修 士 論 文 の 和 文 要 旨

研究科・専攻	大学院 情報理工 学研究科学研究科 情報・通信工学 専攻 博士前期課程		
氏　　名	田村 遼也	学籍番号	1131075
論 文 題 目	GPU への完全オフロード化による TSQR の高速化に関する研究		

要　旨

QR 分解は様々な分野に利用される行列分解計算の一つであり、その中でも行数 m>>列数 n となる縦長行列に対する QR 分解はベクトルの直交式に相当し、様々な数値計算手法で高速でありなおかつ高精度な方法が求められている。従来法のハウスホルダーQR は処理が逐次的で同期や通信多く、並列化が難しい。近年提唱された並列処理に適した Tall Skinny QR(TSQR)アルゴリズムでは縦長行列を行方向に分割した小行列を QR 分解し、得られた上三角行列 R を結合し、QR 分解を繰り返す階層的な構造をしている。

$$A = \begin{bmatrix} A_0 \\ A_1 \end{bmatrix} = \begin{bmatrix} Q_0 R_0 \\ Q_1 R_1 \end{bmatrix} \rightarrow \begin{bmatrix} R_0 \\ R_1 \end{bmatrix} = Q_2 R_2$$

本研究では Graphics Processing Unit (GPU)に着目し NVIDIA 社の CUDA を用いた TSQR の実装を行う。CUDA ではカーネル関数として実装した部分を GPU で並列処理することができる。TSQR では自明な並列性のある複数の行列に対する QR 分解をカーネル関数とするべきである。ここで、CUDA ブロック単位で 1 つの QR 分解を処理することが理想的であるが、現在の CUDA の実行モデルでは 2 分木等の葉の実行完了をブロック間で認識できないので、一旦ホストに戻し、システムレベルで大きな同期を行う必要がある。TSQR の第二階層以降で扱う行列はとても小さいものとなるので、データの移動は少ないものの、実行時間全体に占める同期のコストは大きくなることが問題となる。本研究では、コストが大きい集約操作を 2 分木型ではなく 1 度に全て集約する形とし、同期の回数を最小限に抑える実装を行う。

本実装と CUDA 向けの線形代数ライブラリである MAGMA で QR 分解実行時間の比較実験を行った。MAGMA はハウスホルダーQR を使用している。結果、行数が少ない場合では、MAGMA に大差をつけられるが、1 ブロックあたりで処理するベクトルの長さが十分大きくなる行数領域では本実装の方が高速であった。最適化された実装である MAGMA に対し、最適化が十分でない本実装でも高速である場合が確認でき、最適化を進めることで本実装の優位性が高まることが望める。

さらに、2 分木型でホスト経由の同期を行わない 1 カーネル版の TSQR は理想的な実装であるが、ブロック間同期等の新技術によりこれが実現可能であるとの知見を得ており、今後はその検証にすすむことが今後の課題となる。

平成 25 年度　修士論文
GPU への完全オフロード化による TSQR の
高速化に関する研究

電気通信大学　　情報理工学研究科

1131075　　田村　遼也

指導教員　仲谷　栄伸　教授

副指導教員　龍野　智哉　准教授

平成 26 年 3 月 7 日

目次

第1章. 序章

1.1 研究の背景と目的

QR 分解はベクトルの直交化と同等のアルゴリズムであり、固有値解析など様々な分野に利用される行列分解計算の一つである。とくに縦長(m≫n)の QR 分解は特異値分解やブロックハウスホルダー法、部分空間反復解法の中で重要な役割を果たしており、大きな行列に対して高速でありなおかつ高精度な方法が求められている。QR 分解で多く用いられているハウスホルダーQR 分解は逐次的であり並列化が行いにくいため、近年提唱された Tall Skinny QR(TSQR)アルゴリズムが注目されており TSQR に関する研究は重要性が高いと考えられる。

TSQR についてはこれまでの研究により従来のハウスホルダーQR と比較して以下のことがわかっている。

・並列性が高い[1]

・計算量が増加する[2]

・計算精度が良い[2]

また、TSQR は再帰的に利用可能で再帰数を増やすほど記憶参照の局所性が増すことや、行列が縦長であればあるほど性能が高くなる[3]ことも知られている。

本研究では TSQR の並列化の方法として Graphics Processing Unit (GPU)を利用した並列化に着目している。GPU は画像処理用のハードウェアで主に 3D ゲームなどをモニターに出力することを行っていた。GPU は単純な計算能力だけでは CPU よりも高く、この GPU を画像処理以外の汎用計算にも利用しようという考えが General Purpose computation on GPU (GPGPU)である。本研究では NVIDIA 社より提供されている GPGPU 用開発環境である CUDA を用いて TSQR を実装する。また、近い将来、アクセラレータのみで全てのタスクを処理する時代が来ることを考えると、GPU への完全オフロード化は重要なテーマである。そこで、本研究では CPU 資源による演算を行わない GPU への完全オフロード化による TSQR の高速化を目的とする。本研究では、通常 C で使用される Row-Major ではなく CUDA の特性を生かすことができる Fortran 等で用いられる Column-Major での実装や BLAS などに代表される定型処理のデバイス関数での実装をする。完全オフロード化の際、ボトルネックとなりやすい CUDA ブロック間の同期が問題となるので本研究では同期回数を 1 回のみとするように実装をする。

1.2 本論文の構成

本論文は以下のように構成される。第 2 章ではハウスホルダーQR 分解や TSQR の概念、アルゴリズムについて記述している。第 3 章では本実験で扱う画像処理用ハードウェア GPU と TSQR の GPU での実装について、第 4 章は実験、評価、第 5 章では結論を述べている。

第2章. TSQR

2.1 QR 分解

大きさが m×n (m ≥ n)の行列 A を直交行列 Q と上三角行列 R を使って、式(2.1)のような

$$A \rightarrow QR \tag{2.1}$$

に分解することを QR 分解と呼ぶ。[4]QR 分解には大きさが m×m の Q を求める fatQR タイプと m×n の Q と n×n の R のみを求める thinQR の 2 通りが存在するが、本論文では主に後者のタイプを扱うものとする。QR 分解は特異値分解やブロックハウスホルダー法、部分空間反復解法の中で重要な役割を果たしている。QR 分解の計算方法はグラムシュミット(Gram-schmidt)法やヤコビ(Jacobi)法、ハウスホルダー(Householder)法を利用したものなど、様々な方法があるが、数値計算では計算精度の問題からハウスホルダーQR 分解が用いられることが多い。

2.2 ハウスホルダーQR 分解

ハウスホルダーQR は、式(2.2)のような

$$M = I - 2uu^t \tag{2.2}$$

という形をした直交行列を用いた片側変換として構成される。ここで M は直交行列、I は単位行列、u は行列 A のある一列の要素を使って生成されるベクトルである。第 k 列目のベクトル u は以下のようにして生成する。ここで、$a_{i,j}$ は行列 A の i 行目 j 列目の要素である。

$$s = \sqrt{a_{k+1,k}^2 + a_{k+2,k}^2 + \cdots + a_{n,k}^2}$$

$$\boldsymbol{a} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ a_{k+1,k} \\ a_{k+2,k} \\ \vdots \\ a_{n,k} \end{bmatrix} \quad \boldsymbol{b} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ s \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

$$v = \boldsymbol{a} - \boldsymbol{b} \quad c = \frac{1}{|\boldsymbol{v}|} \quad \boldsymbol{u} = c\boldsymbol{v}$$

行列 M を行列 A の左側から掛けると u の生成に利用した列の特定の要素が 0 になる。

$$A \leftarrow MA$$

これを繰り返していくと行列 A は左から要素が 0 の列が増えていき、最終的に上三角行列となるので、これを行列 R とすると

$$M_n M_{n-1} M_{n-2} \ldots M_2 M_1 A = R$$

と書くことができる。ここで用いた$M_n M_{n-1} M_{n-2} \ldots M_2 M_1$を

$$Q = {M_1}^t {M_2}^t \ldots {M_n}^t$$

とまとめて Q とすればA = QRとなる。

ハウスホルダーQRで得られるQは直交性が高いが、第i列目に対する変換行列M_iを生成するためにはM1、M2、…、Mi-1まで変換行列を第i列目に作用させた後でないとできないので非常に逐次性が強い方法である。

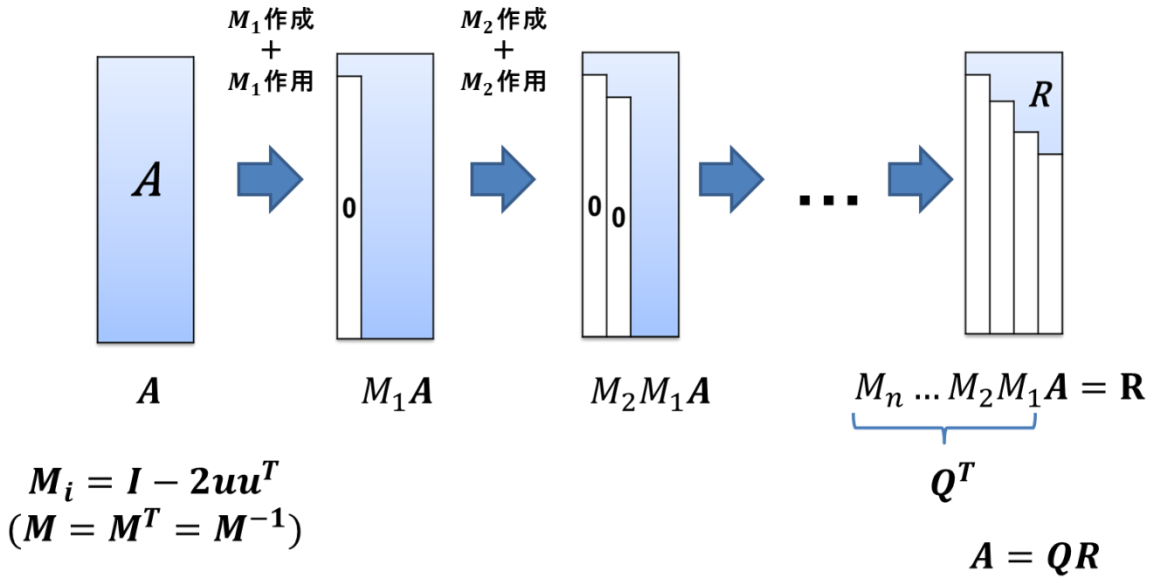


図 2-1 ハウスホルダーの流れ

ここで、Aの更新を

$$A_{k+1} \leftarrow M_k A_k = (I - 2u_k {u_k}^t)A_k = A_k - u_k\left(2({u_k}^t A_k)\right)$$

とし、Aを更新しながらベクトルu_kを並べた行列Uを

$$U \leftarrow [u_1|u_2|u_3|\cdots|u_{n-1}|u_n]$$

同時に生成する。Aの更新が全て終了した後に、

$$Q_n = \begin{bmatrix} I \\ - \\ 0 \end{bmatrix}$$

とした初期行列Q_nに以下のように、Uの一列を用いて

$$Q_{k-1} \leftarrow M_k Q_k = (I - 2u_k {u_k}^t)Q_k = Q_k - u_k\left(2({u_k}^t Q_k)\right)$$

とすれば、行列Mを作らなくてもAとQの更新を行うことができる。また
ハウスホルダーQR分解のアルゴリズムは図 2-2のようになる。図 2-2の(4)、(5)、(6)がAの更新部分、(8)、(9)、(10)がQの更新部分となる。

ハウスホルダーQR のアルゴリズム

for　k = 1 to n {

(1) $s = \sqrt{a_{k+1,k}^2 + a_{k+2,k}^2 + \cdots + a_{n,k}^2}$

(2) $u \leftarrow a - b$

(3) $|u|^2 \leftarrow |a-b|^2 = \sqrt{2s(s - a_{k,k})}$

(4) $w \leftarrow u^t A$

(5) $v \leftarrow \frac{2}{|u|^2} \times w$

(6) $A \leftarrow A - uv$

(7) $U \leftarrow \left[U \left| \frac{u}{|u|} \right.\right]$

}

for　k = n to 1 {

(8) $w \leftarrow {U_k}^t Q$

(9) $v \leftarrow 2w$

(10) $A \leftarrow Q - {U_k}^t v$

}

U_k：行列 U の k 列目

図 2-2 ハウスホルダーQR のアルゴリズム

2.3 ブロックハウスホルダーQR

図 2-2 のアルゴリズムからも分かるようにハウスホルダー変換を行列に作用させる計算は、行列とベクトルの積が中心となるため、メモリアクセスのボトルネックによる影響がとても大きく、計算機の性能を引き出すことが難しくなる。この問題を解決するため、ハウスホルダーQR を実装する場合には、一般的にブロック化と呼ばれる方法を利用する。ブロック化とは行列を列方向に分割し、分割したブロック内でのみ QR 分解を行い、得られたハウスホルダー変換行列を分解した行列以外の部分に作用することを繰り返すことである。図 2-3 にブロックハウスホルダーQR の流れを示す。

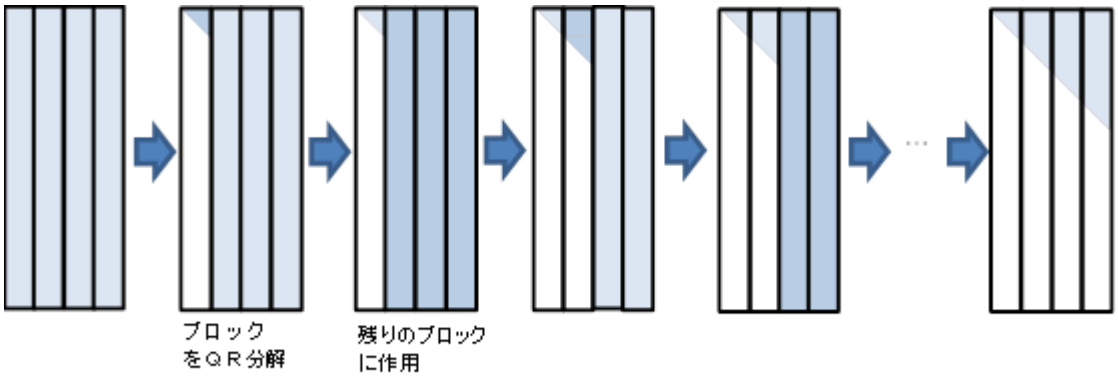


図 2-3 ブロックハウスホルダーQR の流れ

LAPACK などでは WY 表現[5]などの手法で Q の逐次させる部分を高性能化する方法がとられている。WY 表現ではハウスホルダーQR の Q を行列 Y,T を用いて、

$$Q = I - YTY^T \quad \in R^{m\times m}$$

$$\mathrm{Y} \in R^{m\times j}\ (m > j)\ ,\quad T \in R^{j\times j}$$

とする。ここで、行列 Y はハウスホルダーベクトル u を並べた行列、行列 T は上三角行列である。

ハウスホルダー変換行列を式 2.2 とすると、Q の更新は以下のようになる。$Y_0 = u_0$、$T_0 = 2$とする。

$$Q_+ = I - Y_+ T_+ Y_+^T$$

$$Y_+ = [Y \quad u] \in R^{m\times(j+1)}$$

$$T_+ = \begin{bmatrix} T & z \\ 0 & 2 \end{bmatrix} \qquad (2.3)$$

$$z = -2TY^T u \qquad (2.4)$$

図 2-4 中の nb はブロックのサイズを意味する。

$$Q_+ = I - Y_+ T_+ Y_+^T$$

$$Y_0 = u_0\ ,\ Y_+ = [Y \quad u] \in R^{m\times(j+1)}$$

$$T_0 = 2\ ,\qquad T_+ = \begin{bmatrix} T & z \\ 0 & 2 \end{bmatrix}$$

$$z = -2TY^T u$$

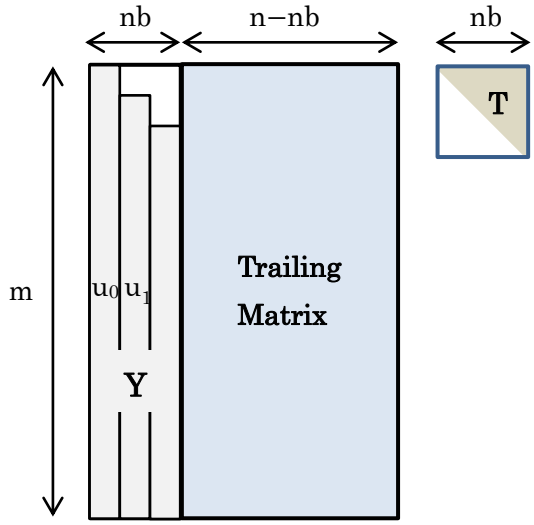


図 2-4 Q の生成

残りのブロック部分を A'とすると A'の更新は$Q^T A' = (I - YTY^T)^T A'$ を計算することが必要となる。これは行列行列積となるのでブロックの分割を適切に行うことで、ブロック化していない場合と比較して、メモリアクセスの問題が改善され、非常に高速な計算を行うことができる。

実装する場合には

$$\begin{aligned}(I - YTY^T)^T A' &= (I - \mathrm{Y}T^T Y^T)A' \\ &= A' - Y(T^T Y^T A') \\ &= A' - Y\left(A'^T YT\right)^T \end{aligned} \qquad (2.5)$$

と計算することで、Q を求める必要がなくなる。ここで

$$Y = [Y_1 \quad Y_2]^T \qquad A' = [A'_1 \quad A'_2]^T \qquad (2.6)$$

と置き、さらに計算用の領域 W を用意する。図で表すと図 2-5 のようになる。

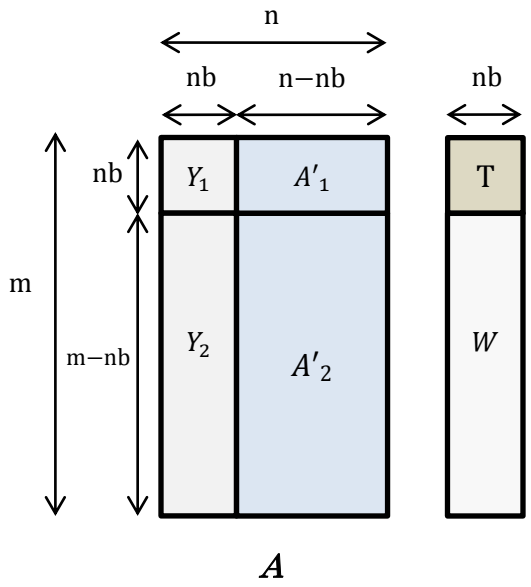


図 2-5 A の更新に必要な計算領域

式(2.5)は以下の手順で計算される。

まず、$W = A'^T Y$の計算を行う。式(2.6)より、

$$W = {A'_1}^T Y_1 + {A'_2}^T Y_2 \tag{2.7}$$

となる。式(2.5)は

$$W \leftarrow {A'_1}^T Y_1 \tag{2.8}$$

$$W \leftarrow W + {A'_2}^T Y_2 \tag{2.9}$$

と計算する。次に

$$W \leftarrow WT \tag{2.10}$$

を行う。この時点で$W = A'^T YT$ である。最後に$A' \leftarrow A' - YW^T$を計算する。

$$A'_2 \leftarrow A'_2 - Y_2 W^T \tag{2.11}$$

この時点で A'_2の更新が完了する。最後に A'_1 更新で以下の計算を行う。

$$\mathrm{W} \leftarrow \mathrm{W}Y^T \tag{2.12}$$

$$A'_1 \leftarrow A'_1 - W^T \tag{2.13}$$

以上の計算で A'の更新が行われたことになる。以降のブロックでも同様の処理を行い、A を Q,R に分解する。大きく 3 段階に分けると WY 表現を使ったブロックハウスホルダーQR のアルゴリズムは図 2-6 のようになる

(1) 列方向に分割されたブロックをハウスホルダーQR 分解する。
(2) 得られたハウスホルダーベクトルより T を計算する。
(3) $(I - YTY^T)^T A'$を計算し、分解した部分以外を更新する

図 2-6 ブロックハウスホルダー法のアルゴリズム

2.4 TSQR

Tall Skinny QR (TSQR) [1]は非常に縦長(m≫n)の行列 A に対して、行列を繰り返し行方向に分割して計算を行うアルゴリズムである。

行列 A を行方向に 2 分割して TSQR を行った場合の様子を図 2-5 に示す。TSQR の流れとしては、まず、A を図 2.4 のように 2 分割し、分割されたそれぞれの行列 A_1、A_2 で QR 分解を行う。得られた Q_1、R_1、Q_2、R_2 のうち R_1、R_2 の上三角行列部分を結合した行列に対して再び QR 分解を行う。ここで得られた R_3 が最終的な R となる。TSQR のアルゴリズムは図 2-7 のようになる。図 2-7 中でハウスホルダーQR(A_i,Q_i,R_i)は行列 A_i をハウスホルダーQR によって Q_i と R_i に分解することを意味する。

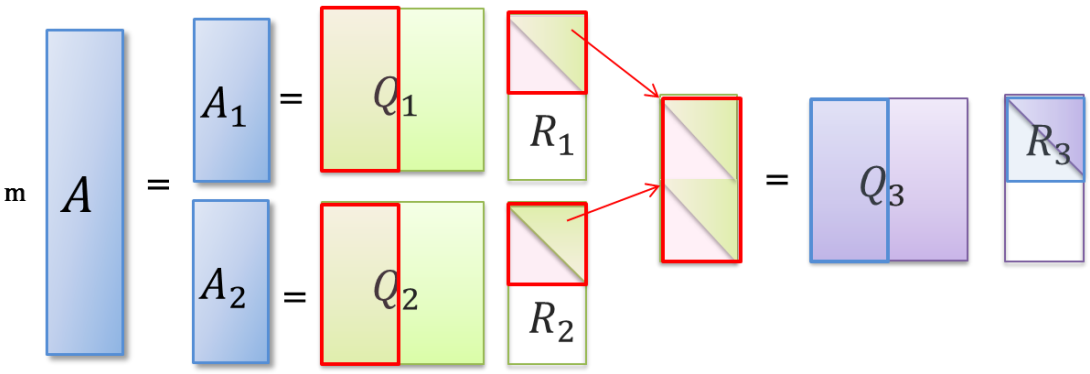


図 2-7 TSQR のイメージ図(2 分割)

TSQR のアルゴリズム

for(k = d to 1) {
  for(i = 1 to 2^k){
    (1)　ハウスホルダーQR(A_i,Q_i,R_i)　　//部分行列の QR 分解
  }
  for(i = 1 to 2^{k-1}) {
    (2)　$A_i = \begin{bmatrix} R_{2i} \\ R_{2i+1} \end{bmatrix}$　　//R の結合
  }
    (3)　$Q \leftarrow Q \begin{bmatrix} Q_1 & \square & \square & \square \\ \square & Q_2 & \square & \square \\ \square & \square & \ddots & \square \\ \square & \square & \square & Q_{2^k} \end{bmatrix}$　　//陽な Q の計算
}

図 2-8 TSQR のアルゴリズム

最終的に元の A に対する Q(陽な Q)を得るためにはそれぞれの Q の掛け算を行う必要がある。その様子を図 2-8 に示した。このアルゴリズムではそれぞれの再帰レベルでの QR 分解、Q の掛け算は独立して行うことができるが、計算量は従来のものより多くなる。

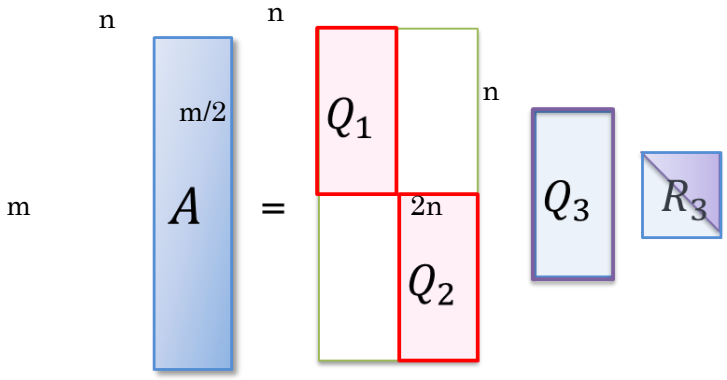


図 2-9　小さな Q の大きさ

TSQR で最初に行われる QR 分解で得られる Q は A を行方向に分割した数を p とすると、図 2-9 より (m/p)×n の大きさのものが p 個になる。2 回目以降のハウスホルダーQR では分解する行列の大きさも小さくなり、得られる Q の大きさは 2n×n となる。TSQR を実装する場合には得られた Q を順番通りにかけて陽な Q を計算するよりも、2 回目以降に得られた Q を掛け、そこに最後に 1 回目で得られた Q を掛けた方が計算量が少なくなる。

2.5 並列計算におけるハウスホルダーQR と TSQR の比較

行列 A を以下のように行方向で上下に 2 分割して並列計算を行う場合でハウスホルダーQR と TSQR の比較

を行う。

$$A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}$$

簡略化した並列ハウスホルダーQR のアルゴリズムを示すと図 2-10 のようになる。

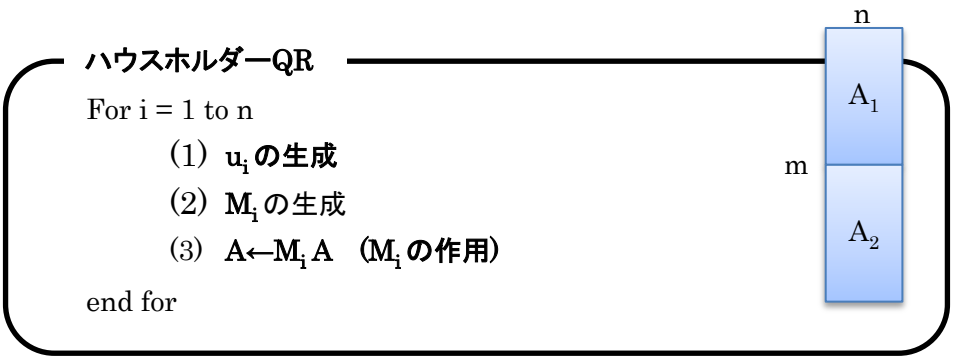


図 2-10 簡易ハウスホルダーQR のアルゴリズム

「(1) u_i の生成」では行列 A の i 列目の要素が全て必要なのでデータ交換の通信が 1 回必要となる。「(2) Mi の生成」では通信は必要ない。「(3) M_i の作用」では M_iA を計算するためにデータ交換の通信が 1 回発生する。したがって、1 ループで通信が 2 回発生することになり、ハウスホルダーQR 全体での通信回数はで 2n 回となる。通信の回数が多くなってしまうと通信によるプロセス間の同期が必要となってしまうので、それによるオーバーヘッドによる非効率化が問題となる。次に、並列 TSQR の簡易アルゴリズムを図 2-11 に示す。

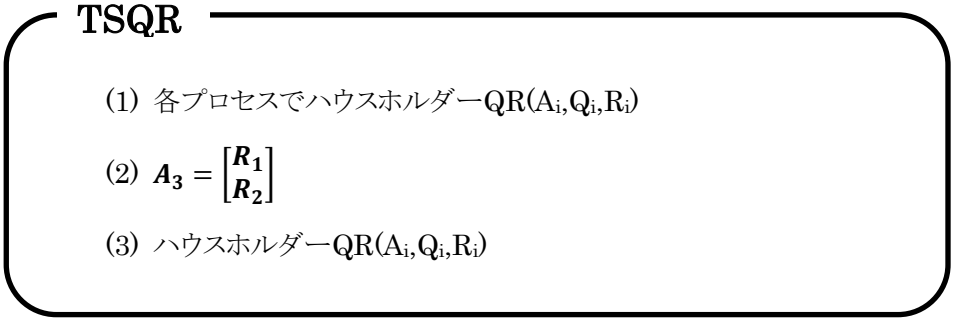


図 2-11 TSQR の簡易アルゴリズム

上記の並列ハウスホルダーQR と異なり、各ハウスホルダーQR は独立して実行されるので通信は発生しない。 通信が必要となるのは図 2-11 中の 2.の R の結合の 1 回のみである。再帰回数が k 回の場合、通信回数は k 回となる。

第3章. GPU による TSQR の並列化

3.1 GPGPU

GPU(Graphic Processing Unit)はグラフィックスの処理を主な目的としたハードウェアである。内部に数百個の演算器を持つため、演算性能に関しては CPU と比べるととても高い。GPU は単純な計算を大量に行うことを得意としており、グラフィックス処理に限定せずに数値計算にも応用されるようになった。

GPU による汎用計算に利用され始めたころは、計算を行うにはシェーディング言語と呼ばれている特殊なグラフィックス専用のプログラミング言語で記述する必要があり、GPU を利用しようとする人にはグラフィックスの知識が求められていた。

しかし、GPU 上での汎用計算開発環境である NVIDIA の CUDA(Compute Unified Device Architecture)[6]の登場により GPU コンピューティングの妨げとなっていた多くの制限を解消することができ、より GPU コンピューティングはより広く受け入れられるようになった。

さらに、3D 画像よりも HPC 向けの NVIDIA Tesla という GPU の登場や、それまで単精度浮動小数のみでしか計算できなかったが、汎用計算向けアーキテクチャの Fermi アーキテクチャでは倍精度浮動小数計算が可能になるなど、GPU コンピューティングによる演算精度も上昇している。

GPU における演算性能も日々進化をしており、近年ではその演算能力の高さのためスーパーコンピュータにも GPU を搭載しているものが数多く存在する。2012 年 11 月にスーパーコンピュータの性能ランキング「TOP500」で第 1 位に選ばれた「Titan(タイタン)」(2013 年 11 月時点で 2 位)にも NVIDIA Tesla K20X が組み込まれている。Tesla K20X は Fermi アーキテクチャの次のアーキテクチャである Kepler アーキテクチャを採用しており、ピーク時の演算性能は単精度浮動小数演算で 3.95T FLOPS、倍精度浮動小数演算で 1.31 T FLOPS の性能を持つとされている。

3.2 CUDA

CUDA は NVIDIA 社製の GPU で使用することができるプログラミング言語であり、それまでのシェーディング言語のような画像処理用の知識を必要とせず、一般的な C や C++のような形でプログラムを記述することができる。図 3.1 に CPU と GPU の関係、GPU の内部構造を示した。

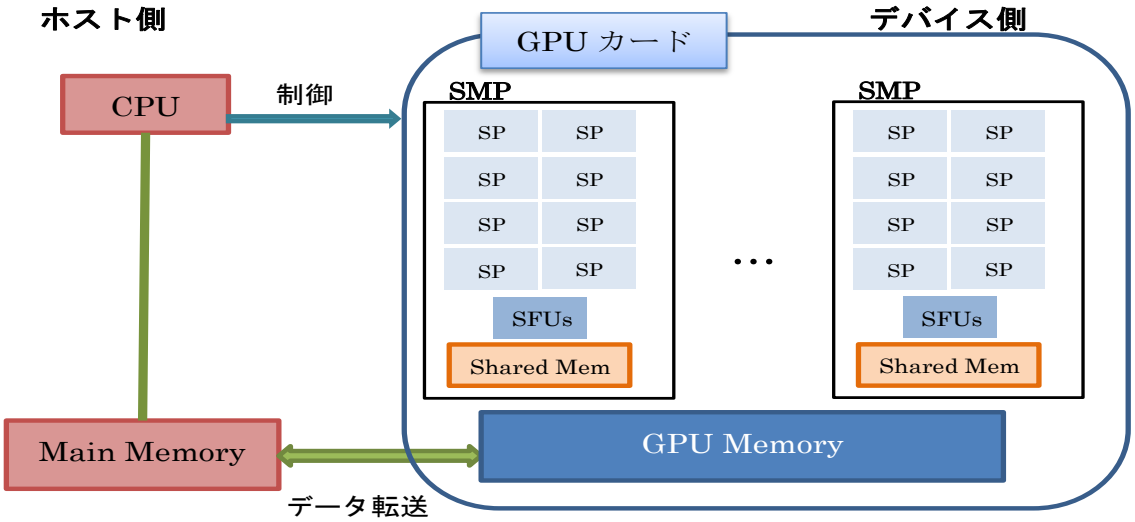


図 3-1 GPU の内部構造

GPU は内部に SP(Streaming Processor)と呼ばれる演算装置を数多く持っている。SP は最小単位の演算処理装置であり、SP と浮動小数の乗算などを行う SFU(Special Function Unit)を用いることで高速な計算を実現している。この SP と SFU と共有メモリで構成されるまとまりを SMP(Streaming Multi Processor)という。

CUDA プログラミングは GPU で計算を処理したい部分をホスト(CPU)コードからカーネルカーネル(GPU)関数を呼ぶ形で実装する。カーネルは 1 つのグリッド(Grid)から構成され、グリッドは複数のブロック(Block)から構成されている。さらにブロックは複数のスレッド(Thread)を持つ。

CUDA での並列計算はブロックとスレッドにより行われる。スレッドはプログラミングモデルからみた場合の実行の最小単位で、ブロックは指定した数からなるスレッドの集合である。スレッドには連続したスレッドが連続したメモリに同時にアクセスすることで高速にメモリアクセスを行うことができるという特徴がある。図 3-2 に CUDA の階層的なモデルを示した。

ブロック内のスレッドは SMP によって 32 スレッド毎に並列実行され、このまとまりをワープ(warp)と呼ぶ。これは全スレッド数によらず固定の値なので、32 の倍数のスレッド数で実行することで計算速度の性能を引き出すことができる。

CUDA はメモリも階層的になっており、全ブロックから参照できるグローバルメモリ、一つのスレッドからのみ参照できるローカルメモリ、ブロック内全体で共有のシェアードメモリなどがある。グローバルメモリ、ローカルメモリはオフチップのメモリであり低速であるが、シェアードメモリはオンチップなので高速にアクセスできるので、これらの高速なメモリやレジスタをうまく扱うことで計算速度を上げることができる。

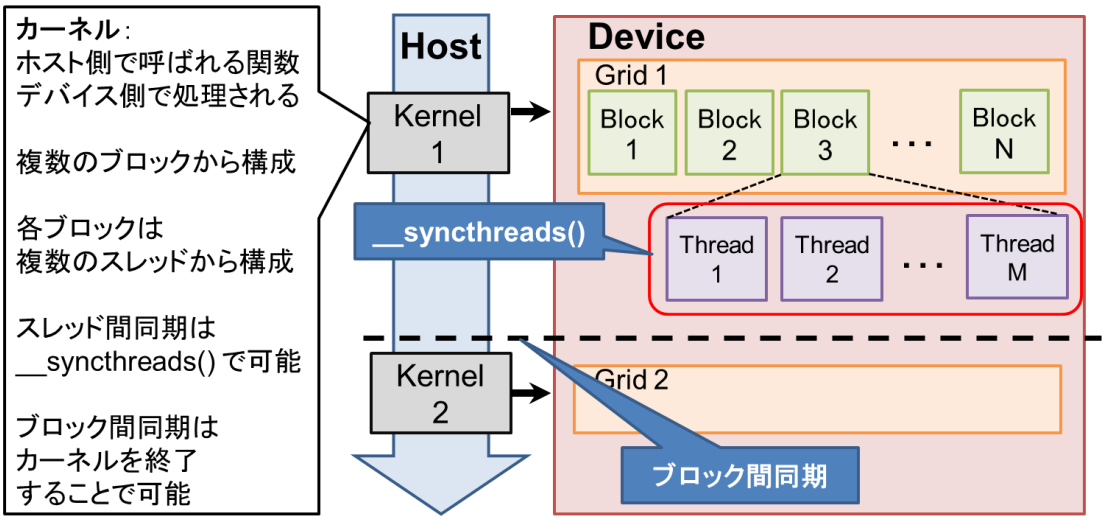


図 3-2 CUDA の階層的モデル

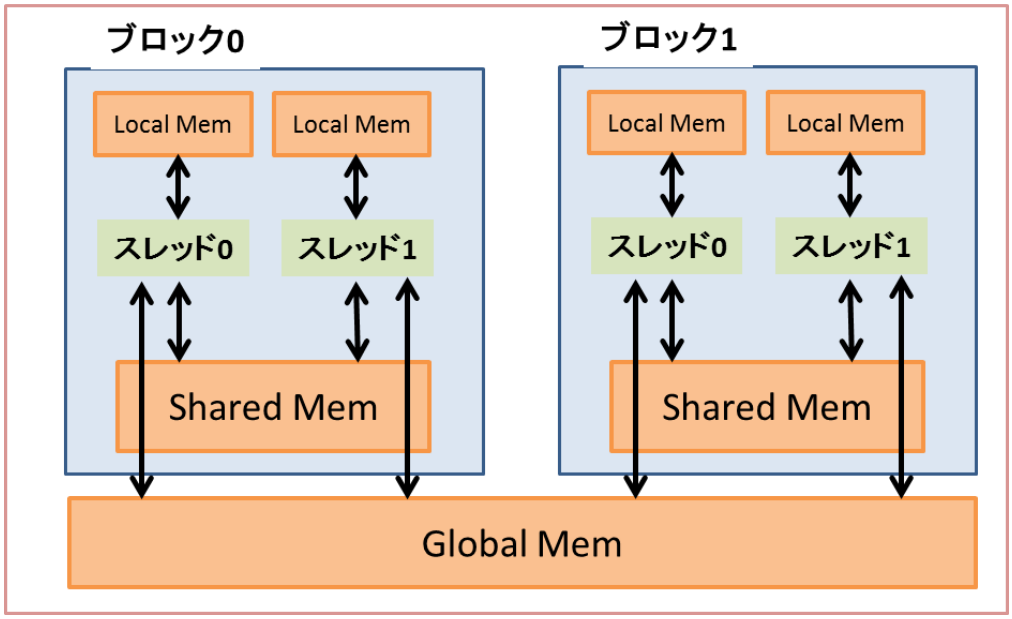


図 3-3 GPU のメモリ階層(一部分)

3.3 MAGMA

Matrix Algebra on GPU and Multicore Architectures (MAGMA)[6]は NVIDIA 社が提供する GPU 向け(CUDA 向け)の線形代数ライブラリである。CPU と GPU を同時に使うハイブリッド型のライブラリであるので、GPU 単体よりも高速となる。MAGMA の現在の最新バージョンは 2014/01/08 にリリースされた MAGMA1.4.1 である。MAGMA ライブラリで倍精度ハウスホルダーQR 分解を行う関数はである magma_dgeqrf()であるが、この関数では LAPACK 同様にブロックハウスホルダーQR による QR 分解で実装されおり、本研究で扱う GPU への完全オフロードによる TSQR の実装をした関数は MAGMA では実装が行われていない。また、GPU で実装が困難と思われる部分を CPU で実装しており GPU に完全オフロードするためにはデバイス側での高度な実装が必要と考えられる。

3.4 CUDA TSQR の実装

従来の研究ではマルチコアによる並列 TSQR などが実装[3][7]されているが、本研究では CUDA による GPU への完全オフロード TSQR の実装を行う。この節では本研究での CUDA TSQR の実装についての説明を行う。

3.4.1 CUDA TSQR

CUDA で TSQR を並列化する場合、

(1) TSQR をホスト関数、デバイス関数へ割り振り方

(2) CUDA スレッドの割り振り方

(3) TSQR の各階層間の Q,R の同期方法

を考える必要がる。

(1)は TSQR で並列計算可否部の扱いの問題で、本研究では並列計算できる TSQR 各階層での複数個の小行列の QR 分解と Q の行列積をカーネル関数でとした。

(2)については、まず、CUDA の各ブロックで T スレッドを用いて T 個のハウスホルダーQR を処理させる実装が考えられる。この方法では A の行方向分割数を N 個、1 つの CUDA ブロックのスレッド数を T とした場合、$(N+T-1)/T$個のブロックを生成し、1 ブロックあたり T 個のハウスホルダーQR を処理することになる。この様子を図 3-4 に示す。ここで、各 QR 分解はブロックハウスホルダーQR を用いて分解している。

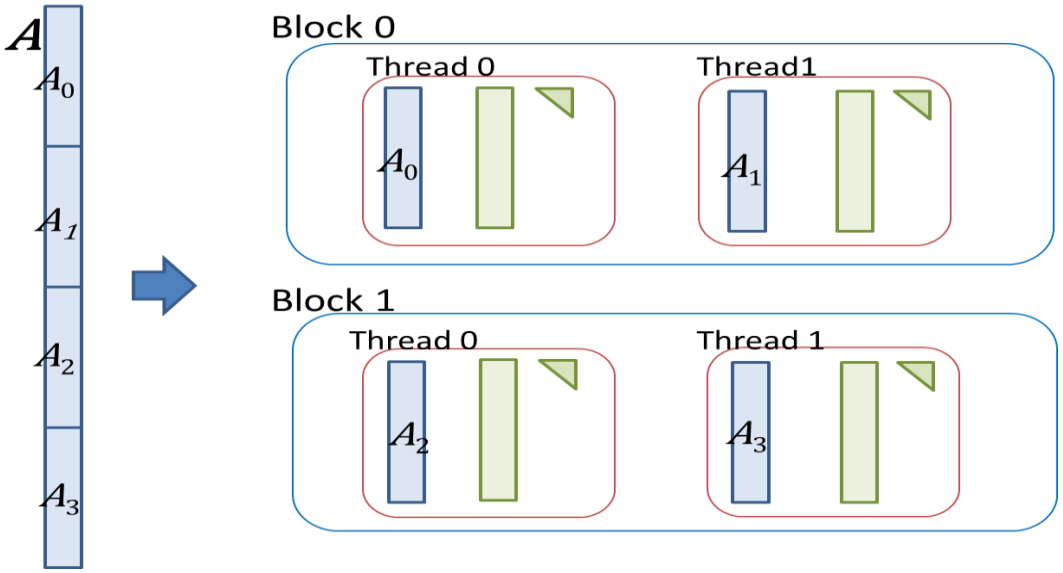


図 3-4　1 スレッドで 1 つの QR 分解する場合(2 スレッド)

しかし、3.2 節で述べたように CUDA では各スレッドがメモリ上で連続したデータを参照できないとメモリの転送時間が多くなってしまう。上記の方法の場合では、CUDA の各スレッドがメモリ上で連続したデータを参照することができず離れた位置にあるデータを扱うこととなり GPU の計算能力を生かすことができない。

そこで、次に CUDA の各ブロックに 1 つのハウスホルダーQR をマルチスレッドで処理させる並列化が考えられる。この実装方法ではデバイス側では N 個のブロックを生成し、各ブロックでは T スレッド並列で 1 つの小行列の QR 分解を処理することになる。先行研究[9]でもこの実装が行われている。図 3-5 にその様子を示す。

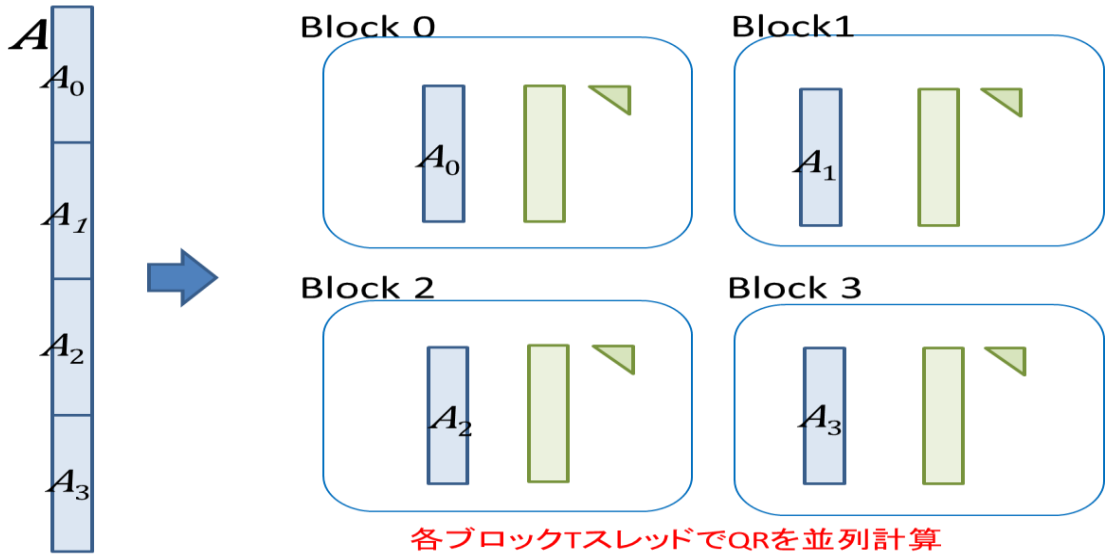


図 3-5　T スレッドで 1 つの QR を並列計算

(3)について、CUDA ではスレッド間では__syncthreads 命令を用いることで同期することができるが、ブロック間でグローバルメモリを同期する命令は用意されていない。この実装の場合、2 分木構成 TSQR の再帰レベルごとにカーネルを一旦終了し、再び次の再帰レベルでカーネルを起動する必要がある。つまり、システムレベルでの同期が複数回発生することになる。TSQR の第二階層以降で扱う行列はとても小さいものとなるので、データの移動は少ないものの、実行時間全体に占める同期のコストは大きくなることが問題となる。図 3-6 にその様子を示した。

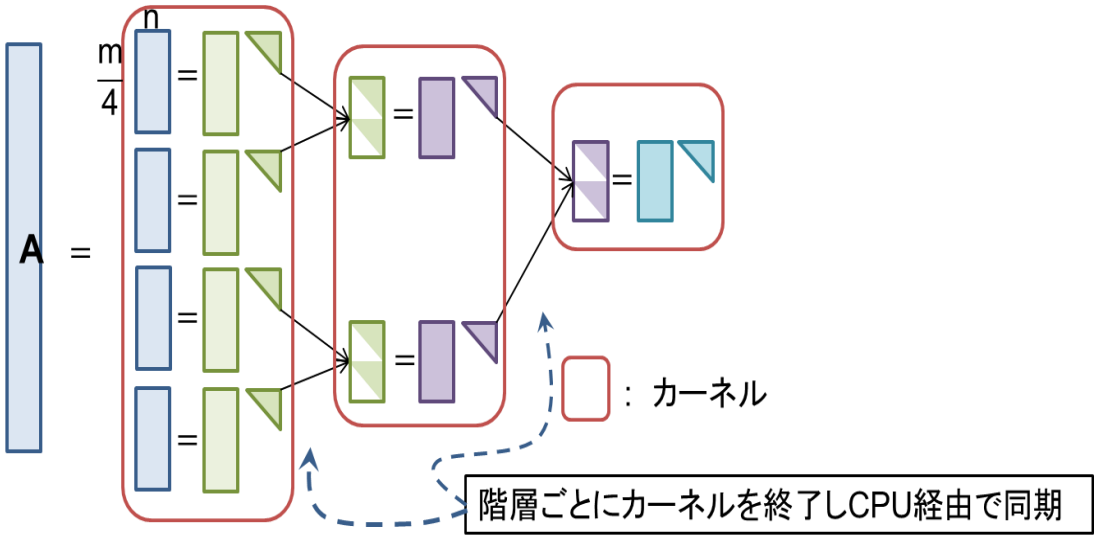


図 3-6　CUDA TSQR - 複数カーネル版(2 分木構成)

CUDA で並列計算を実装する場合、CPU 経由の同期はオーバーヘッドが大きいのでるので、少ないデータ量に対して何度もカーネル関数を呼ぶと、その操作がボトルネックとなってしまい、GPU の計算性能を発揮

しきることができなくなってしまう。ここで、TSQR の場合、m>>n なので第一段階の QR 分解以外で扱う行列はとても小さくなる。本研究では TSQR アルゴリズム中、最初の階層の QR 分解で得られた R を全て結合し、それに対して QR 分解を行い、Q と R を得る、再帰レベルが常に 2 の TSQR を実装する。図 3-7 はその様子を示している。

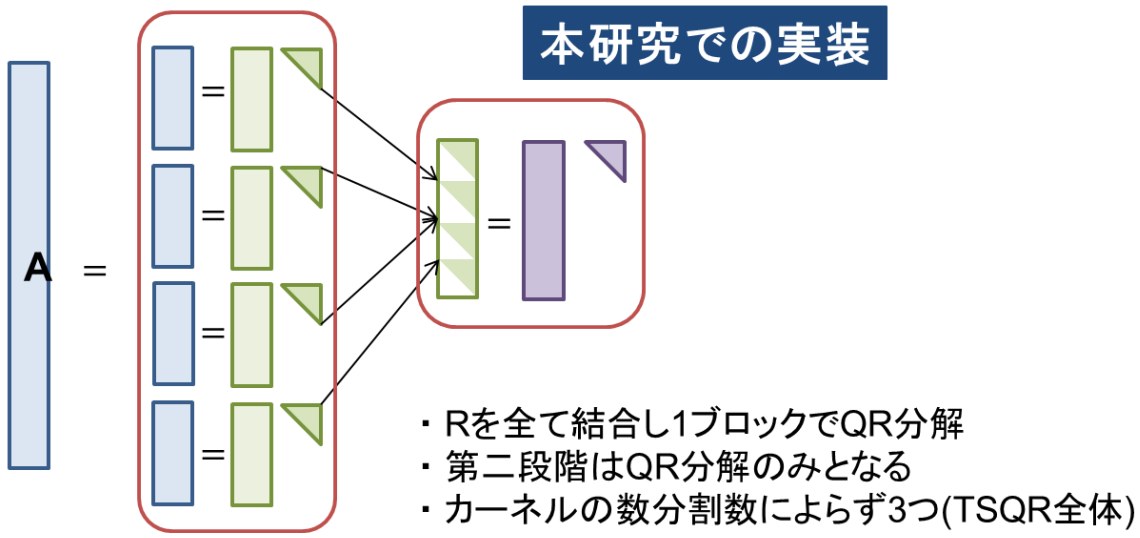


図 3-7 CUDA TSQR - 複数カーネル(1 回で全ての結果を集約)

3.4.2 デバイス側でのハウスホルダーQR のマルチスレッド処理

本研究の CUDA-TSQR では CUDA の各ブロックは分割された複数個の行列に対してハウスホルダーQR を行っているが、図 2-2 のアルゴリズム中で u の計算部分の(1)、 A の更新部分の(4)の行列計算をマルチスレッドで処理している。

(1) ベクトルの内積計算

図 2-2 の(1)のようなではまず 1 つのスレッドが 1 サイクルで 1 つの数値の 2 乗計算を行い、各スレッドに対応付けされたシェアードメモリに加算する処理を繰り返す。すべてのスレッドが計算を終了したら、各スレッドの計算結果を足していくことでs^2を求めている。図 3-8 は 4 スレッドで(1)を処理する場合の様子を示している。

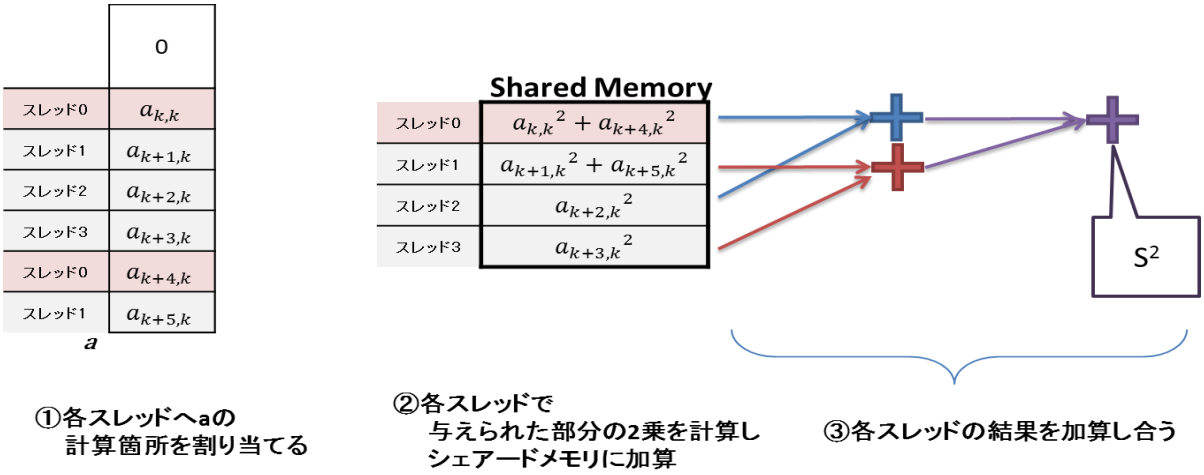


図 3-8 (1) $s=\sqrt{a_{k+1,k}^2+a_{k+2,k}^2+\cdots+a_{n,k}^2}$のマルチスレッド計算(4 スレッド)

(2) ベクトル行列積

図 2-2 の(4) $w \leftarrow u^t A$ の行列ベクトル積についても、同様に

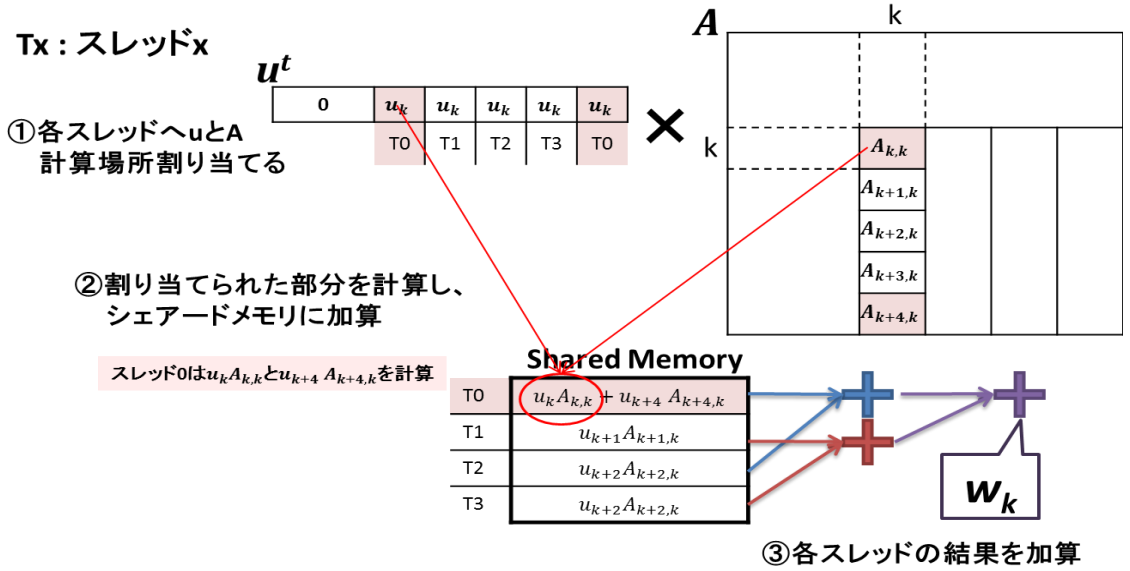
① スレッドに計算する値を割り当てる

② 各スレッドで計算した結果をシェアードメモリ自身に割り当てられた部分に加算

③ シェアードメモリの結果のすべての和を計算する。

④ ①~③を k 列目から n-1 列目まで繰り返す

図 3-9 にその様子を示す。



④ ①~③をAのk列目からn-1列目まで繰り返す

図 3-9 $w \leftarrow u^t A$のマルチスレッド計算(4 スレッド)

第4章. 実験

4.1 実験環境

以下で行う実験では 3 つの環境で 3 種類の GPU を使用した。は各実験環境、GPU について表 4-1、表 4-2、に示した。

表 4-1 3 つの実験環境

実験環境 1	
OS	Fedora 18 3.10.11-100.fc18.x86_64
CPU	Intel(R) Core(TM) i5 CPU 750 @ 2.67GHz *4
GPU	GeForce GTX 560 Ti
実験環境 2	
OS	Fedora 18 3.10.12-100.fc18.x86_64
CPU	Intel(R) Core(TM) i7-2700K CPU @ 3.50GHz *4
GPU	GeForce GTX 590
実験環境 3	
OS	Fedora 18 3.11.10-100.fc18.x86_64
CPU	Intel(R) Core(TM) i7-4770S CPU @ 3.10GHz *8
GPU	GeForce GTX 780

表 4-2 各環境の GPU の概要

GPU name	SMP 数	CUDA core 数	Global Mem	Compute capability
GeForce GTX 560 Ti	8	384	2.0GB	2.1
GeForce GTX 590	16	1024	1.5GB	2.0
GeForce GTX780	12	2304	3.0GB	3.5

4.2 CUDA TSQR 中の処理時間の割合

本研究で実装した CUDA TSQR では TSQR の第二階層の QR 分解処理は1つの CUDA ブロックで実行されており、コストは A の分割数に比例する。第一階層の QR 分解、Q の乗算は分割数が適切な数であれば並列処理が可能になるため高速化が期待できるが，第二階層の QR 分解の影響もあるので、適切な分割数を選ぶ必要がある。ここでは、今回の環境ではどのような分割数が適切であるかを実験する。図 4-1、図 4-2、図 4-3 は各環境で A:524288×64 として、分割数を変化させて実験し、TSQR の実行時間を第一階層の QR(QR1)、第二階層(QR2)、Q の乗算、CPU-GPU 間データ転送の時間に分けてグラフにしたものである。

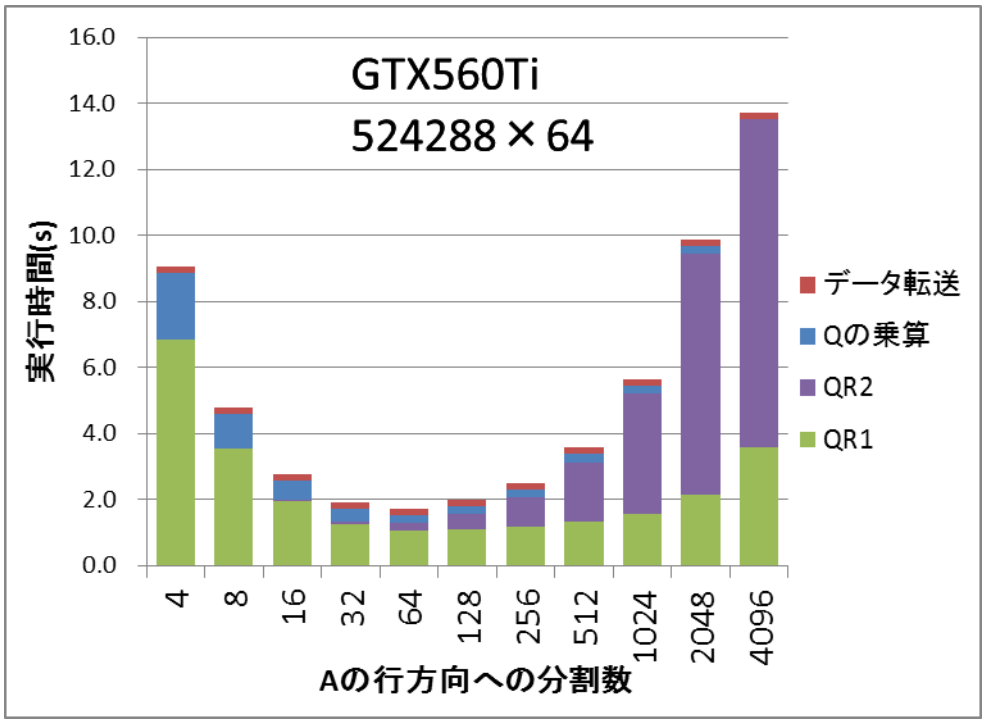


図 4-1 実験環境 1(GTX560Ti)

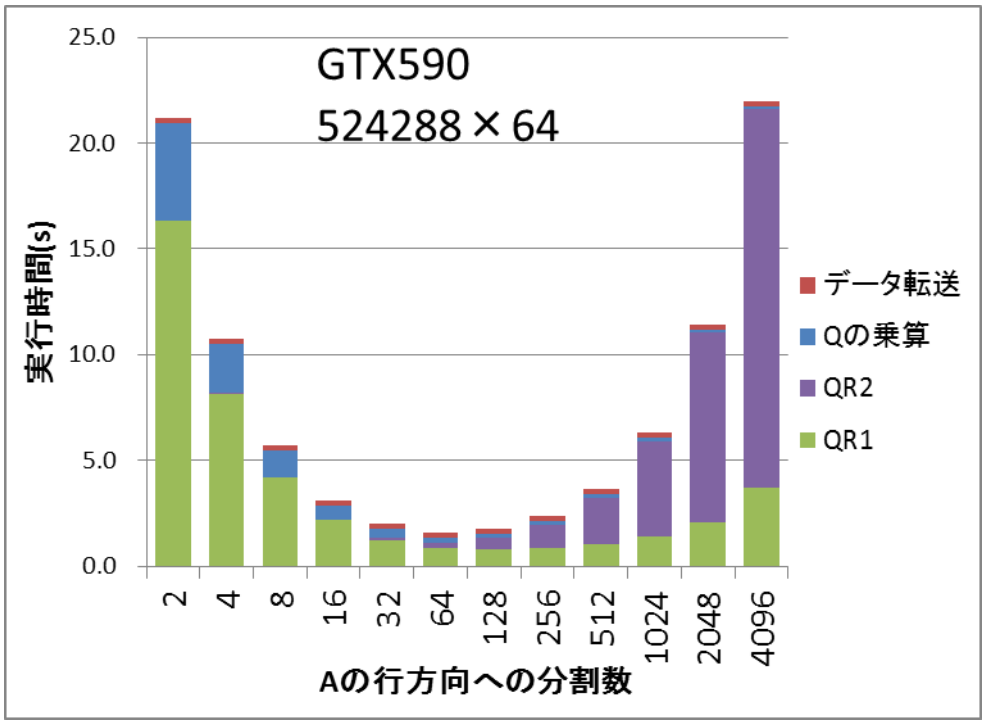


図 4-2 実験環境 2(GTX590)

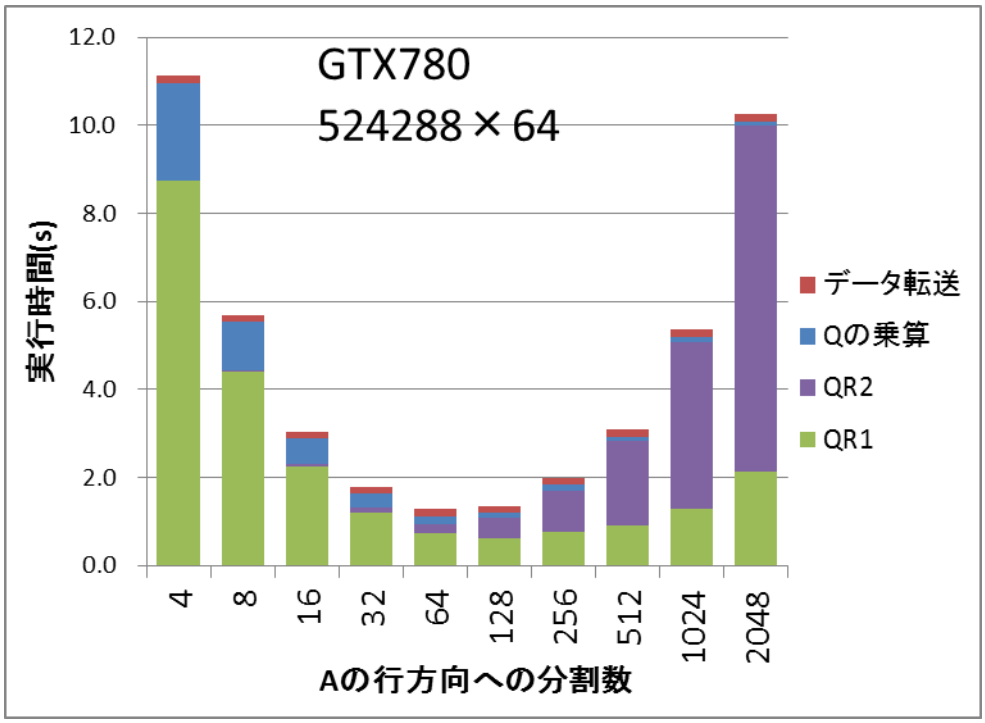


図 4-3 実験環境 3(GTX780)

図 4-1、図 4-2、図 4-3 よりどの環境でも分割数を増やすことで QR1、Q の乗算の計算時間が減少していることがわかる。しかし、分割数を増やしすぎると、QR2 で分解する行列が大きくなり計算時間が急激に増加している。また、CUDA ブロックは 65535 個まで生成可能だが実際に並列計算される数は GPU の SMP の数とキューに積むことのできるブロック数によって限られているので QR1 部分も分割数を増やしすぎると計算時間が増加していることがわかる。

4.3 既存ライブラリ MAGMA との実行時間比較

MAGMA と本研究で実装した CPU 版(逐次版)TSQR と Row-major CUDA TSQR、Column-major CUDA TSQR の実行時間を比較した。その結果が図 4-4、図 4-5、図 4-6 である。各図中の TSQR(X)は X 分割で処理した場合を意味する。

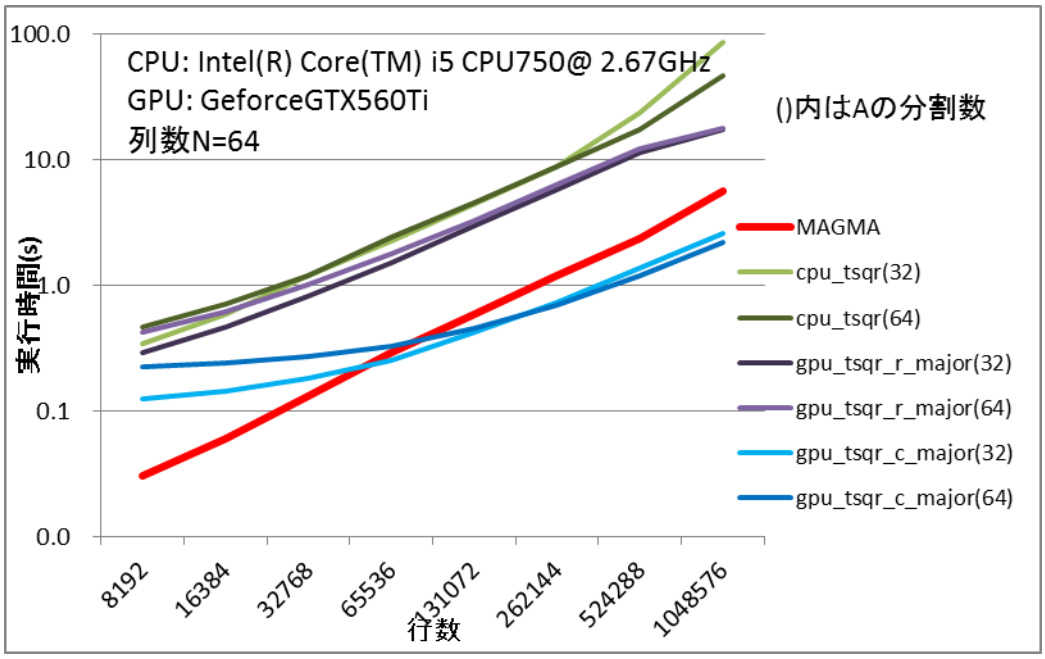


図 4-4 行数毎の実行時間(GTX560Ti)

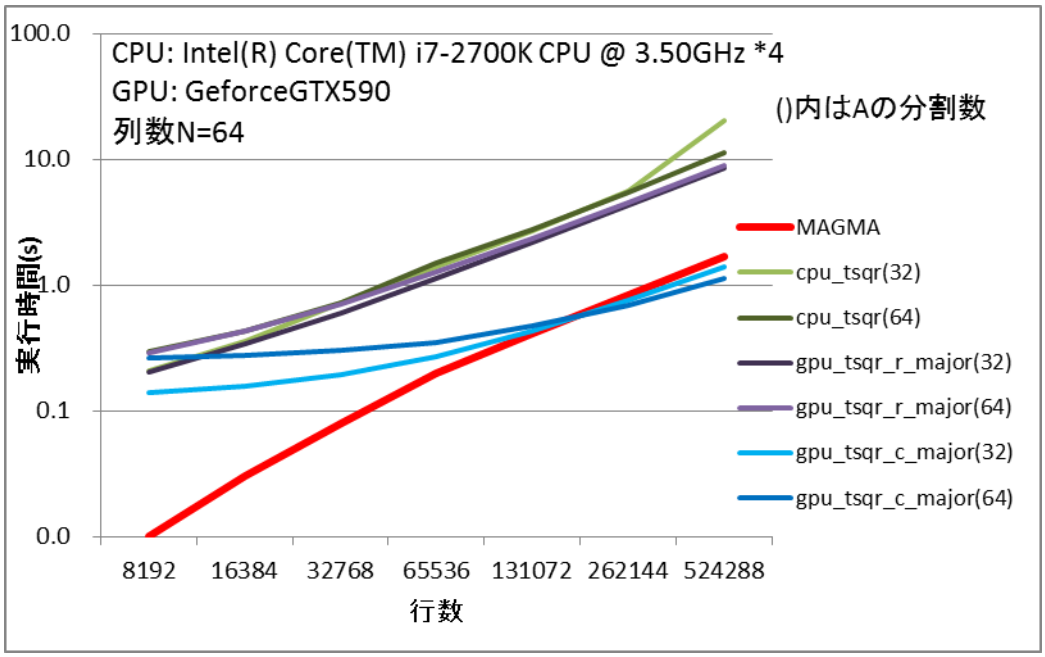


図 4-5 行数毎の実行時間(GTX590)

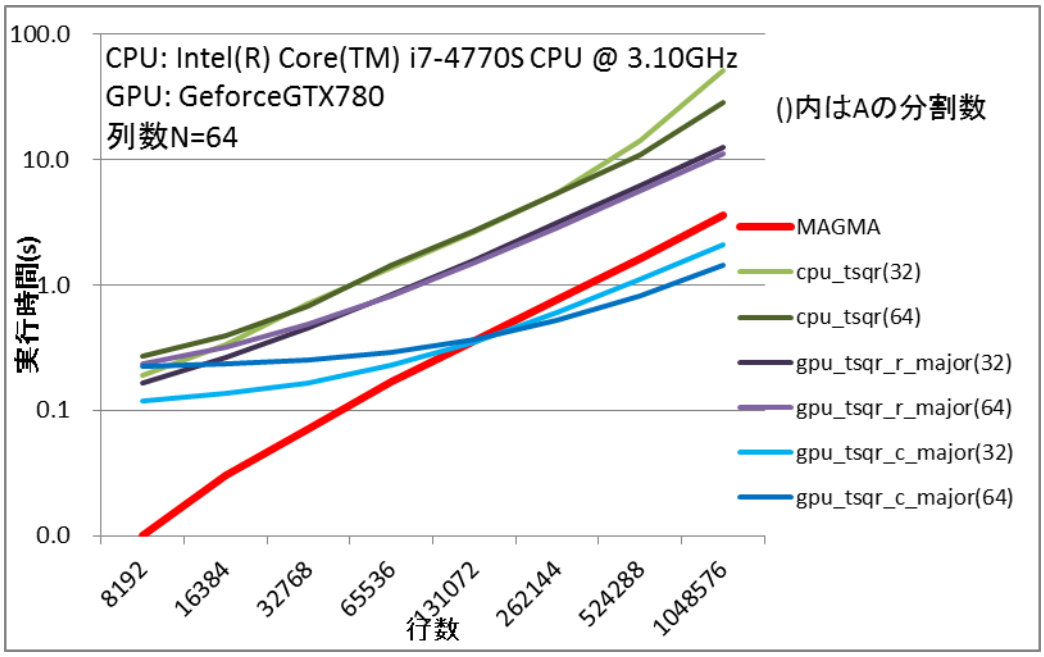


図 4-6 行数毎の実行時間(GTX780)

各図より基本的に Row-major よりも Column-major での実装の方が高速である。これは、TSQR では縦長のベクトルの内積計算が多いためであると考えられる。しかし、ベクトル長が十分に長くない場合には Row-major の方が高速である場合もある。

MAGMA との比較でアは行列サイズが大きい場合、GTX560Ti で 2.56 倍、GTX590 で 1.50 倍,GTX780 で 2.50 倍の高速化となっていることがわかる。現状での CUDA TSQR はデバイス側ブロックハウスホルダーQR 分解部分の実装は最適化されておらず MAGMA は最適化されたライブラリであるので、TSQR の並列性の高さは GPU でも有効であることが確認できる。本実装の最適化が進めばグラフの gpu_tsqr_r_major(紫色のライン)と gpu_tsqr_c_major(青色のライン)のが全体的に下がることになり、さらに CUDA-TSQR の優位性が高まることが考えられる

第5章. 結論

5.1 まとめ

実験結果より本研究で実装した TSQR は行列サイズによっては既存ライブラリより高速となることがわかった。本研究での QR 分解部分はまだ最適化されておらず MAGMA は最適化されたライブラリであるので、TSQR の並列性の高さは GPU でも有効であることが確認でき、デバイス側での HouseholderQR 処理部分の最適化によりさらなる高速化が望める。

また、本実装では TSQR を再帰数 2 で、2 階層目の QR 分解は GPU の 1 ブロックのみで処理しているため、後半部分で GPU の計算能力を発揮しきれていない。よって、この部分の並列性を高める必要がある。

5.2 今後の展望

5.2.1 デバイス側での QR 分解の高速化

CUDA TSQR の現状の実装ではデバイス側での QR 分解中の行列演算プログラムが低速であり、最適化の必要がある。最適化の方針の一つとしてループを shared メモリに収まるようにブロック化してループアンローリングする方法が考えられる。

また、現在最新の Kepler アーキテクチャで Compute Capability が 3.5 以上の GPU ではカーネルから動的に別のカーネルを起動することが可能(ダイナミック並列処理)である。これを利用してブロックハウスホルダーQR 内の TrailingMatrix の更新処理部分や Q の乗算の行列行列演算に CUBLAS の高速な関数を使うことができれば、更なる高速化が可能であると考えられる。

5.2.2 TSQR の並列性を高めた実装

今回は TSQR 再帰階層中の最初の QR 分解部分をのみをデバイス側で処理するような実装であったが、TSQR の処理を全てデバイス側で実行できれば更なる高速化が見込める。しかし、第 3 章でも述べたように TSQR の各再帰レベルごとに CPU-GPU 間のデータ転送が発生してしまう。

しかし、これまでの研究で図 5-1[10]のプログラムのように、CUDA で用意されている atomic 演算の atomicCAS()を利用することでカーネルを終了せずにブロック間の同期ができることが分かっている。さらに、Kepler アーキテクチャでは従来のものより atomic 演算のスループットが改善されている[11]。このように、ブロック間同期を行うことで図 5-2 で示すような TSQR の 1 カーネル実装も可能であると考えられる。ただし、アクティブでないブロックが存在する場合にデッドロックに陥る可能性があり、実装時にはそのあたりを考慮する必要がある。

```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <ctype.h>
#include <string.h>
#include <stddef.h>
#include <stdint.h>
#include <cuda_runtime.h>
#include <cuda.h>
#if CUDA_VERSION < 5000
#     include <cutil.h>
#endif

__device__ int lock_var[3] = { 0, 0, 0 } ;
#define    lock( ... ) ¥
              do { if ( !atomicCAS( lock_var, 0, 1 ) ) break; } while ( 1 )
#define    unlock( ... ) ¥
              atomicExch( lock_var, 0 )


__device__ void
interblock_barrier ( void ){
      int is_master = (!threadIdx.x) && (!threadIdx.y) && (!threadIdx.z);
      int num_block = gridDim.x * gridDim.y * gridDim.z;
      __syncthreads( );
       if ( is_master ) {

            volatile int t;

            lock( );
            t = lock_var[1] + 1;
            if ( t == num_block ) lock_var[2] = t;
            lock_var[1] = t;
            unlock( );

            while ( 1 ) {
                  lock( );
                  t = lock_var[1];
                  unlock( );
                  if ( t == num_block ) break;
            }

            lock( );
            t = lock_var[2] - 1;
            if ( t == 0 ) lock_var[1] = t;
            lock_var[2] = t;
            unlock( );

            while ( 1 ) {
                  lock( );
                  t = lock_var[2];
                  unlock( );
                  if ( t == 0 ) break;
            }
      }
      __syncthreads( );
}
```

図 5-1 atomic 演算を使ったブロック間同期

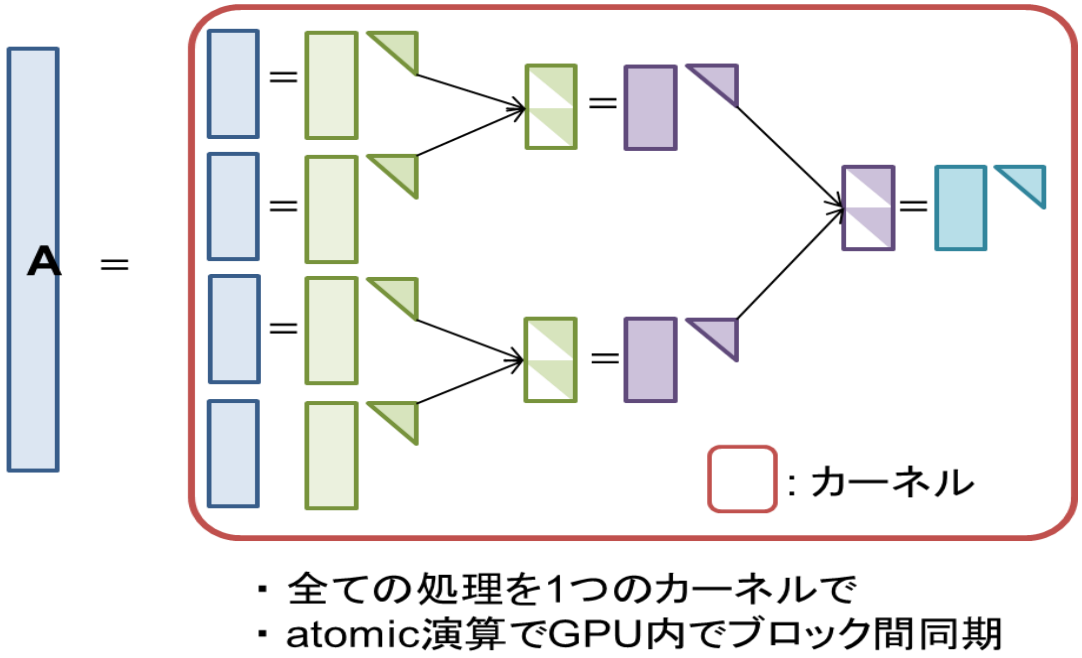


図 5-2 CUDA TSQR - 1 カーネル版

参考文献

[1] James Demmel, Laura Grigori,Mark Hoemmen, and Julien Langou, Communication-avoiding parallel and sequential QR factorizations, EECS-2008-74
May 29, 2008

[2] 森 大介, 山本 有作, 超 紹良, 情報処理学会研究報告. [ハイパフォーマンスコンピューティング] 一般社団法人情報処理学会 2008(99) (20081008), pp25-29, 2008

[3] 深谷 猛, 山本 有作, 超 紹良, 情報処理学会研究報告. [ハイパフォーマンスコンピューティング] 一般社団法人情報処理学会 2009-HPC-121(18) (20090728), pp1-7, 2009

[4] 戸川 隼人, 科学技術計算ハンドブック, サイエンス社, 1998

[5] Robert Shreiber , Charles Van Loan , A Storage Efficient WY Representation for Products of Householder Transformations , SIAM J SCI STAT COMPUT vol 10, No 1, pp53-57, 1989

[6] http://www.nvidia.co.jp/page/home.html

[7] Matrix Algebra on GPU and Multicore Architectures, http://icl.utk.edu/magma/

[8] 村上 弘, マルチコア CPU システムおよび小規模 SMP 並列システム上での Tall Skinny 型 QR 分解法の実験, 情報処理学会シンポジウムシリーズ v2009 , pp273-289, 2009

[9] Michael Anderson , Grey Ballard , James Demmel , Kurt Keutzer Communication-Avoiding QR Decomposition for GPUs, EECS Department
University of California, Berkeley Technical Report No. UCB/EECS-2010-131
, 2010

[10] 今村俊幸, private communication, 2014 年 1 月

[11] Kepler コンピュート・アーキテクチャ・ホワイトペーパー
http://www.nvidia.co.jp/content/apac/pdf/tesla/nvidia-kepler-gk110-architecture-whitepaper-jp.pdf

[12] Jason Sanders, Edward Kandrot, CUDA by example 汎用 GPU プログラミング入門, インプレスジャパン, 2011

謝辞

本研究を進めるにあたって、今村俊幸先生には多くの事柄についてご指導を承りました。誠に感謝しております。また、指導教員の仲谷栄伸先生、龍野智哉先生にも大変お世話になり、感謝しております。最後に、旧今村研究室のメンバーのみなさんからも大変有益な助言をいただきました。深く感謝いたします。