大学経済論集 Vol. 13 No. 2)

[2] 森野慎也監訳 (2015) CUDA C プロフェッショナル プログラミング (インプレス)
原著: Cheng, Grossman, McKercher (2014) Professional CUDA C Programming (Wrox)

[3] Hisa Ando (2017) GPU を支える技術 (技術評論社)

Construction of the GPU Parallel Calculation Environment and Thread Hierarchy Structure

ITOH Makoto

Key Words: GPU, CUDA, Warp, parallel calculation

Abstract

We constructed the personal parallel calculation environment using GPU and CUDA Toolkit. It is very important to recognize the concept of Warp of NVIDIA GPU in CUDA programming. We showed that Warp size(32) had the decisive influence on calculation time. We also showed the relation between number of threads and calculation time, and in the CUDA program the number of the threads in each block must be a multiple of 32(Warp size).