卒業論文

GPU を用いた行列演算の高速化

氏　　名：桑山　裕樹
学籍番号：2260060042-0
指導教員：山崎 勝弘　教授
提 出 日：2010 年 2 月 17 日

立命館大学　理工学部　電子情報デザイン学科

内容梗概

最近では 3D グラフィックスなどを扱うゲーム等が広く普及してきている。そこでその処理で活躍している技術に GPU といわれるものがある。本来 GPU とは描画処理の高速化を目的とした処理装置である。

この GPU のグラフィックス処理能力を画像処理以外である汎用的な計算にも行えるようにした技術が GPGPU もしくは GPU コンピューティングと呼ばれるものであり、今日性能が CPU より高く、注目を集めている。本研究はこの注目を集めている GPU を用いて並列化を行い、高速化を目的としている。

今回使用しているマシンは GPU コンピューティングの 1 つである Tesla C1060 を用いており、その中で自由に汎用計算を行える新しいハードウェア・アーキテクチャ、およびソフトウェア環境である CUDA を使用している。CUDA では C 言語に似た言語使用となっており、GPU 上への汎用計算をマッピングすることができる。

本研究では、この CUDA を使用し、行列演算の高速化を行っている。元は描画処理向けであった GPU を汎用的な計算にも行えるようにした CUDA で行列演算の処理時間を計測し、その評価を目的としている。その性能を評価するため、その処理時間を CPU で同じ処理にかかった処理時間と比較をしている。また、スレッド数別での計測をし、比較すること、スレッド数における性能の評価を行った。

目次

図目次

表目次

1. はじめに

現在、コンピュータの性能は格段にあがり、パソコンであっても少し前では信じられないような機能が実現できるようになった。特にビデオカードの表示性能は高くなり、本物とみまがうばかりの非常にクオリティの高い描画を行える。[1]

今日では 3D(3 次元)グラフィックスを扱うゲームや 3D CAD(Computer Aided Design)などが広く普及している。3D グラフィックス処理の分野では、長年、並列処理技術が利用されてきた。このような分野で GPU(Graphics Processing Unit)が使用されている。汎用的な計算を得意とする一般のマイクロプロセッサ（CPU）で同じ処理を実行した場合、GPU ほどのグラフィックス処理能力を得ることはできない。このため GPU は、グラフィックス・ボードの形でパソコンに組み込まれたり、PlayStation3 や Wii といった家庭用ゲーム機にも搭載されたりしている。しかもここ数年では GPU は演算性能、メモリバンド幅とともに CPU のそれら大きく上回るようになってきているのだ。[7]

この演算能力に着目し、グラフィックス分野で蓄積された並列処理技術の資産を応用しようというのが、3D グラフィックス処理以外の汎用的な計算を行わせる「GPGPU（General Purpose Computation on Graphics Processing Unit）」、あるいは「GPU コンピューティング」と呼ばれる技術がここ数年、注目を集めている。GPGPU は、画像処理はもちろんのこと、流体計算、電磁波シミュレーション、天文シミュレーション、たんぱく質などの挙動を解析する数値シミュレーション、バイオ・インフォマティクス、金融工学など、 幅広い分野への適用が行われている。このような試みは数値計算の分野においても活発である。また、GPU コンピューティングと GPGPU とを正式に区別することは難しいが、簡単に言えば、GPGPU よりも本腰を入れ、より汎用的な HPC（High Performance Computing）の実現に向けた取り組みが GPU コンピューティングである。[7]

本研究ではこの GPU コンピューティングの一つである Tesla C1060 を用いて、行列演算の高速化を目的とする。行列演算は行列積と行列演算を行っている。まず、CPU における 2 つの行列演算の処理時間を計測し、次に GPU における処理時間を計測する。その 2 つを比較し、処理時間の違いを検証する。その後はスレッド別ごとの処理時間の比較をする。スレッド数を変えることで処理時間の違いを検証していく。

本研究の第 2 章では本研究で使用している GPU 、Tesla C1060 の説明と NVIDIA 社が提供する GPU 向けの C 言語の統合開発環境である CUDA の説明をする。
第 3 章では、GPU での並列化、行列演算がどのように並列化されているかについて述べる。
第 4 章では実験として GPU と CPU との行列演算の処理時間の比較を行い、その違いのついての検証、また、スレッド別における処理時間の比較の検証を行い、最後にそれらについての考察を述べる。第 5 章では本論文のまとめについて述べる。

2. GPU のアーキテクチャ

2．1　Tesla C1060 のアーキテクチャ

GPU というプロセッサの中の一つに本研究で使用する Tesla という GPU コンピューティングプロセッサボードがある。

GPU の機能である 3D グラフィックス処理をそれ以外の汎用的な計算行わせるようにできたのが「GPGPU（General Purpose Computation on Graphics Processing Unit)」、もしくは「GPU コンピューティング」と呼ばれるものである。[7]

GPGPU は、画像処理はもちろんのこと、流体計算、電磁波シミュレーション、天文シミュレーション、たんぱく質などの挙動を解析する数値シミュレーション、バイオ・インフォマティクス、金融工学など、 幅広い分野への適用が行われている。

GPU コンピューティングは、GPGPU よりも、より汎用的な HPC の実現に向けた取り組みが GPU コンピューティングである。Tesla はこの GPU コンピューティングに特化したプロセッサボードなのである。

Tesla は、GPU のメーカーである NVIDIA 社が開発した GPU コンピューティングプロセッサボードの 1 つである。Tesla シリーズは GPU の構成から、映像の出力機能をカットしたものである。Tesla の特徴としてはその精度が挙げられる。

本研究で使用している Tesla C1060 は Tesla シリーズの一つであり、240 個のプロセッサコアに 1296MHz の動作を行うことができる。Tesla C1060 は科学者やアナリスト、その他の技術専門家のための HPC アプリケーションに膨大なマルチスレッドアーキテクチャ を初めてもたらしたものである。

Tesla C1060 には GPU コンピューティングに特化した Tesla GPU を 1 基搭載しており、240 基の CUDA プロセッサコアにより、最大 933GFLOPS の単精度浮動小数点演算性能を実現できる。また、従来では単精度演算のみであった処理を倍精度による演算も可能となり、極めて複雑な集約的計算の解決を可能にする。[15]

Tesla C1060 の詳細を表 1 に示す。また、Tesla のアーキテクチャを図 1 に示す。

表 1　Tesla C1060 の特徴

名前	NVIDIA Tesla C1060
搭載 GPU 数	1 基
CUDA コア数	240 基
プロセッサ周波数	1296MHz
単精度演算性能	最大 933GFlops
搭載メモリ	4GB
メモリインターフェース	GDDR3 SDRAM
浮動小数点	IEEE 754 単精度 / 倍精度浮動小数点
ホスト接続	PCI Express x16(PCI-E2.0 対応)
メモリ転送帯域	102 GB / sec
TDP	187.8 W(標準値)
最大消費電力	225 W

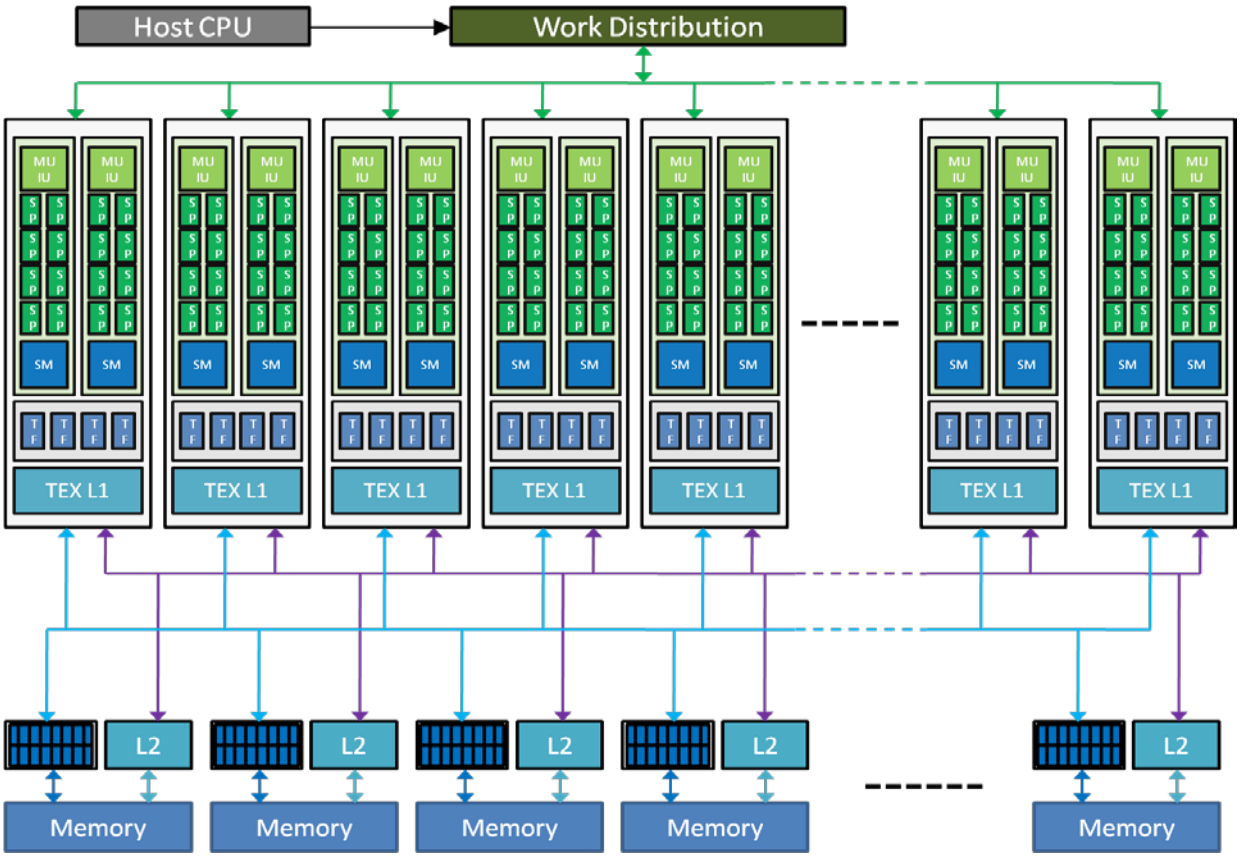

図 1　Tesla のアーキテクチャ

図 1 において、SP はスカラプロセッサであり、MT IU はマルチスレッドインストラクションユニットである。これはどのワープの実行をすべきかの選択を行うのを決めるものである。スカラプロセッサが 8 基集まったものをストリーミングマルチプロセッサという。SM はシェアードメモリである。TF とはテクスチャフィルタという。これはテクスチャユニットのことでテクスチャを呼び出してフィルタリングを行う機能である。TEX L1 はテクスチャキャッシュを指す。L2 は L2 キャッシュである。その隣にある青色のブロックは GPU でいう ROP ユニットに相当する。これは、ピクセルやテクセルのグラフィックスメモリの書き出しを担当するところである。Memory はメモリであり、ここの中ではグローバルメモリ、テクスチャメモリ、コンスタントメモリすべてのことを指している。

細かく説明すると、Tesla のアーキテクチャの中で、いちばん小さい単位がスカラプロセッサという。このプロセッサは CPU 等のプロセッサとは大きく違うところがある。CPU 等のプロセッサには、命令デコードといわれる機能があるが、スカラプロセッサには、命令デコードをするプログラムはない。だからといってスカラプロセッサが次に行う処理が取得できないわけではない。それはストリーミングマルチプロセッサというプロセッサに命令デコード機能が含まれており、これでデコードされた命令がスカラプロセッサに伝えられる。搭載されるすべてのスカラプロセッサに、すべて同じ内容が伝えられる。伝えられたスカラプロセッサでは、同じプログラムが動作する。これにより「同じプログラムが複数のスカラプロセッサ上で並列に動く」ことになる。[1]

これの最大の特徴は、スカラプロセッサ 1 つ 1 つはただ処理をするだけだが、それをまとめたストリーミングマルチプロセッサ上では、同じプログラムが複数のスカラプロセッサ上で並列動作し、大量のデータを並列に処理できる点である。ストリーミングマルチプロセッサの構成を図 2 に示した。

図 2 には、ストリーミングマルチプロセッサには、スカラプロセッサ(SP)とそれを制御するためのユニット、Instruction Unit があり、他には 2 基の Special Function Unit、全スカラプロセッサからアクセス可能な共有メモリである Shared Memory、読み込みのみ定数(Constant)メモリ、全スカラプロセッサからアクセス可能で、テクスチャメモリのアクセス速度を向上させるテクスチャキャッシュ(Texture Cache)メモリがある。

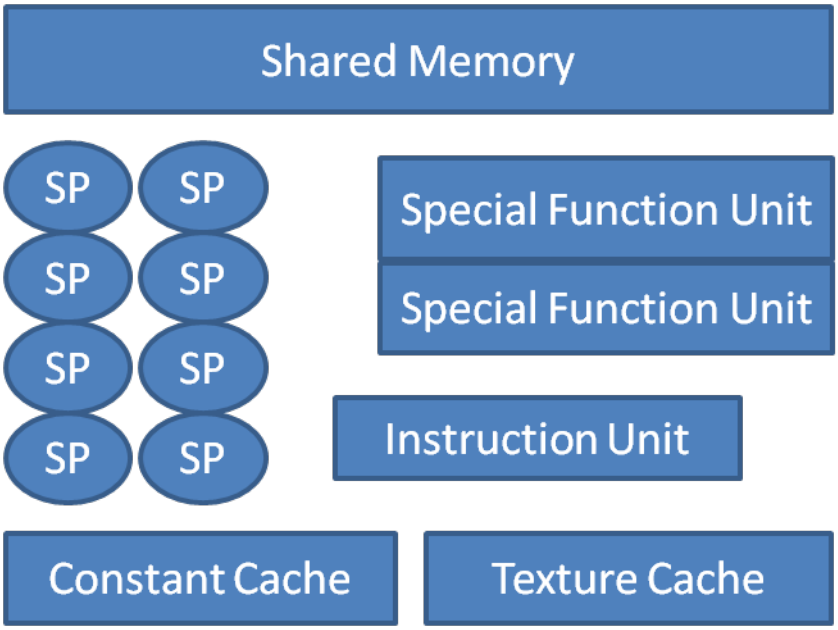

図 2　ストリーミングマルチプロセッサの構成

2．2　並列プログラミング環境 CUDA

今日、数百以上発売されている CUDA 搭載の GPU で、ソフトウェア開発者、科学者、研究者達は、イメージや画像処理、計算処理を行う生物学や化学、流体シミュレーション、CT 画像処理、地質調査、レイトレーシングなど、幅広い CUDA の使用について探求を行われている。

CUDA とは Compute Unified Device Architecture といわれるものであり、GPU 上で自由に汎用計算を行わせるための新しいハードウェア・アーキテクチャ、およびソフトウェア環境である。これは NVIDIA 社のハードウェア上で提供されている開発環境であり、GPGPU のための高レベル API を目指したものである。[2]

CUDA では C 言語に似た言語仕様になっており、3D グラフィックス・ライブラリを使用せずに、GPU 上へ汎用計算をマッピングすることができる。

CUDA 対応の GPU は、すべてのシェーダが同じ機能を持つ統合シェーダと呼ばれるアーキテクチャを採用している。シェーダとは、GPU 内に複数個搭載されている小型プロセッサ・コアのことを言う。統合シェーダそのものは単精度浮動小数点演算を行える小型のプロセッサ・コアとして設計されている。[7]

ソフトウェアにおいては、新しいハードウェア・アーキテクチャをより効率的に使用できるように、CUDA を用いたプログラミングに、「スレッド」、「ブロック」、「グリッド」、「カーネル」などの新しい概念が導入されている。これらは本格的な信号処理計算や科学

技術計算にも耐えうる仕様になっている。

(1) CUDA のプログラミングモデル

CUDA では、CPU 側の系をホストと呼び、GPU 側の系をデバイスと呼ぶ。ホストプログラムと呼ばれるホスト側のプログラムは、ホスト上の CPU で動作し、ホスト上のメモリを利用する。デバイス上で動作するプログラムを、カーネルプログラムという。カーネルとは、CUDA において、GPU 側に実行させるプログラムのことをいう。カーネルプログラムは、C 言語を元に拡張された言語体系で行われており、デバイス上で動作し、GPU が処理を行い、デバイスのメモリを利用する。そのプログラムの流れを図 3 に示す。

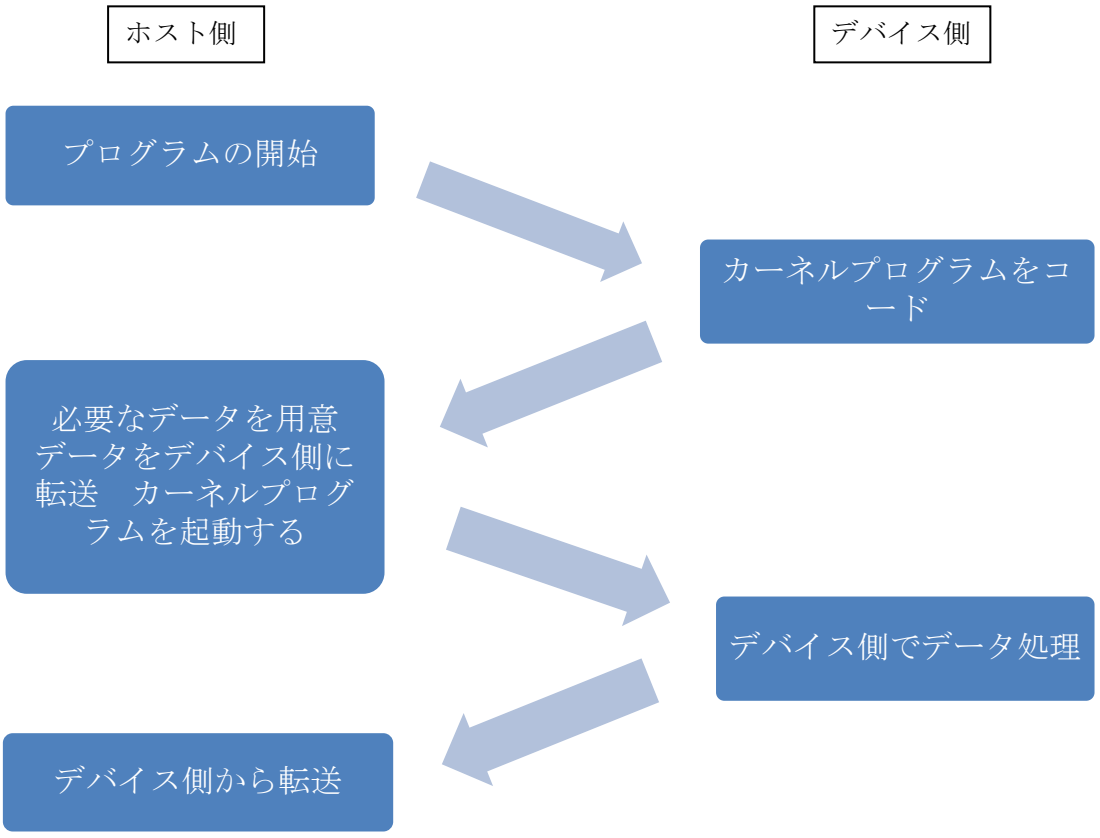

図 3　ホスト／デバイス側のプログラムの流れ

この CUDA プログラミングにおいて注意する点は、メモリ転送である。カーネルプログ

ラムを起動する前に、メモリ上のデータをホスト側からデバイス側に転送する必要がある。また、カーネルプログラムの終了後、結果をホスト側に転送する。処理の前後にメモリ転送の時間がかかるので、その時間を CUDA の並列処理により短縮できるだけの効率が必要である。逆にトータルで考えた場合、メモリ転送にかかる分が短縮できなかった場合、CUDA を利用しない場合も考えられる。

(2) スレッド、ブロック、グリッドの概念

CUDA には新しい概念である、「スレッド」、「ブロック」、「グリッド」というものに分けられている。階層は上から「グリッド」、「ブロック」、「スレッド」となる。

スレッドはスカラプロセッサ上で動作するプログラムの単位である。スレッド自体は、ホスト側から起動される。実行されるプログラムはすべて同じだが、タイミングは別となる。このプログラムの実行がバラバラの状態で動くことを、非同期の動作という。

CUDA のプログラミングで重要なのは、たくさんの処理がスレッドとして並列に動作することである。しかし、処理はすべて非同期である。それを同期にするにはプログラム上での関数として__syncthreads()がある。この関数により、非同期で動作していたプログラムの同期をとることができる。[1]

ブロックはスレッドをさらにまとめたものである。1 つのブロックには最大 512 スレッドがまとめられている。また、このスレッドの表現を最大 3 次元まで拡張できる。

グリッドは、ブロックをさらにまとめたものである。グリッド内のブロックは、2 次元で表現される。1 グリッド内の最大のブロック数は、65535 である。

また、グリッドの上位に、デバイスといわれる仕組みがあり、それが GPU 本体のことを意味している。しかし、現在グリッドとデバイスの明確な定義がないので、「1 グリッド=1 デバイス」と考えるのが適切である。このグリッド、ブロック、スレッドの構成を図 4 で示す。

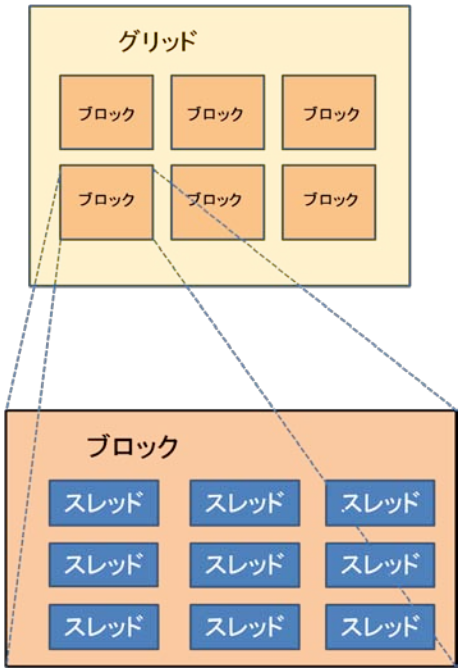

図 4　CUDA のグリッド、ブロック、スレッドの構成

また、スレッドは 32 スレッドごとに動作することになっている。この 32 スレッドを 1 つと見なすものをワープという。ここで、重要なのは「スレッドの処理は 32 ずつだと区切りがいい」ということである。32 倍数ではないスレッド数ですると、プロセッサの仕事に空きができて、GPU の処理に無駄ができてしまう。だから同時に動かすスレッド数は 32 の倍数が望ましいわけである。

(3) CUDA におけるメモリ

CUDA でプログラムを動かす上で重要な要素にメモリというものがある。一番大きい要領を占めるのがグローバルメモリである。これは、読み書きができるメモリで、容量が大きいが、GPU より遠い位置にあるため、アクセスするためにはスレッド間同期ためのコストがかかり、アクセス速度が遅いという特徴がある。

共有メモリの容量はあまり大きくないが、アクセス速度が非常に高速である。同一ブロック内の共有メモリには読み書きともに高速にアクセスすることができ、データ同期のためのコストも小さい。これは、ブロック内のスレッドはほぼ同時に進行するという仕組みによるものである。グリッド全体で参照できるメモリには、グローバルメモリ、コンスタントメモリ、テクスチャメモリがある。コンスタントメモリとテクスチャメモリはデバイスからは書き込むことができず、そのためアクセスにスレッド間同期のコストがかからな

い。ホストからデバイスへデータを渡すときには、グローバルメモリ、コンスタントメモリ、テクスチャメモリへ書き込む。ホストが直接共有メモリを操作することはできない。グローバルメモリ・コンスタントメモリ・テクスチャメモリと共有メモリのデータのやりとりは、デバイスが必要に応じて行う。また、計算結果をデバイスからホストに渡す際には、グローバルメモリを利用する。

以上のことから、十分なパフォーマンスを得るためには、

(ⅰ)CPU と GPU の間のデータ転送をできるだけ少なくする

(ⅱ)スレッドからのグローバルメモリをできるだけ少なくする

(ⅲ)ブロック分割を効率的に行い、共有メモリを効率よく利用する

これらの考慮が必要となる。[1]

CUDA のメモリのモデルを図 5 に示す。

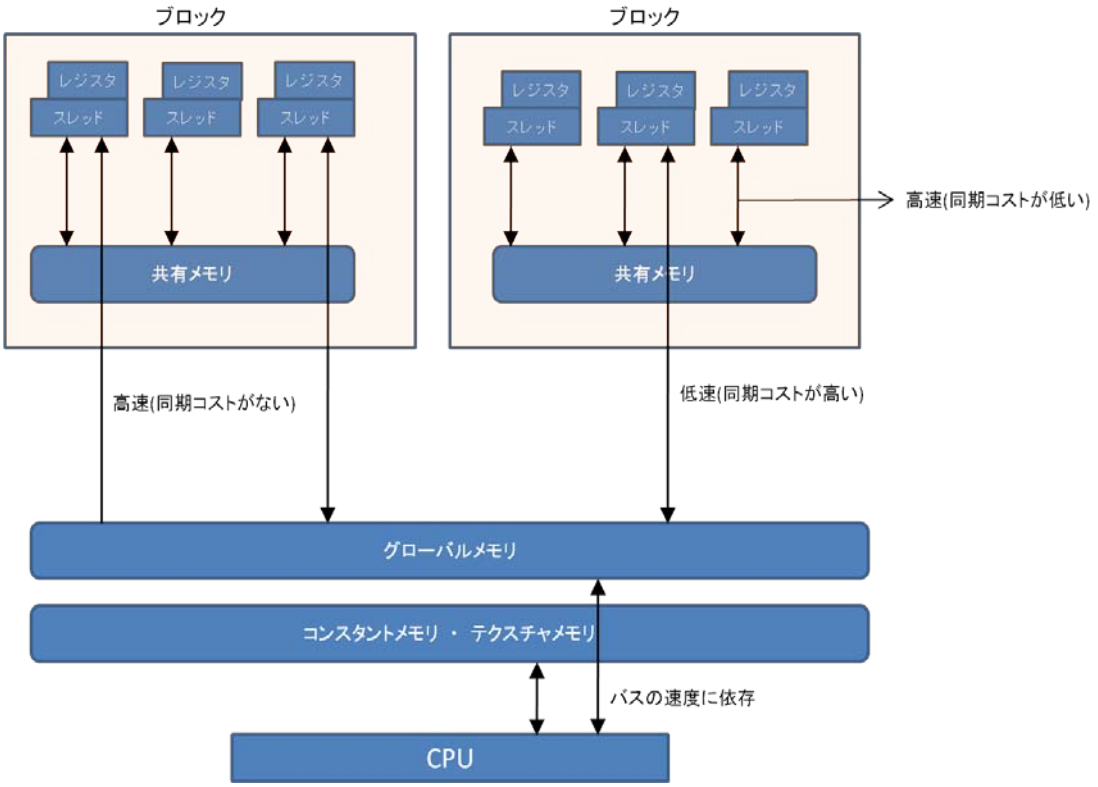

図 5　CUDA のメモリ参照モデル

3. GPU を用いた行列演算の並列化

3. 1　行列積の並列化

CUDA を利用した行列積について説明する。例として、2*2 の行列積は次の図 6 の通りになる。

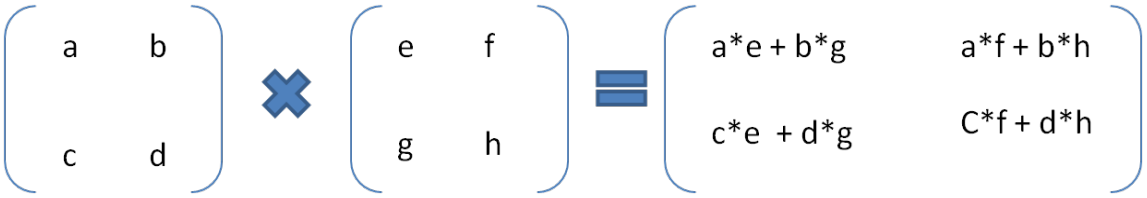

図 6　2*2 行列の乗算

行列の演算を行う時、C の対象となる要素と同じ A の行と、B の列の要素を掛け合わせる。これを一般的な行列に当てはめ、サイズ N*N の行列とすると、結果は

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \qquad (i, j = 1, 2, \cdots, n)$$

となる。

この N*N の行列演算を GPU を用いて計測する。与えられたサイズの大きさを SIZE とした正方行列 A と B に対して、積 C=A*B を計算するものとしている。

ここで示すカーネルは、シェアード・メモリは使用せずに，グローバル・メモリに格納された二つの行列データを直接計算し、その計算結果をグローバル・メモリへ書き込んでいくものである。

CPU の行列乗算のプログラムと見比べると、3 重ループであったものが GPU では 1 重ループになっていることが分かる。これは，GPU の場合、最も内側のループである行ベクトルと列ベクトルの乗算を行う処理を、各スレッドで並列実行させるためである。これを図 7 として示す。

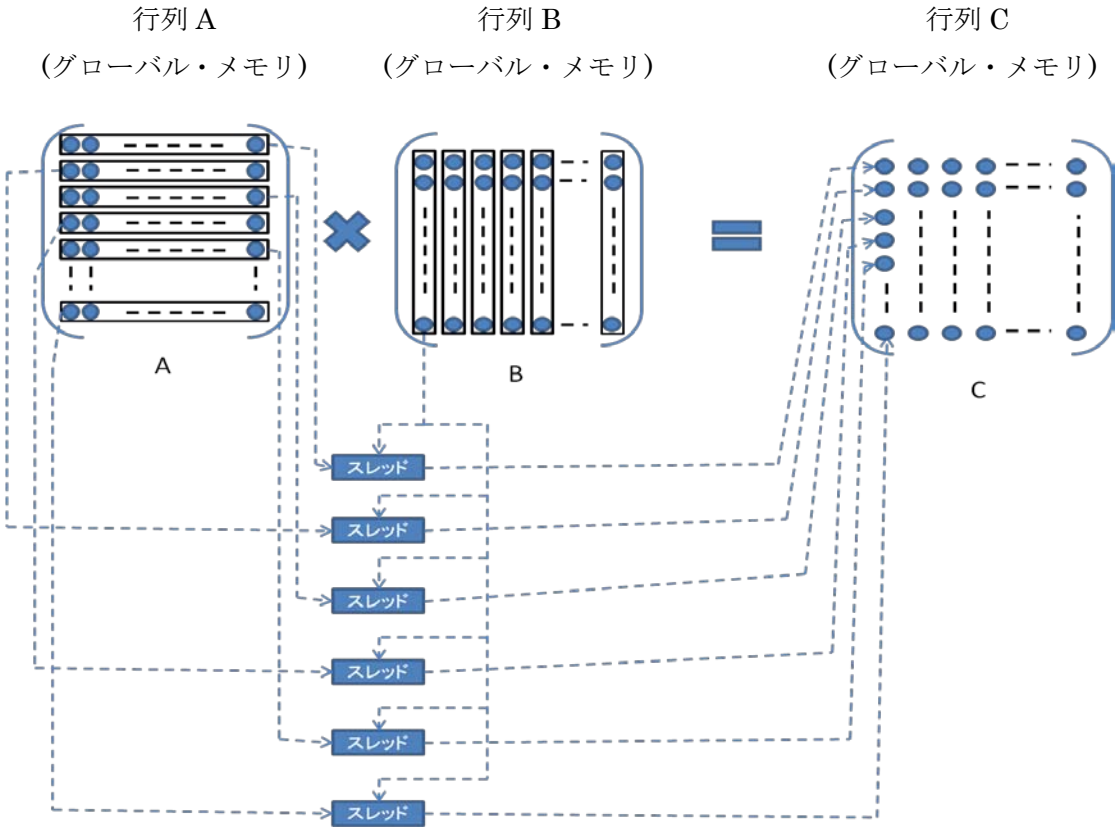

図 7　複数スレッドのよる行列積の並列化

1 つ 1 つのスレッドが行列 A の行と行列 B の列の計算をして、行列 C の結果として出力されている。行列積での計算の場合、1 つずつのスレッドで行と列の計算をしているため、行列サイズが大きくなるごとにその計算量が増えていくことが分かる。

行列 C の 1 要素を 1 スレッドで計算しているので､行列サイズ分スレッドが必要になる。その必要なスレッドはブロック内にあるスレッドの数とグリッド内にあるブロックの数で行列サイズ分のスレッド数を示している。

そのため、本研究のプログラムでは (ブロックサイズ)＊ (ブロックサイズ) としてスレッド数を計算しており、ブロックサイズの値を変えることでスレッド数を決めることができる。このスレッド数は上限があり、1 ブロックあたり 512 スレッドとなっている。そのため、それを超えないように設定しなければいけない。

スレッドで補えない部分はブロックの数を増やして補う。1 グリッド内でのブロックの数は行列サイズとブロックサイズで計算しており、｛ (行列サイズ) / (ブロックサイズ) ｝＊｛(行列サイズ) / (ブロックサイズ)｝ として計算されている。そのため、スレッド数が多くなることでブロック数は減ることになる。また、プログラムで (行列サイズ) > (ブロックサイ

ズ）としなければならない、かつ、この行列サイズで割り切れる数でのブロックサイズとしなければならない。1 グリッド内のブロック数は最大で 65535*65535 となっており、1 次元につき 65535 ブロックとなっている。このように設定することで行列サイズ分のスレッドを用意している。

本研究の実験では、このブロックサイズの変化による計測も行っている。そのため、ブロックサイズを 1、2、4、8、16 での計測を行い、その時のスレッド数は 1、4、16、64、256 となる。ブロック数はそれと行列サイズによって変化する。

また、シェアードメモリを使用する際には、シェアードメモリに行列 A、B の要素をシェアードメモリ分ずつコピーし、そこからスレッドで計算を行い、結果をグローバルメモリへ書き込むものである。

3．2　行列加算の並列化

次に CUDA を利用した行列加算について説明する。例として、2*2 の行列加算は次の図 8 ようになる。

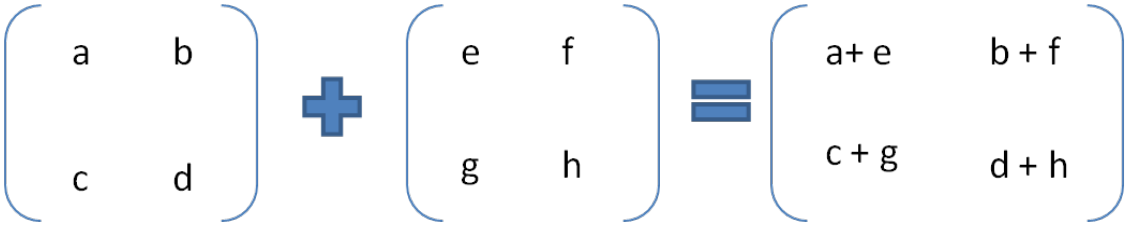

図 8　2*2 の行列の加算

これを N*N の行列加算を GPU を用いて計測を考える。行列 C の計算は行列 A の要素と行列 B の要素を足し合わせる。

各スレッドに A の行列の値と B の行列の値を読み込み、C の行列の要素に相当する計算を行う。スレッド 1 つ 1 つに加算の計算が行われている。行列積と同じようにグローバル・メモリに格納された二つの行列データを直接計算し、その計算結果をグローバル・メモリへ書き込んでいくものである。CPU の行列加算のプログラムと見比べると、2 重ループであったものが GPU ではループなしになっていることが分かる。これは，GPU の場合、行列の加算を行う処理を、各スレッドで並列実行させるためである。これを図 9 として示す。

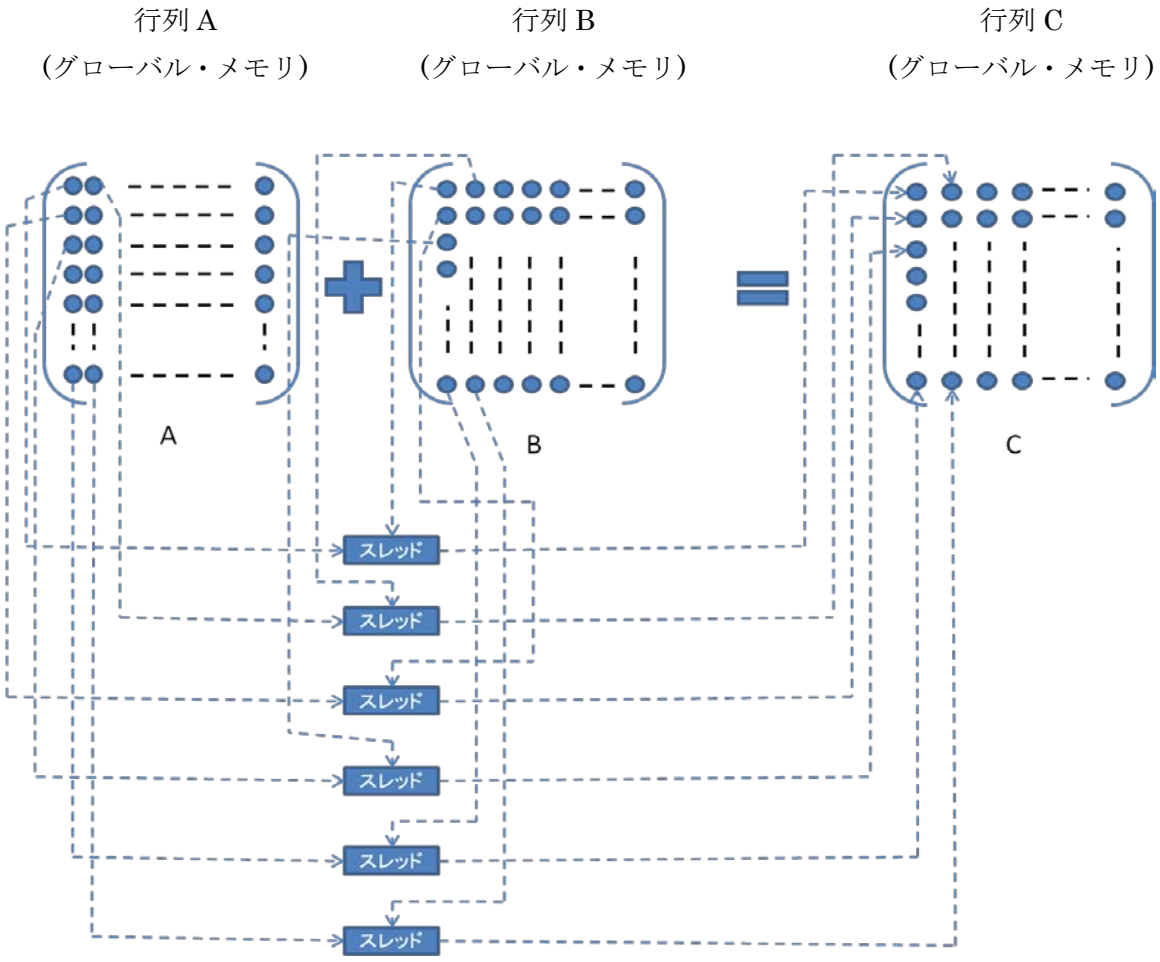

図 9　複数スレッドのよる行列加算の並列化

1 つ 1 つのスレッドが行列 A と行列 B の値の加算の計算をして、行列 C の結果として出力されている。行列加算の計算の場合、1 つずつのスレッドでの計算は行列積より少ないことが分かる。

行列加算の場合でもスレッド数とブロック数の数により行列サイズ分のスレッドを用意する。

スレッドの数は行列積同様、変化させることができ、行列加算でも（ブロックサイズ）＊

(ブロックサイズ) としてスレッド数を計算しており、ブロックサイズの値を変えることでスレッド数を決めることができる。これもスレッド数の上限があり、それを超えないように設定しなければいけない。また、ブロックの数は行列積の時と同様に、行列サイズとブロックサイズで計算しており、｛(行列サイズ) / (ブロックサイズ)｝＊｛(行列サイズ) / (ブロックサイズ)｝として計算されている。これにより行列サイズ分のスレッドを用意することができる。

本研究の実験では、行列積と同じくブロックサイズの変化による計測も行っている。そのため、ブロックサイズを 1、2、4、8、16 での計測を行っており、その時のスレッド数は 1、4、16、64、256 となる。

また、行列加算でもシェアードメモリを使用する際には、シェアードメモリに行列 A、B の要素をシェアードメモリ分ずつコピーし、そこからスレッドで計算を行い、結果をグローバルメモリへ書き込むものである。

4. 実験と考察

4．1　行列積の実験

GPU を用いた行列積の処理時間を計測した。これを CPU でも計測した行列積の処理時間と比較する。この結果を表 2 として示す。

この表 2 で使用している計測時間の単位はミリ秒で、ブロックサイズ(1 ブロック内のスレッド数と 1 グリッド内のブロック数を決める値)は１６である。この場合ブロックサイズが 16 なので、スレッド数は 256、ブロック数は 4096(64*64)となる。

本研究で使用しているマシンは CPU では Core2Duo3.16G(E8500 3.18GHz)、GPU は Tesla C1060 であり、プロセッサコア数は 240 である。内部の CPU は Core2Quad2.66G(Q9400 2.66GHz)である。表 2 の縦軸の速度向上は CPU/GPU として計算され、GPU が CPU の何倍速いかを表示している。

表 2　行列積における CPU と GPU の実行時間　　単位：ミリ秒(ms)

行列サイズ	16*16	64*64	128*128	512*512	1024*1024	2048*2048
CPU	0.054	3.37	27.84	1283.56	14222.48	460456.39
GPU	0.10	0.17	0.31	21.77	117.47	800.83
速度向上	0.54	20	90	59	121	575

結果は演算の処理が大きいほど、GPU での方が速い結果が得られた。この処理時間の速度向上をグラフとして図 10 に示す。

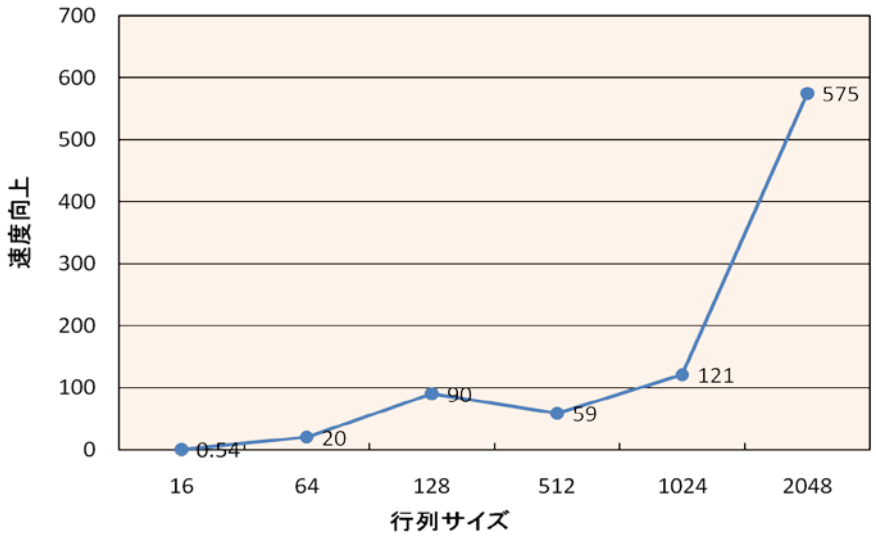

図 10　GPU の CPU に対する速度向上

これを見ると行列サイズが大きければ大きいほど、GPU での高速化を確認することができる。

次にブロックサイズ別の検証をした。ブロックサイズを変えることで 1 ブロック内にあるスレッドの数を決めることができる。スレッド数は（ブロックサイズ）＊（ブロックサイズ）としている。

これを踏まえ、GPU でブロックサイズの比較を行い、その結果を表 3 に示す。ここでのブロックサイズは 1、2、4、8、16 としており、それでのスレッド数は、1、4、16、64、256 ということである。1 スレッドが解の 1 要素のためブロック数もスレッド数によって変化する。

表 3　ブロックサイズ毎の実行時間(行列積)　　単位：ミリ秒(ms)

	行列サイズ	16*16	64*64	128*128	512*512	1024*1024
ブロックサイズ	1	0.12	0.89	6.08	442.65	
	2	0.10	0.32	1.69	174.57	1295.39
	4	0.11	0.22	0.68	83.28	442.06
	8	0.11	0.18	0.53	38.55	182.81
	16	0.10	0.17	0.31	21.77	117.47

表 3 の中のブロックサイズが 1 で行列サイズが 1024*1024 の時の空白部分は計測するとタイムアウトエラーが発生しているため、空白としている。

また、この結果を CPU での処理時間と比較してその速度向上を図 11 のグラフとして示す。

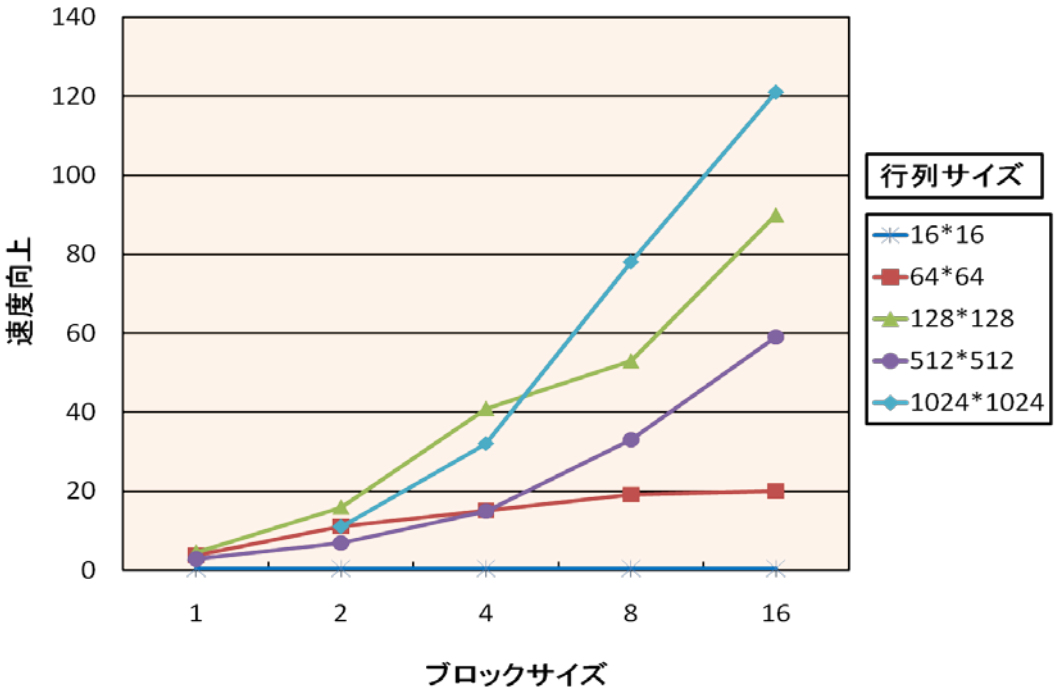

図 11　ブロックサイズ別の CPU との速度向上(行列積)

この結果からブロックサイズ 16(スレッド数は 256)の時が処理時間において一番速い結果が得られた。

次にシェアードメモリを使用してのプログラムを組んで速度向上を行った。その時の処理時間の計測結果を表 4 にして示す。GPU1 がシェアードメモリを使用していないパターン、GPU2 が使用しているパターンである。この時、表 4 で示している計測時間の単位はミリ秒で、ブロックサイズは１６である。

表 4　行列積における CPU と GPU1 と GPU2 の実行時間　　単位:ミリ秒(ms)

行列サイズ	16*16	64*64	128*128	512*512	1024*1024	2048*2048
CPU	0.054	3.37	27.84	1283.56	14222.48	460456.39
GPU1	0.10	0.17	0.31	21.77	117.47	800.83
GPU2	0.095	0.11	0.14	2.14	11.75	88.39

ここで GPU2 と GPU1 を比較するため、速度向上を表 5 に示す。表 5 の縦軸にある速度向上は、GPU1/GPU2 とし、GPU2 の GPU1 に対する速度向上を表示している。

表 5　GPU2 の GPU1 に対する速度向上　　単位：ミリ秒(ms)

行列サイズ	16*16	64*64	128*128	512*512	1024*1024	2048*2048
GPU1	0.10	0.17	0.31	21.77	117.47	800.83
GPU2	0.095	0.11	0.14	2.14	11.75	88.39
速度向上	1.05	1.55	2.21	10	10	9

結果はシェアードメモリを使用した方が処理時間において速い結果が得られた。この処理時間の速度向上をグラフとして図 12 に示す。

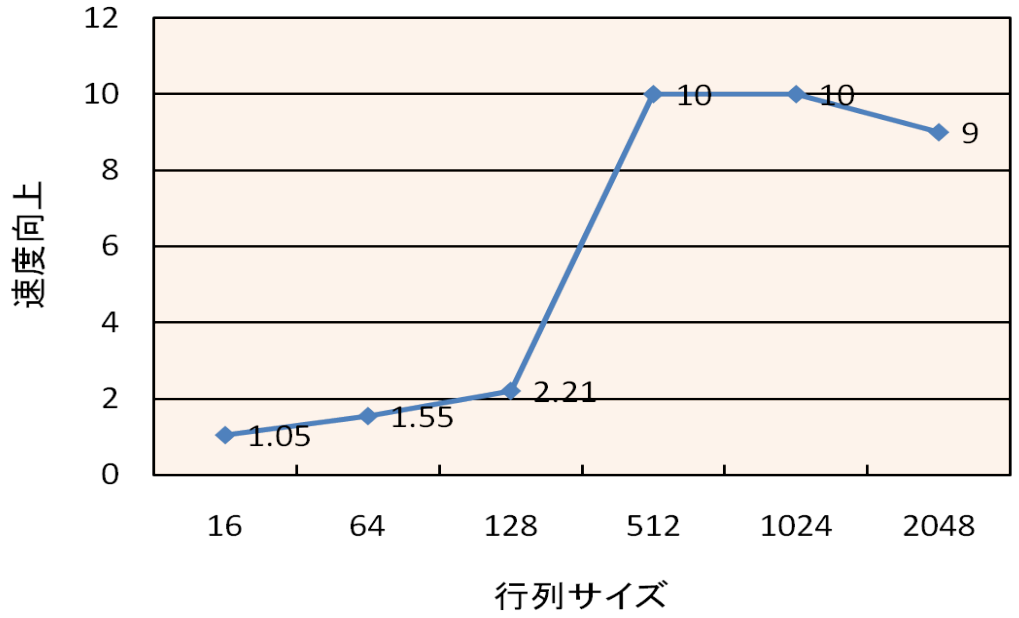

図 12　GPU2 の GPU1 に対する速度向上

その結果、シェアードメモリを使用した方が使用していない場合よりも、最大で約 10 倍の速度向上が見られた。

4．2　行列加算の実験

次に行列積と同じように GPU での行列加算での処理時間を計測した。それを CPU で計測した処理時間と比較し、その結果を表 6 として示す。

この表 6 で示している計測時間の単位はミリ秒で、ブロックサイズは１６である。

使用しているマシンは両方、行列積と同じマシンである。表 6 の縦軸にある速度向上は

行列積の時と同様に、CPU/GPU としており、GPU の CPU に対する速度向上を表示している。

表 6　行列加算における CPU と GPU の実行時間　　単位:ミリ秒(ms)

行列サイズ	16*16	64*64	128*128	512*512	1024*1024	2048*2048	3072*3072
CPU	0.003	0.046	0.18	2.90	7.36	29.63	69.31
GPU	0.10	0.11	0.11	0.17	0.34	0.78	1.63
速度向上	0.03	0.42	1.64	17	22	38	43

結果は行列加算の時と同様、行列サイズが大きいほど、処理時間が速い結果が得られた。この処理時間の速度向上をグラフとして図 13 に示す。

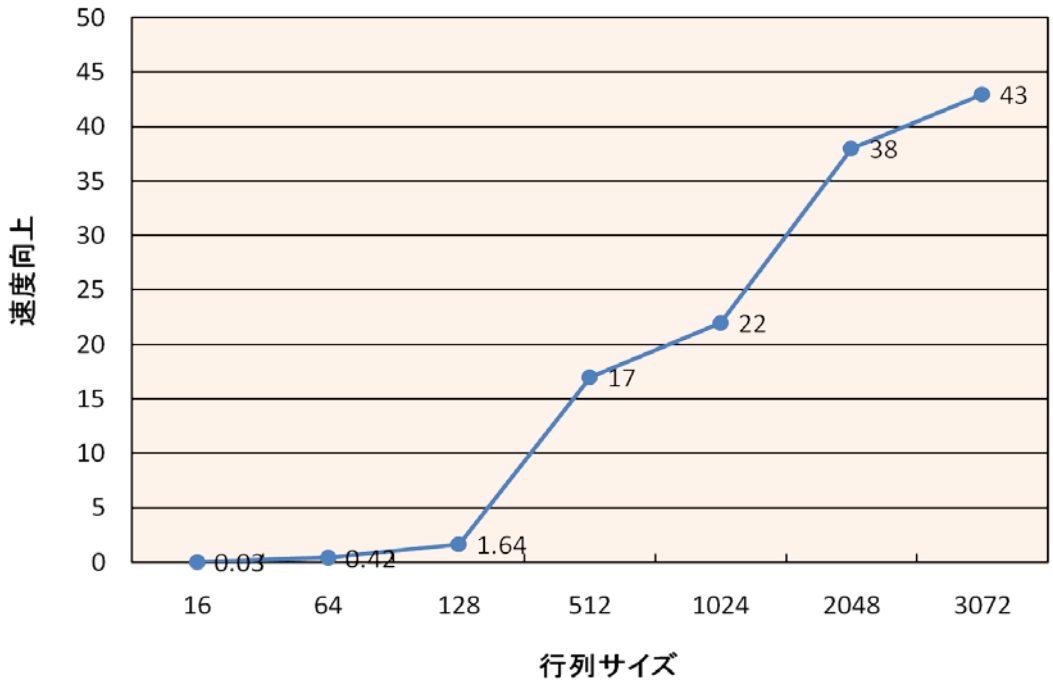

図 13　GPU の CPU に対する速度向上

次に行列積の時と同様にブロックサイズ別の検証を行った。この行列加算のプログラムでも行列積と同様にブロックサイズを変えることで 1 ブロック内にあるスレッドの数と 1 グリッド当たりのブロック数を決めることができる。ここでもスレッド数は (ブロックサイズ) * (ブロックサイズ) としている。ここのブロックサイズは 1、2、4、8、16 としており、スレッド数は、1、4、16、64、256 ということになる。こちらもブロック数はスレッド数によって変化する。この結果を表 7 として示す。

表 7　ブロックサイズ毎の実行時間(行列加算)　　単位：ミリ秒(ms)

	行列サイズ	16*16	64*64	128*128	512*512	1024*1024	2048*2048	3072*3072
ブロックサイズ	1	0.11	0.13	0.23	2.04	7.67	30.03	67.33
	2	0.11	0.11	0.15	0.62	2.06	7.71	17.11
	4	0.10	0.11	0.12	0.30	0.65	2.18	4.75
	8	0.11	0.12	0.11	0.21	0.40	1.07	2.01
	16	0.10	0.11	0.11	0.17	0.34	0.78	1.63

この結果を CPU での処理時間と比較してその速度向上を図 14 のグラフとして示す。行列サイズが小さい時、速度の向上は見られなかったため、図 14 では省き、行列サイズ 128*128 からを載せている。

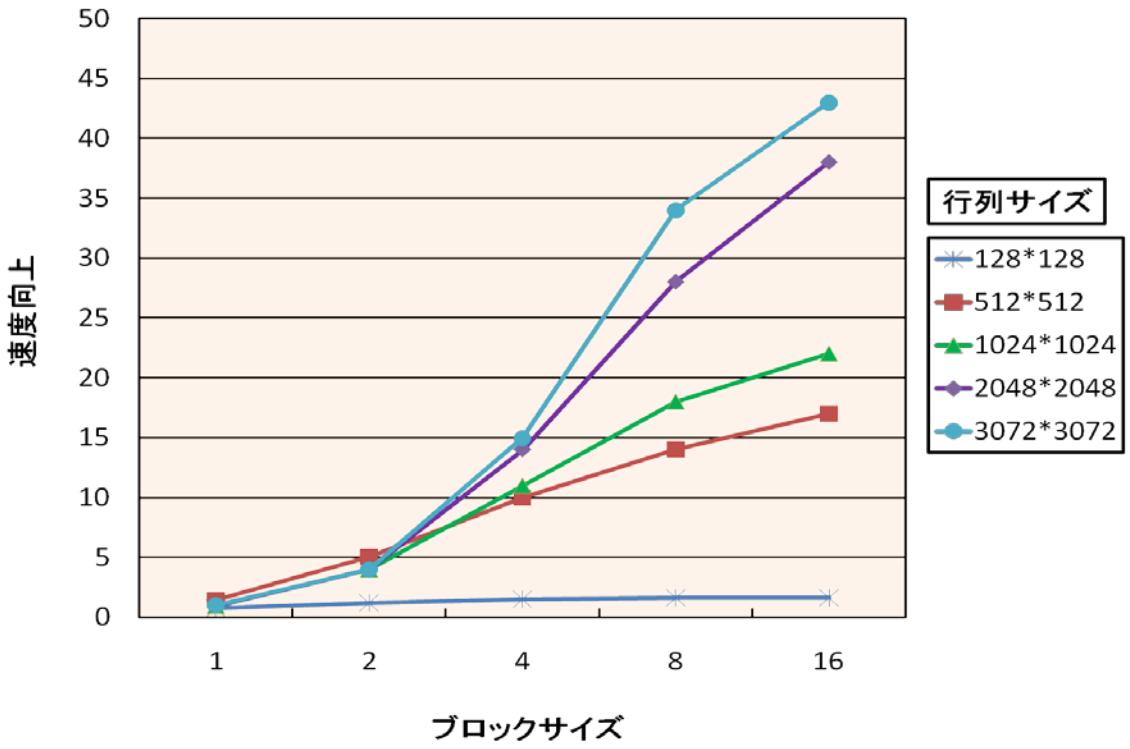

図 14　ブロックサイズ別での CPU との速度向上(行列加算)

この結果からブロックサイズ 16(スレッド数は 256)の時が処理時間おいて一番速い結果が得られた。

次にシェアードメモリを使用しての行列加算での計測を行った。その時の処理時間の計測結果を表 8 にして示す。行列積同様に GPU1 がシェアードメモリを使用していないパターン、GPU2 が使用しているパターンである。この時、表 8 で示している計測時間の単位はミリ秒で、ブロックサイズは１６である。また、GPU2 と GPU1 を比較するため、速度向上を表 9 に示す。表 9 の縦軸にある速度向上は、GPU1/GPU2 とし、GPU2 の GPU1 に対する速度向上を表示している。

表 8　行列加算における CPU と GPU1 と GPU2 の実行時間　　単位 : ミリ秒(ms)

行列サイズ	16*16	64*64	128*128	512*512	1024*1024	2048*2048
CPU	0.003	0.046	2.90	7.36	29.63	69.31
GPU1	0.10	0.11	0.11	0.17	0.34	0.78
GPU2	0.094	0.10	0.11	0.17	0.31	0.78

表 9　GPU2 の GPU1 に対する速度向上　　単位 : ミリ秒(ms)

行列サイズ	16*16	64*64	128*128	512*512	1024*1024	2048*2048
GPU1	0.10	0.11	0.11	0.17	0.34	0.78
GPU2	0.094	0.10	0.11	0.17	0.31	0.78
速度向上	1.06	1.1	1	1	1.1	1

行列加算ではシェアードメモリを使用しても処理時間は変わらなかった。これはブロックサイズ別でも同じであった。そのため、シェアードメモリを使用しても 1 つあたりの計算が小さすぎると効果がないと考えられる。

4．3　考察

実験の結果から行列積では、64*64 以降の行列サイズの場合は高速化された結果を得ることができた。しかも、行列サイズが大きくなればなるほど、速度向上が格段に上がり、非常に高いパフォーマンスを得ることができた。

行列加算の場合でも同じく行列サイズが大きければ大きいほど高速化され、高いパフォーマンスを得ることができた。行列加算の場合は、行列サイズ 512*512 からでないと速度向上が見られなかった。

行列積、加算共に行列サイズが小さい時は思ったようなパフォーマンスを得ることはできなかった。

GPU におけて行列積をシェアードメモリ(共有メモリ)を使用して計測した結果、使用し

ていない場合より、最大で約 10 倍の高速化が見られた。このため、シェアードメモリをうまく使用することでより高速化できることが考えられる。また、行列加算の場合は、処理時間があまり変わらなかった。これは、1 つあたりの計算が小さすぎると効果がでにくいことが原因だと考えられる。この時行列加算を 1 スレッドを 1 行ずつの計算として計測することを 1 つの課題と考えている。

しかし、シェアードメモリを使うことで処理が遅くなることはないと考えられるので、プログラムを組む際、シェアードメモリを使用する方が有効であると考えられる。

このように GPU のプログラムをより高速化するには、最適化を考える必要があり、今後の課題ではさらに高速化できるように最適化を検討することが課題である。

ブロックサイズ(1 ブロック内のスレッド数と 1 グリッド内のブロック数を決める値)での検証の結果は、行列積、行列加算ともにブロックサイズ 16(スレッド数 256、ブロック数 64)の時に処理速度が一番速い結果が得られた。ブロックサイズが大きいほど処理が速くなることが分かる。

また本研究では本論文に書かれていない大きな行列演算も行ったが、処理中にタイムアウトのエラーが生じる問題があった。この現象は Windows ではグラフィックドライバ上で GPU プログラムの実行時間に制限があったためであり、ゲームなどの GPU を用いるプログラムによるハングアップにより画面が写らない等の現象が起こり、ユーザがリセットボタンを押さざるを得ない状況を避けるために設定されている。今回使用しているのではだいたい 2 秒だと実験の中から判断できた。この問題の解決策はこの OS の設定を変えるか、もしくはこの制限時間内に処理を終わらす必要があることが考えられる。

これらのすべての実験結果を通して考えられることは GPU で動かす場合は、演算はできるだけ大きい方が好ましいことが伺える。また、大きければ大きいほどその性能は CPU より格段に優れた結果を得ることができると実験から判断できた。このため、今まで時間が要する計算でも GPU を利用することによって処理時間を大幅に短縮できると予想される。その上で最適化も考えることでより優れた高速化が可能だと考えられる。

5. おわりに

本研究では、GPU である CUDA を用いて行列演算の処理時間を計測し、CPU との処理時間を比較して CUDA における高速化を検証する研究を行った。行列サイズを変えての比較やブロックサイズを変えての比較を行った。この結果から CUDA は、小さい演算を行う場合は適切でないことが分かった。しかし、大きい演算になればなるほど、高い高速化の結果を得ることが分かった。今まで時間が要する計算でも CUDA を扱うことで時間を短縮できると予想される。また、スレッド数でも適する数があり、32 の倍数のスレッド数が望ましいことも分かった。

また、シェアードメモリを使用した時との比較により、シェアードメモリを使用した方がさらに高速化できることが検証された。そのため、プログラムを組む際にシェアードメモリを使用する方が高速化するにあたって好ましいことが

今後の課題では CUDA プログラムを最適化し、より速く高速化ができるようにすることである。本研究はシェアードメモリ(共有メモリ)を使用した最適化まで行ったが、それ以外にもメモリアクセスの効率化や分岐処理の効率化をすることにより、より最適化できるかを検討することが課題である。その中でプログラムによっては共有メモリを利用する時に複数のスレッドが同一アドレスにアクセスすることでアクセス先バンクでの衝突が引き起こされてしまうバンクコンフリクトやワープ内のスレッドの分岐の方向が異なる場合、同時に動作するスレッドが減ってしまい、ワープの作業量が増加してしまうワープダイバージェントの削減を目指すことでさらに速く、最適とした高速化が行えると考えている。

また､行列加算においても計算を 1 スレッド 1 要素としているところを 1 スレッド解の 1 行として計算して計測することも課題として考えている。GPU を用いて他の汎用計算等に行ってみることも課題とする。

この他にも課題として行列計算を使用したアプリケーションに GPU を使用すろことや行列演算以外の処理を GPU を用いて高速化を検証することを課題と考えている。

謝辞

本研究の機会を与えてくださり、ご指導を頂きました山崎勝弘教授に深く感謝致します。また、本研究にあたり、いろいろな面での貴重なご意見やご指導頂きました本研究室の皆様、先輩方に深く感謝致します。

参考文献

[1] 小山田耕二【監修】　岡田賢治 : CUDA 高速 GPU プログラミング入門, 秀和システム, 2010.

[2] 加藤公一：CUDA 概説，2008. dev.tyzoh.jp/trac/KatoLab/attachment/wiki/docs/cuda_report.pdf?format=raw cuda 概説

[3] CUDA Zone.　http://www.nvidia.com/object/cuda_home.html.（CUDA 開発者のコミュニティサイト）.

[4] CUDAcasts.　(http://www.nvidia.com/object/cuda_develop.html. イリノイ大学における講義の動画から入手可能）.

[5] CUDA SDK Code samples. http://www.nvidia.com/object/cuda_get.html.

[6] NVIDIA CUDA Compute Unified Device Architecture Programing Guide 2.0 beta2. http://developer.download.nvidia.com/compute/cuda/2.0-Beta2/docs/Programming_Guide_2.0beta2.pdf.

[7] 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（前編）　グラフィックス分野で磨かれた並列処理技術を汎用数値計算に応用，2008. http://www.kumikomi.net/archives/2008/06/12gpu1.php

[8] 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）　FFT を利用した光波の伝播（フレネル回折）を GPU で高速計算，2008. http://www.kumikomi.net/archives/2008/10/22gpu2.php

[9] 額田 彰 : CUDA による高速フーリエ変換，2010. http://ci.nii.ac.jp/els/110007658156.pdf?id=ART0009474154&type=pdf&lang=jp&host=cinii&order_no=&ppv_type=0&lang_sw=&no=1297073366&cp=

[10] 松井学，伊野文彦，荻原兼一 : プログラマブル GPU における LU 分解の設計と実装(GPU 応用)，2005. http://ci.nii.ac.jp/els/110002769830.pdf?id=ART0003069139&type=pdf&lang=jp&host=cinii&order_no=&ppv_type=0&lang_sw=&no=1297261703&cp=

[11] 山本鉄兵 : CUDA プログラミングの難点に関する調査，2009. http://www.cis.fukuoka-u.ac.jp/~tsato/thesises/2009/B/yamamoto.pdf

[12] 丸山直也 : GPU コンピュータ(CUDA)講習会 CUDA によるプログラミング基礎, 2009. http://gpu-computing.gsic.titech.ac.jp/sites/default/files/docs/20091028_tutorial-cuda-introduction.pdf

[13] 丸山直也 : GPU を用いた HPC:基礎と応用　CUDA の基礎その１. http://matsu-www.is.titech.ac.jp/~naoya/teaching/gpgpu-tutorial-gsic-2009-cuda-intro-part-1.pdf

[14] 丸山直也：GPU を用いた HPC:基礎と応用　CUDA の基礎その２.
http://matsu-www.is.titech.ac.jp/~naoya/teaching/gpgpu-tutorial-gsic-2009-cuda-intro-part-2.pdf
[15] NVIDIA サイト：NVIDIA Tesla C1060 コンピューティングプロセッサ - ワークステーション向けメニーコアスーパーコンピューティング.
http://www.nvidia.co.jp/object/tesla_c1060_jp.html
[16] CUDA プログラミング.　http://www8.plala.or.jp/b4zabeat/cuda/index.html
[17] CUDA - PikiWiki Plus.　http://imd.naist.jp/~fujis/cgi-bin/wiki/index.php?CUDA
[18] NIDIA：CUDA テクニカルトレーニング　vol. I CUDA プログラム入門，2008.
http://www.nvidia.co.jp/docs/IO/59373/VolumeI.pdf
[19] NIDIA：CUDA テクニカルトレーニング　vol. II CUDA プログラム入門，2008.
http://www.nvidia.co.jp/docs/IO/59373/VolumeII.pdf

付録

(1) CPU における行列積のソースコード

```
#include<stdio.h>
#include<malloc.h>
#include<stdlib.h>

#include<windows.h>

#define  SIZE 1024

        LARGE_INTEGER start_pc, end_pc, freq_pc;
        double sec_pc;

int
main(int argc, char** argv)
{
        unsigned int       x, y;
        int *matrixA;
        int *matrixB;
        float *matrixC;

        unsigned int i, j;
        int a1, b1;
        float c1;
        int num = 0;

        i, j = 0;
        a1 = b1 = 0;
        c1 = 0;

        matrixA = (int*)malloc(sizeof(int) * SIZE * SIZE);
        matrixB = (int*)malloc(sizeof(int) * SIZE * SIZE);
        matrixC = (float*)malloc(sizeof(float) * SIZE * SIZE);
```

```
	for(y = 0; y < SIZE; y++){
		for(x = 0; x < SIZE; x++){
			matrixA[y * SIZE + x] = rand() % 50;
			matrixB[y * SIZE + x] = rand() % 50;
			matrixC[y * SIZE + x] = 0;
		}
	}

	QueryPerformanceFrequency(&freq_pc);
	QueryPerformanceCounter(&start_pc);

	for(y = 0; y < SIZE; y++){
		for(x = 0; x < SIZE; x++){
			for(i = 0; i < SIZE; i++){
				a1 = (unsigned int )matrixA[SIZE * y + i];
				b1 = (unsigned int )matrixB[SIZE * i + x];
				c1 += a1 * b1;
				(float )matrixC[y * SIZE + x] = c1;
			}
			c1=0;
		}
	}

	QueryPerformanceCounter(&end_pc);
	sec_pc = (end_pc.QuadPart - start_pc.QuadPart) / (double)freq_pc.QuadPart;
	printf("times = %lf seconds¥n", sec_pc);

	free(matrixA);
	free(matrixB);
	free(matrixC);

	return (0);
}
```

(2) GPU における行列積のソースコード

```
#include<stdio.h>
#include<malloc.h>
#include<stdlib.h>

#include<cutil_inline.h>

#define  SIZE     1024
#define  BLOCK_SIZE     16

__global__ void
matrixMul(int* A, int* B, float* C);


int
main(int argc, char** argv)
{

        unsigned int size = sizeof(int) * SIZE * SIZE;
        unsigned int Size = sizeof(float) * SIZE * SIZE;



        int*       hA;
        int*       hB;
        float*     hC;


        hA = (int*)malloc(size);
        hB = (int*)malloc(size);
        hC = (float*)malloc(Size);


        unsigned int       x, y;
        for (y = 0; y < SIZE; y++) {
```

```
            for (x = 0; x < SIZE; x++) {
                        hA[y * SIZE + x] = rand() % 50;
                        hB[y * SIZE + x] = rand() % 50;
            }
}


int*        dA;
int*        dB;
float*      dC;

/* デバイス側のメモリの確保 */
cutilSafeCall(cudaMalloc((void**)&dA, size));
cutilSafeCall(cudaMalloc((void**)&dB, size));
cutilSafeCall(cudaMalloc((void**)&dC, Size));

/* ホスト側からデバイス側への転送 */
cutilSafeCall(cudaMemcpy(dA, hA, size, cudaMemcpyHostToDevice));
cutilSafeCall(cudaMemcpy(dB, hB, size, cudaMemcpyHostToDevice));
cutilSafeCall(cudaMemcpy(dC, hC, Size, cudaMemcpyHostToDevice));

/* ブロックサイズとグリッドサイズの設定 */
dim3      block(BLOCK_SIZE, BLOCK_SIZE);
dim3      grid(SIZE/BLOCK_SIZE, SIZE/BLOCK_SIZE);

/* タイマーの設定と開始 */
unsigned int        cudaTimer=0;
cutilCheckError(cutCreateTimer(&cudaTimer));
cutilCheckError(cutStartTimer(cudaTimer));

/* カーネルの起動 */
matrixMul<<<grid, block>>>(dA, dB, dC);

/* スレッドの同期 */
cudaThreadSynchronize();
```

```
	/* タイマーを止める */
	cutilCheckError(cutStopTimer(cudaTimer));
	printf("time = %lf ms ¥n", cutGetTimerValue(cudaTimer));
	cutilCheckError(cutDeleteTimer(cudaTimer));

	/* デバイス側からホスト側へのメモリ転送 */
	cutilSafeCall(cudaMemcpy(hC, dC, Size, cudaMemcpyDeviceToHost));

	/*デバイス側のメモリ開放*/
	cutilSafeCall(cudaFree(dA));
	cutilSafeCall(cudaFree(dB));
	cutilSafeCall(cudaFree(dC));


	free(hA);
	free(hB);
	free(hC);

	/* 終了処理 */
	cudaThreadExit();
	cutilExit(argc, argv);
}


__global__
	void matrixMul(int* A, int* B, float* C)
	{
		unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
		unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;
		unsigned int i;
		float c1 =0;

		for(i = 0; i < SIZE; i++){
			c1 += A[SIZE * y + i] * B[SIZE * i + x];
	__syncthreads();
		}
```

```
                C[y * SIZE + x] = c1;
        }
```

(3) CPU における行列加算のソースコード

```
#include<stdio.h>
#include<malloc.h>
#include<stdlib.h>

#include<windows.h>

#define  SIZE     1024

        LARGE_INTEGER start_pc, end_pc, freq_pc;
        double sec_pc;

int
main(int argc, char** argv)
{
        unsigned int       x, y;
        int *matrixA;
        int *matrixB;
        int *matrixC;

        int a1, b1;
        int c1;
        int num =0;

        a1 = b1 = 0;
        c1 = 0;

        matrixA = (int*)malloc(sizeof(int) * SIZE * SIZE);
        matrixB = (int*)malloc(sizeof(int) * SIZE * SIZE);
        matrixC = (int*)malloc(sizeof(int) * SIZE * SIZE);
```

```
	for(y = 0; y < SIZE; y++){
		for(x = 0; x < SIZE; x++){
			matrixA[y * SIZE + x] = rand() % 500;
			matrixB[y * SIZE + x] = rand() % 500;
			matrixC[y * SIZE + x] = 0;

		}
	}
	QueryPerformanceFrequency(&freq_pc);
	QueryPerformanceCounter(&start_pc);


	for(y = 0; y < SIZE; y++){
		for(x = 0; x < SIZE; x++){
				a1 = (unsigned int )matrixA[SIZE * y + x];
				b1 = (unsigned int )matrixB[SIZE * y + x];
				c1 = a1 + b1;
				(unsigned int )matrixC[y * SIZE + x] = c1;
				c1=0;
		}
	}

	QueryPerformanceCounter(&end_pc);
	sec_pc = (double)(end_pc.QuadPart - start_pc.QuadPart) /
(double)freq_pc.QuadPart;
	printf("times = %lf seconds¥n", sec_pc);

	free(matrixA);
	free(matrixB);
	free(matrixC);

	return (0);
}
```

(4) GPU における行列加算のソースコード

```
#include<stdio.h>
#include<malloc.h>
#include<stdlib.h>

#include<cutil_inline.h>

#define  SIZE     1024
#define  BLOCK_SIZE     16

__global__ void
matrixAdd(int* A, int* B, int* C);

int
main(int argc, char** argv)
{

        unsigned int size = sizeof(unsigned int) * SIZE * SIZE;


        int*       hA;
        int*       hB;
        int*       hC;


        hA = (int*)malloc(size);
        hB = (int*)malloc(size);
        hC = (int*)malloc(size);


        unsigned int       x, y;
        for (y = 0; y < SIZE; y++) {
                for (x = 0; x < SIZE; x++) {
                        hA[y * SIZE + x] = rand() % 500;
```

```
                hB[y * SIZE + x] = rand() % 500;
        }
}


int*    dA;
int*    dB;
int*    dC;

/* デバイス側のメモリの確保 */
cutilSafeCall(cudaMalloc((void**)&dA, size));
cutilSafeCall(cudaMalloc((void**)&dB, size));
cutilSafeCall(cudaMalloc((void**)&dC, size));

/* ホスト側からデバイス側への転送 */
cutilSafeCall(cudaMemcpy(dA, hA, size, cudaMemcpyHostToDevice));
cutilSafeCall(cudaMemcpy(dB, hB, size, cudaMemcpyHostToDevice));
cutilSafeCall(cudaMemcpy(dC, hC, size, cudaMemcpyHostToDevice));

/* ブロックサイズとグリッドサイズの設定 */
dim3    block(BLOCK_SIZE, BLOCK_SIZE);
dim3    grid(SIZE/BLOCK_SIZE, SIZE/BLOCK_SIZE);

/* タイマーの設定と開始 */
unsigned int      cudaTimer=0;
cutilCheckError(cutCreateTimer(&cudaTimer));
cutilCheckError(cutStartTimer(cudaTimer));

/* カーネルの起動 */
matrixAdd<<<grid, block>>>(dA, dB, dC);

/* スレッドの同期 */
cudaThreadSynchronize();

/* タイマーを止める */
cutilCheckError(cutStopTimer(cudaTimer));
```

```
	printf("time = %lf ms ¥n", cutGetTimerValue(cudaTimer));
	cutilCheckError(cutDeleteTimer(cudaTimer));

	/* デバイス側からホスト側へのメモリ転送 */
	cutilSafeCall(cudaMemcpy(hC, dC, size, cudaMemcpyDeviceToHost));

	/*デバイス側のメモリ開放*/
	cutilSafeCall(cudaFree(dA));
	cutilSafeCall(cudaFree(dB));
	cutilSafeCall(cudaFree(dC));


	free(hA);
	free(hB);
	free(hC);

	/* 終了処理 */
	cudaThreadExit();
	cutilExit(argc, argv);
}


__global__
	void matrixAdd(int* A, int* B, int* C)
	{
		unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
		unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;
		unsigned int c1 =0;

		if(y < SIZE && x < SIZE){
			c1 = A[SIZE * y + x] + B[SIZE * y + x];
	__syncthreads();
		}
		C[y * SIZE + x] = c1;
	}
```

(5) GPU におけるシェアードメモリ使用した行列積のソースコード

```
#include<stdio.h>
#include<malloc.h>
#include<stdlib.h>

#include<cutil_inline.h>

#define  SIZE     1024
#define  BLOCK_SIZE     16


__global__ void
matrixMul(int* A, int* B, float* C);

int
main(int argc, char** argv)
{
        /* 行列のサイズをバイト単位で算出 */
        unsigned int size = sizeof(unsigned int) * SIZE * SIZE;
        unsigned int Size = sizeof(float) * SIZE * SIZE;

        /* 変数を宣言 */
        int*       hA;
        int*       hB;
        float*     hC;

        /* ホスト側のメモリの確保 */
        hA = (int*)malloc(size);
        hB = (int*)malloc(size);
        hC = (float*)malloc(Size);

        /* 初期値を設定 */
        unsigned int       x, y;
        for (y = 0; y < SIZE; y++) {
                for (x = 0; x < SIZE; x++) {
```

```
                hA[y * SIZE + x] = rand() % 50;
                hB[y * SIZE + x] = rand() % 50;
        }
}

/* デバイス側の変数の設定 */
int*    dA;
int*    dB;
float*  dC;

/* デバイス側のメモリの確保 */
cutilSafeCall(cudaMalloc((void**)&dA, size));
cutilSafeCall(cudaMalloc((void**)&dB, size));
cutilSafeCall(cudaMalloc((void**)&dC, Size));

/* ホスト側からデバイス側への転送 */
cutilSafeCall(cudaMemcpy(dA, hA, size, cudaMemcpyHostToDevice));
cutilSafeCall(cudaMemcpy(dB, hB, size, cudaMemcpyHostToDevice));
cutilSafeCall(cudaMemcpy(dC, hC, Size, cudaMemcpyHostToDevice));

/* ブロックサイズとグリッドサイズの設定 */
dim3    block(BLOCK_SIZE, BLOCK_SIZE);
dim3    grid(SIZE/BLOCK_SIZE, SIZE/BLOCK_SIZE);

/* タイマーの設定と開始 */
unsigned int      cudaTimer=0;
cutilCheckError(cutCreateTimer(&cudaTimer));
cutilCheckError(cutStartTimer(cudaTimer));

/* カーネルの起動 */
matrixMul<<<grid, block>>>(dA, dB, dC);

/* スレッドの同期 */
cudaThreadSynchronize();

/* タイマーを止める */
```

```
        cutilCheckError(cutStopTimer(cudaTimer));
        printf("time = %lf ms ¥n", cutGetTimerValue(cudaTimer));
        cutilCheckError(cutDeleteTimer(cudaTimer));

        /* デバイス側からホスト側へのメモリ転送 */
        cutilSafeCall(cudaMemcpy(hC, dC, Size, cudaMemcpyDeviceToHost));


        /*デバイス側のメモリ開放*/
        cutilSafeCall(cudaFree(dA));
        cutilSafeCall(cudaFree(dB));
        cutilSafeCall(cudaFree(dC));

        /* ホスト側のメモリ開放 */
        free(hA);
        free(hB);
        free(hC);

        /* 終了処理 */
        cudaThreadExit();
        cutilExit(argc, argv);
}


__global__
void matrixMul(int* A, int* B, float* C)
{
                int bx = blockIdx.x;
                int by = blockIdx.y;
                int tx = threadIdx.x;
                int ty = threadIdx.y;
                float c1 = 0;

                __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
                __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
```

```
                for (int a = 0, b = 0 ; a < SIZE; a += BLOCK_SIZE, b += BLOCK_SIZE)
{

                int a_ad = SIZE * BLOCK_SIZE * by + a;
                int b_ad = BLOCK_SIZE * bx + SIZE * b;

                As[ty][tx] = A[a_ad + SIZE *ty + tx];
                Bs[ty][tx] = B[b_ad + SIZE *ty + tx];
                __syncthreads();

                for (int i = 0; i < BLOCK_SIZE; i++) {
                c1 += As[ty][i] * Bs[i][tx];
                }

        __syncthreads();
        }

    int ad = SIZE * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[ad + SIZE * ty + tx] = c1;
}
```