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

田村 遼也[1,3]　今村 俊幸[2,3,a)]　仲谷 栄伸[1]

概要：近年提唱されている, 特に縦長行列 ($m \times n$ かつ $m \gg n$) の QR 分解アルゴリズム Tall Skinny QR(TSQR) は 階層的な構造による高並列性と通信最適性により極めて注目が集まっている解法である. 本研究では GPGPU による TSQR の実装の中でも, CPU 資源を極力使わず, 主たる計算部分を GPU に担当させる完全オフロード実装に関する研究を進めた. 現時点では TSQR のフルバリエーションは未完成ではあるが, 上三角行列 R の集約計算を限定したものの完全オフロード化に成功している. 既存の GPU 数値計算ライブラリの代表格である MAGMA と比較しても, 高速化するケースがあり, TSQR の並列性と GPU の高い処理能力が立証されたといえる.

1. はじめに

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. 特に縦長行列 ($m \times n$ かつ $m \gg n$ とする) 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, 大規模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 Householder QR 分解は Householder 変換自身の並列性はあるものの反復的かつ逐次的であり同期回数が多く高効率でスケーラブルな並列化が行いにくいとされてきた. 近年提唱された Tall Skinny QR(TSQR) アルゴリズムでは, アルゴリズム自身が持つ階層性とともに自明な並列性, 更に通信最適であると高い注目を集めている [1], [2], [3], [4]. TSQR に関する様々な並列処理の研究は重要性が高い.

従来研究から, TSQR については Householder QR 分解と比較して以下のことがわかっている.

- 並列性が高い
- 計算量が増加する
- 計算精度が良い

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 GPGPU による並列化を扱う. GPU は画像処理用のハードウェアで主に 3D ゲームなどのグラフィックスアクセラレータ処理を行っていた. GPU はアーキテクチャが単純であり, 簡単な演算だけでは CPU よりも計算能力が高く, 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 Graphics Processing Units (GPGPU) である. NVIDIA 社より GPGPU 専用の GPU である NVIDIA Tesla も登場し, 近年ではスーパーコンピュータにも GPU を搭載したものが存在する. 2012 年 11 月にスーパーコンピュータ性能ランキング TOP500 で 1 位となった「Titan」には Tesla K20x が数多く搭載されており, HPC において GPGPU が核技術となっていることは周知の事実といえる. 本研究では CUDA を利用した GPGPU による TSQR の高速化を目的とし, 本報告では来る将来のアクセラレータ時代の数値計算ライブラリの可能性を視野に入れ CPU 資源による演算を行わない (CPU は GPU 起動やブロック間の同期のみを行う) 完全オフロード化に関する報告を行う.

2. QR 分解と TSQR アルゴリズム

2.1 QR 分解 (QR Decomposition)

QR 分解は m 行 n 列の行列 A を直交行列 Q と上三角行

1 電気通信大学大学院情報理工学研究科
Graduate School of Informatics and Engineering, University of Electro-Communications, Chofu, Tokyo
2 理化学研究所 計算科学研究機構
RIKEN Advanced Institute for Computational Science, Kobe, Hyogo
3 科学技術振興機構 CREST
CREST JST, Kawaguchi, Saitama
a) imamura.toshiyuki@riken.jp

© 2014 Information Processing Society of Japan

列 R により

$$A = QR \tag{1}$$

の形に分解する線形代数の基本演算の一つである．QR 分解には, $m \times m$ の Q と $m \times n$ の R を求める fatQR タイプと $m \times n$ の Q と $n \times n$ の R のみを求める thinQR の 2 通りが存在する．本研究では後者の thinQR を扱うものとする．

序でも述べた通り, 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．特に, thinQR はベクトルの正規直交化に相当しており, 線形代数の教科書にも登場する計算方法として Gram-Schmidt のアルゴリズムが thinQR に相当している．更に, Jacobi 法、Householder QR, Cholesky QR など様々な方法が存在するが, 数値計算では計算精度の観点から Householder QR 分解が用いられることが多い．

2.2 Householder QR 分解

Householder QR 分解は

$$M = I - 2uu^t \quad (\|u\| = 1) \tag{2}$$

で定義される鏡像変換に対して, $\|x\| = \|y\|$ となる任意のベクトル x, y 間で $u = (x-y)/\|x-y\|$ ととれば $Mx = y$ となるように変換 M を決めることができる性質を利用する．

詳細は数値計算の教科書に委ねるが, m 次元ベクトルの列, $\{a_1, a_2, \ldots, a_n\}$ に対して,

(1) $x_k = [0, \ldots, 0, e_k, \ldots, e_m]a_k$, $y_k = \pm\|x_k\|e_k$ として上記変換 M_k を作成 (但し, e_k は k 要素のみが 1 の単位ベクトル).

(2) $[a_k, \ldots, a_n] := M_k[a_k, \ldots, a_n]$ の様に M_k を左側から作用させて更新する．

上記操作を $k = 1, ..., n$ と繰り返す方法である．最終的に $[a_1, \ldots, a_n]$ の上部 $n \times n$ 部分が R, $Q = M_1 M_2 \cdots M_n E_n$ となる．

2.3 Block Householder QR 分解

既存研究でも報告がある様に, 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．そのため, 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, Householder 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．Sorrensen Dongarra らによって提案されたパネル分割による Trailing Matrix の更新操作のブロック化 (行列–行列積によるアルゴリズム構成) が有名である．

LAPACK などでは WY 表現などの手法で複数の M の作用をまとめて Q の作用をさせる方法がとられている．WY 表現では Householder QR の Q を行列 Y, T を用いて、

$$Q = I - YTY^T \in \mathbf{R}^{(m \times m)}, \tag{3}$$
$$Y \in \mathbf{R}^{(m \times j)} \ (m > j),\ T \in \mathbf{R}^{(j \times j)}$$

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 とすると, Q の更新は以下のようになる．但し, $Y_0 = u_0, T_0 = 2$ とする．

$$\begin{aligned} Q_{k+1} &= I - Y_{k+1} T_{k+1} Y_{k+1}^T \\ Y_{k+1} &= [Y_k, u_k] \in R^{(m \times (j+1))} \\ T_{k+1} &= \begin{bmatrix} T_k & z_k \\ 0 & 2 \end{bmatrix},\ z_k = -2 T_k Y_k^T u_k \end{aligned} \tag{4}$$

$A = [a_1, a_2, \cdots, a_n]$ の前半のブロック部分を上記の Householder QR によって QR 分解を実施した後, 残りのブロック部分を A' とすると A' の更新は Q^T を A' に作用させ, A' を計算する．

$$A' := A' - Y(T^T(Y^T A')) \tag{5}$$

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, 高速な計算を行うことができる．

2.4 Tall Skiny QR 分解 (TSQR)

Tall Skinny QR (TSQR) は非常に縦長 ($m \gg n$) の行列 A に対して行列を繰り返し行方向に分割して QR 分解を行うアルゴリズムである．

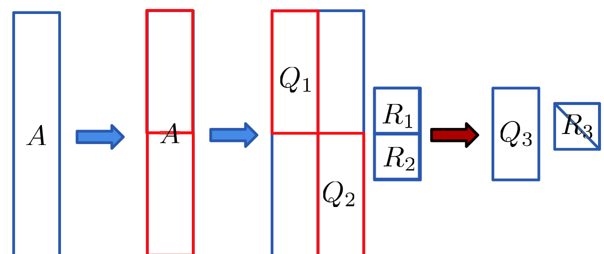

図 1 TSQR の最小単位 2 分割 QR

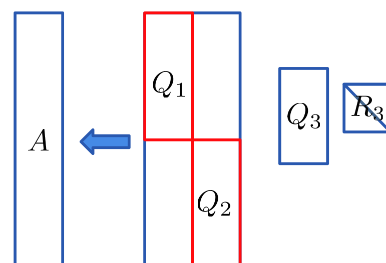

図 2 TSQR の 2 分割 QR における Q の合成方法

まず, 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 1 に示す．TSQR の流れとしては

(1) A を図 1 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．

(2) 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 ($[R_1; R_2] = Q_3 R_3$) を行う．

ここで得られた R_3 が最終的な R となる．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 の乗算を行う必要がある (図 2).

TSQR のアルゴリズムは図 3 のようになる．図 3 中で Householder 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．Q は 2 分木を構成する段階で計算できるため, ここで示す以外にも様々な実装がありうるが, 上向きのと下向きの sweep を 2 つに分けた例を図 3 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」, また「最終的な Q の計算」に対してなど, アルゴリズムそのものに自明で疎粒度のタスクを含んでいる．つまり, 並列計算に適しているといえるが, 計算量は従来の HouseholderQR よりも多くなる．

Divide A into $A = [A_1; A_2; \cdots; A_{2^d}]$.
for(i=1 to 2^d){
  (1) **do** HouseholderQR(A_i, $Q_i, R_i^{(d+1)}$)
    // 2.3 節の Block Householder QR などを用いる
}
for(k=d downto 1){
  for(i=1 to 2^{k-1}){ // this loop can be parallelized
    (2) **Concat** $B_i^{(k)} = \begin{bmatrix} R_{2i}^{(k+1)} \\ R_{2i+1}^{(k+1)} \end{bmatrix}$,
    and **do** Householder QR($B_i^{(k)}, Q_i^{(k)}, R_i^{(k)}$)
  }
}
for(k=2 to d){
  for(i=1 to 2^{k-1}){
    (3) $Q_{2i-1}^{(k)} = Q_{2i-1}^{(k)}$upper_half($Q_i^{(k-1)}$),
    $Q_{2i+0}^{(k)} = Q_{2i+0}^{(k)}$lower_half($Q_i^{(k-1)}$),
  }
}
(4) $Q := [Q_1 Q_1^{(d)}; Q_2 Q_2^{(d)}; \cdots; Q_{2^k} Q_{2^d}^{(d)}]$,
  and $R = R_1^{(1)}$

図 3 TSQR のアルゴリズム (2 分木による Q, R の集約構成操作の場合)

3. CUDA TSQR の実装

3.1 CUDA for Numerical Linear Algebra

CUDA 環境において最も強力な数値線形計算のツールは CUBLAS[6] である．ホスト側からカーネル関数を呼び出し, 関数機能の計算を完全にデバイス側に移し, CUBLAS の最適化された実装により高性能計算を実現することができる．CPU でも性能測定のベンチマークとして利用される DGEMM では, ピーク性能に対して 70% 以上を記録する (例として, Tesla K20c でピーク 1.17TFLOPS に対して 1.05TFLOPS を記録する.)．ほぼフルセットの BLAS の機能が提供されており, BLAS の関数呼び出し部分を CUBLAS のプレフィックスをつけるのみで GPU にオフロードする thunking mode も存在している．フルセットサポートではあるが全ての関数が最適化されているとは言い難く, 次に述べる MAGMA に性能面での歩がある関数も存在する．さらに, CUDA 5.0 以降のバージョンでは CUDA Capability 3 以上の GPU コアで, カーネル関数内からカーネル関数を呼び出すデバイス API が用意されている．この機能は, 後述の CUDA Dynmic Parallel Programming の機能に基づくため全ての世代に有効というわけでない．

Matrix Algebra on GPU and Multicore Architectures (MAGMA) [7] 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*1．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ため, GPU 単体よりも高速となることがある．MAGMA は BLAS 機能を実現する MAGMABLAS と LAPACK 相当の機能を実現する部分がある．

MAGMA ライブラリで倍精度 Householder QR 分解を行う関数は `magma_dgeqrf()` である．この関数では LAPACK 同様に Block Householder QR による QR 分解を行っており, 本研究で扱う GPU への TSQR の実装はない, 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．

その他の線形数値計算ライブラリとして CULA[8] や BLAS に特化した GLAS[9], [10], ASPEN.K2[11] なども存在している．いずれも, CUBLAS のデバイス API に相当する機能は提供されておらず, ホストである CPU 側からの呼び出しにより関数が動作する制御形態になっている．

3.2 CUDA-TSQR の設計と実装

3.2.1 オフロード範囲の選択

CUDA-TSQR の実装において, オフロード範囲と制御方式を明確に設計方針として打ち出しておく必要がある．TSQR アルゴリズムは図 3 にある様に, 大きく 4 段階に分けられるが第二段階と第三段階の Q, R の再帰的な構成部分を纏めて同時に実行すると大きく 3 部分で構成されることになる．

まず, 最も計算コスト高い第一段階の A_i の Householder QR はその自明な並列性からも, GPU にオフロードすべきである．次に, 計算コストが高く並列性も明らかな第三段階 (図 3 の (4) $Q_i Q_i^{(d)}$ の計算) も GPU に確実にオフロードすべきである．最後に, 第二段階の計算を現在の CUDA の枠

*1 ただし, 1.0 リリース以降は CUDA のみならず, OpenCL 向け clMAGMA や Intel Xeon Phi 向け MAGMA MIC のパッケージも開発されている．

© 2014 Information Processing Society of Japan

組みで実装するには, 計算タスクの割り振り, 同期, データアクセスに関する排他制御その他の議論が必要となる.

詳細の議論の前に, 第二段階の処理を CPU にさせる, CPU-GPU 連携方式, GPU に完全にオフロード方式について整理したい. 図 4 から 6 までについて説明する.

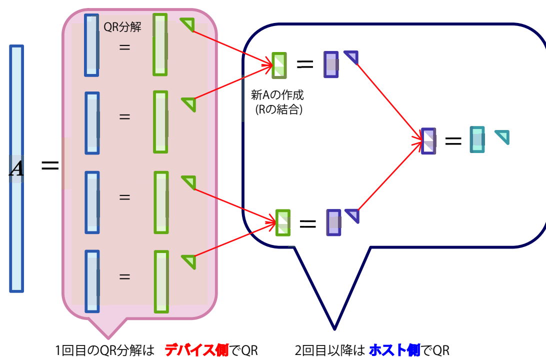

図 4 CPU+GPU の形態での TSQR の設計例

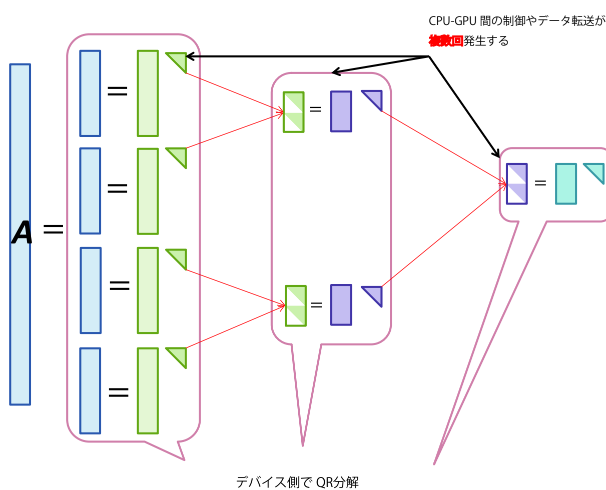

図 5 GPU のみで動作させる TSQR の設計例 (複数カーネル使用)

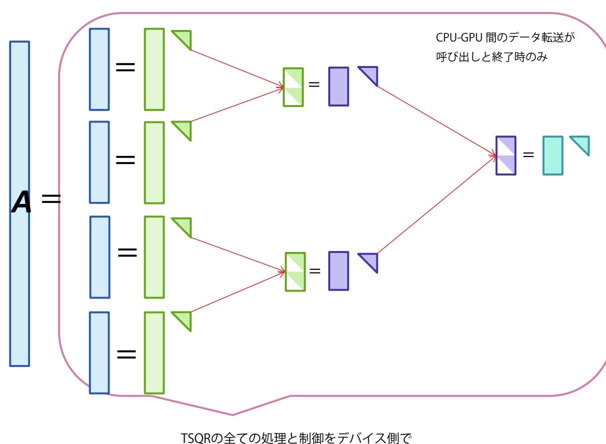

図 6 GPU のみで動作させる TSQR の設計例 (1 カーネル版)

図 4 は GPU での実装が最も困難な 2 分木操作以降を CPU 側で処理するものであり, 第一段階での Householder QR をブロックごとに割り当てて, 所謂 Ebarassingly Parallel で処理することに相当する. 各ブロックで処理された Householder QR の結果 Q, R をホストに返し, 2 分木操作により第二, 第三段階を CPU で処理をする.

図 5 は GPU に全てオフロードするが, 現在の CUDA の実行モデルでは 2 分木等の葉の集約の実行完了をブロック間で認識できない部分を, 一旦ホストに制御を移した上でシステムレベルで大きな同期を行うというものである. 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.

図 6 も GPU に完全オフロードするが, 1 カーネル関数で実現するモデルである. このモデルでは, カーネル関数が 1 つであるため, GPU に制御を移した後は CPU はその終了を待つのみであり, 理想的な完全オフロードの形態といえる. しかしながら, このモデルではブロック間同期の実現が前提になるため, 現時点での CUDA の実行モデルでは実装は困難といえる. カーネル関数呼び出し時に指定するブロック数などに一定の条件を与えた場合には, atomic 関数などを用いてブロック間同期の模擬は可能であるが, アクティブでないブロックが存在する場合に GPU がデッドロックする危険があるので注意が必要である.

MAGMA は Householder QR の前段階のパネル内の QR 分解を CPU 側で行い, 後半の Trailing Matrix の Level 3BLAS でのランク更新操作に GPU を使用している. 図 4 と似た構成といえるが, 実際にはパネル更新はその操作を反復しており, CPU-GPU 間の操作がある程度存在する. しかしながら, 非同期的な通信により実現が可能な部分もある. MAGMA では効果ははっきりしないがそういった非同期実装も試みられている.

本研究で開発する CUDA-TSQR は, 基本的に図 5 の実装を行う. 但し, GPU 上で二分木の集約操作はコストが大きいため, TSQR の数学的な性質を利用して, 全葉を 1 度に集約させ第二段階のカーネルは 1 回の操作にとどめる実装を採用する. この場合, 第二段階の処理は 1 ブロックで実行されコストは A の分割数に比例する. 第一, 第三段階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, 第二段階の影響によりトレードオフを考慮しなくてはならない.

3.2.2 Micro CUDA BLAS implementation

前節で CUBLAS がカーネル関数から呼び出すことのできるデバイス API をサポートしていることを述べた. しかし, この機能は CUDA5 以降の Dynamic Parallel Processing に基づくものである [5]. Kepler 世代以降に限定されるため, 本研究では CUBLAS のデバイス API を使用せず, カーネル関するから呼び出すことのできる BLAS 相当機能を実現するデバイス関数群 (Micro CUDA BLAS, MCBLAS と呼ぶ) を実装して対応した. デバイス関数としての実装であ

るので，カーネル関数同様の実装技術は応用できるが，大規模の問題を分解して多数のスレッドブロックにより並列処理するのとは異なる手法を取らなくてはならない．つまり，中規模の問題を一定数のスレッドで処理をするマルチコアCPU の戦略である．但し，共有メモリというプログラマブルなキャッシャ兼レジスタを有効活用することができる．

現時点では TSQR 実装に必要な以下の BLAS 関数を実装した．

- `dnorm`：ベクトルの 2 ノルム $a = \|x\|$
- `dgemvN`：行列 (非転置) ベクトル積 $y = \alpha Ax + \beta y$
- `dgemvT`：行列 (転置) ベクトル積 $y = \alpha A^T x + \beta y$
- `dger`：行列の 1 階更新 $A = A + \alpha xy^T$
- `dgemmNN`：行列行列積 $C = \alpha AB + \beta C$
- `dgemmTT`：行列行列積 $C = \alpha A^T B^T + \beta C$
- `dgemmNT`：行列行列積 $C = \alpha AB^T + \beta C$
- `dgemmTN`：行列行列積 $C = \alpha A^T B + \beta C$

MCBLAS の基本性能については，4 章で紹介する．CUDA-TSQR は Householder QR の実装方式については，LAPACK の `dgeqrf` を参考にして MCBLAS の呼び出しにより出来るだけコンパクトで LAPACK ソースコードとの親和性を保持する工夫をした．

4. 数値実験結果

4.1 Micro CUDA BLAS(MCBLAS) の基本性能

図 7 に Level 2 BLAS の dgemv(行列ベクトル積) 関数の性能を示す．性能測定は，GeForce GTX780 の 1 ブロックのみを使用した．図 8 には Level 3 BLAS の degmm(行列行列積) 関数の性能を示す．dgemvT の性能が低めであるが，1.1〜1.2GFLOPS 程度となっている．

現在の MCBLAS の実装では 1 ブロックあたり 64 スレッドを起動しているのみであるため，CUDA core を最大限に利用した実装とは言い難い[*2]． 次節の表 1 から判断するに，倍精度小数点計算の理論性能は約 165GFLOPS であるから，1MP あたりの性能に換算すると 165/12=13.75GFLOPS/MP となり 1/10 程度の性能しか出ていない．最適化の余地は十分にあることが分かる．dgemvN のスパイクなども共有メモリのバンクコンフリクトや各種メモリアクセスの問題が解決されていないことが明確である．

4.2 CUDA-TSQR の性能

図 9 と図 10 に今回開発した CUDA-TSQR の実行時間を示す．それぞれ，GeForce GTX560Ti と GeForce GTX780 を使用している．QR 分解の対象行列 A はグラフ中の行数 × 列数 64 固定の乱数行列である．いくつ行列 A を分割したかについて 32, 64 の 2 通りについて挙げている．図中の

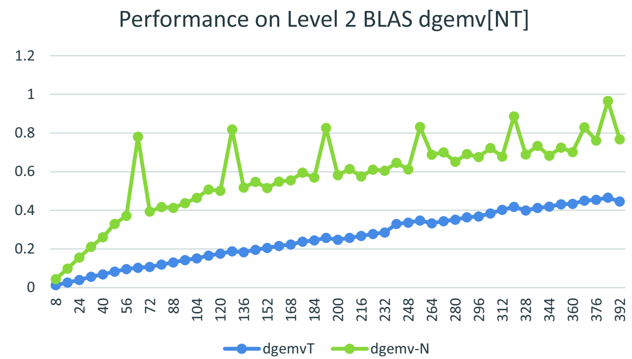

図 7 Micro CUDA BLAS の性能 (GFLOPS)，dgemv-[NT] on GTX780 (1 ブロックのみ使用)

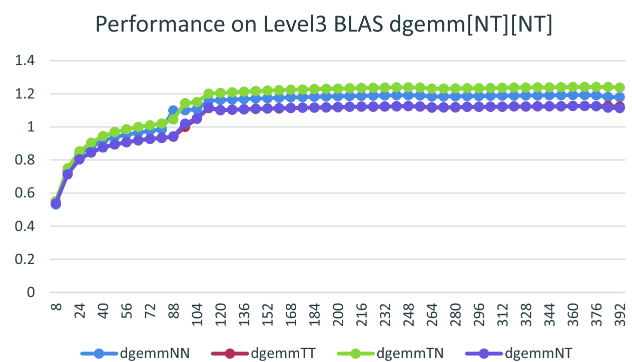

図 8 Micro CUDA BLAS の性能 (GFLOPS)，dgemm-[NT][NT] on GTX780 (1 ブロックのみ使用)

r_major や c_major は行列データの index 順序について 2 種類作成したものである．

GTX560Ti と GTX780 の簡単な性能諸言は次表 1 にあげておく．CUDA は 5.5 を使用し，ホスト側のコンパイラは gcc を使用している．CPU 側で使用する BLAS は ATLAS(version 3.8.4) を使用している．

比較のために，開発途中で作成した CPU 単独版など各種バージョンと MAGMA の `magma_dgeqrf` のテストコードの実行時間も併せてプロットしている．

CUDA-TSQR では計算が縦長のベクトル同士の内積計算があるため，C_major が有利であり，実装からもそれが読み取ることができる．Householder QR を使用する MAGMA との比較では，行数の小さな領域では CUDA-TSQR は大差をつけられているが，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 CUDA-TSQR が高速である．TSQR アルゴリズムはもともと Householder QR に比べて，資源間の頻繁な同期を必要としない疎粒度並列の解法である．それ故に，TSQR が HouseholderQR に対して高速であることは十分に納得ができるのであるが，最適化が十分とは言えない MCBLAS を使用した現実装においても CUDA-TSQR が高速であるということであり，MCBLAS の最適化が更に進めば CUDA-TSQR の優位性

*2 GTX780 は 192 コア/MP 構成である．

© 2014 Information Processing Society of Japan

が高まることになる.

表 1　GTX560Ti と GTX780 の性能諸言

	GTX560Ti	GTX780
コア世代	Fermi	Kepler
CUDA Capability	2.1	3.5
CUDA コア数	384	2304
MP 数	8	12
倍精度理論性能	150	165

5. まとめ

本研究では, 近年提唱されている TSQR に対して CPU 資源を極力使わず, 主たる計算部分を GPU に担当させる完全オフロード実装に関する研究を行った. 現時点では TSQR の第二段階を限定した実装を完成させたにとどまるが, 数値実験の速報的な数値から判断すると既存の GPU 数値計算ライブラリの代表格である MAGMA と比較しても, 高速化するケースがあり, TSQR の並列性と GPU の高い処理能力が立証されたといえる. 今後は MCBLAS の高速化や 1 カーネル関数版などの実装も含めて第二段階の二分木アルゴリズムの実装にも取り組み必要があると考えている.

最後に本研究は新学術領域研究 (課題番号 : 22104003) の支援を受けている.

参考文献

[1] James Demmel, Laura Grigori,Mark Hoemmen, and Julien Langou, Communication-avoiding parallel and sequential QR factorizations, CoRR abs/0806.2159. 2008
[2] Anderson M., Ballard, G., Demmel J., and Keutzer, K., Communication-Avoiding QR Decomposition for GPUs, Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International. IEEE, 2011.
[3] 森大介, 山本有作, 張紹良. ハウスホルダー QR 分解のための AllReduce アルゴリズムの性能と精度, 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] Vol.2008(99), pp.25-29, 2008.
[4] 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, 情報処理学会論文誌コンピューティングシステム, Vol. 2(2), pp.19-29, 2009.
[5] 例えば, NVIDIA: whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf
[6] NVIDIA : CUDA CUBLAS Library. http://developer.download.nvidia.com
[7] Agullo, E., Demmel, J., et al. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects, J. of Physics: Conference Series 180 (2009)
[8] Humphrey, J. R., Price, D. K., et al., CULA: Hybrid GPU Accelerated Linear Algebra Routines, SPIE Defense and Security Symposium (DSS) (2010)
[9] Sorensen, H. H. B., Auto-tuning Dense Vector and Matrix-Vector Operations for Fermi GPUs, Parallel Processing and Applied Mathematics, Lecture Notes in Computer Science, Vol. 7203 (2012) 619–629
[10] GPUlab: GLAS library version 0.0.2, http://gpulab.imm.dtu.dk/docs/glas_v0.0.2_C2050_cuda 4.0_linux.tar.gz
[11] Imamura, T., ASPEN-K2: Automatic-tuning and Stabilization for the Performance of CUDA BLAS Level 2 Kernels, 15th SIAM Conference on Parallel Processing for Scientific Computing (PP2012), http://www.siam.org/meetings/pp12/
[12] NVIDIA, ホワイトペーパー, NVIDIA の次世代型 CUDA コンピュート・アーキテクチャ Kepler GK110, http://www.nvidia.co.jp/content/apac/pdf/tesla/nvidia-kepler-gk110-architecture-whitepaper-jp.pdf

© 2014 Information Processing Society of Japan

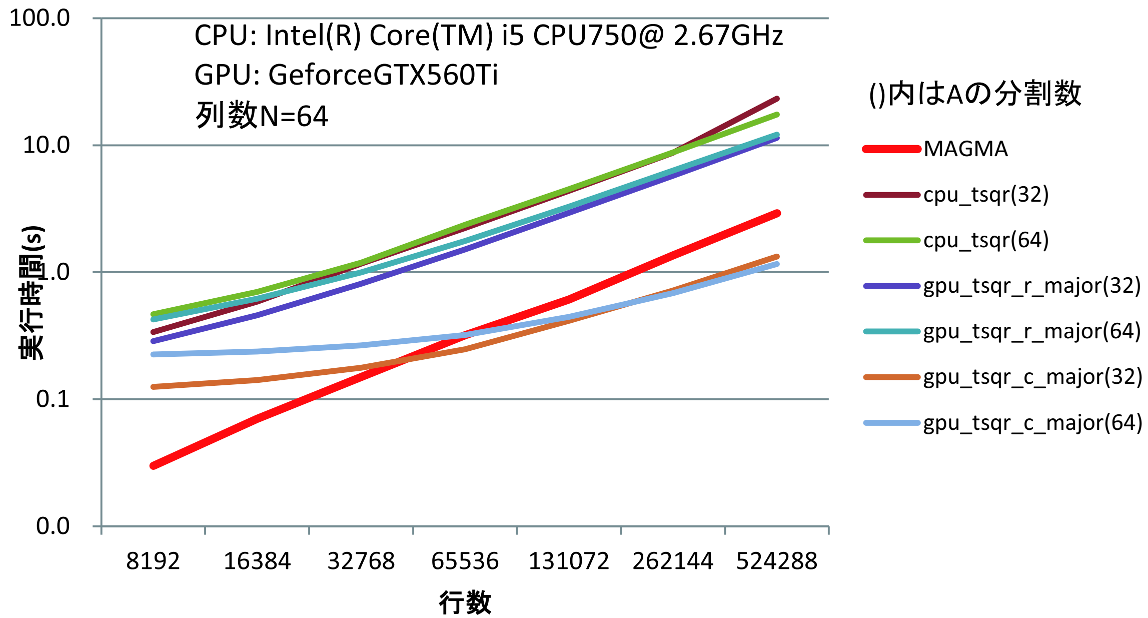

図 **9** Numerical Results of fully-offloaded CUDA-TSQR on a GTX560Ti (elapsed time)

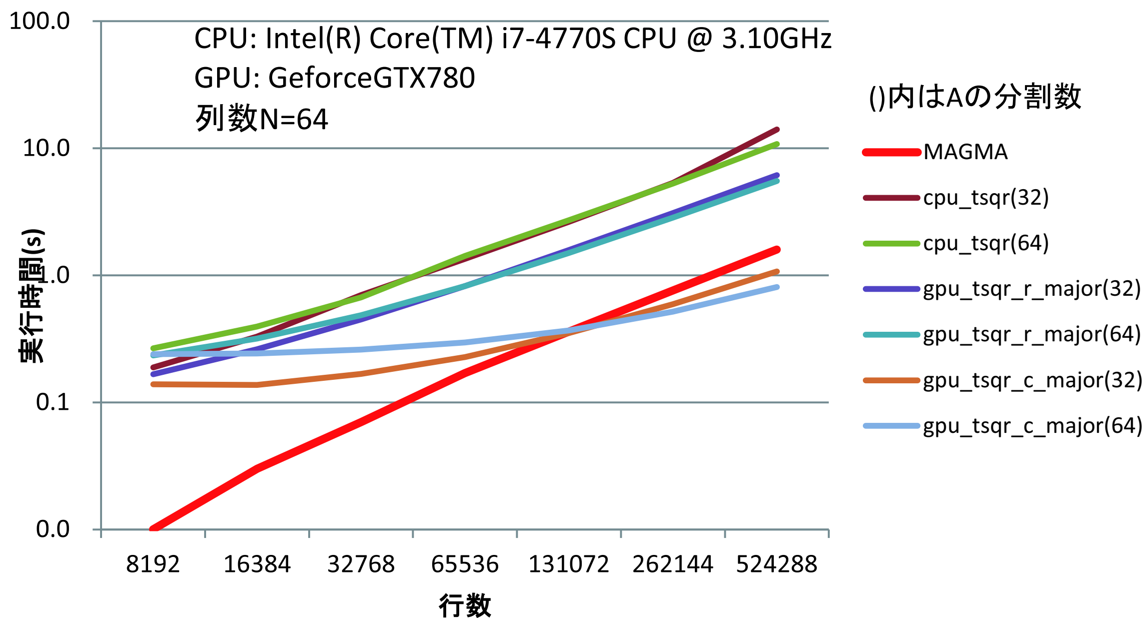

図 **10** Numerical Results of fully-offloaded CUDA-TSQR on a GTX780 (elapsed time)

© 2014 Information Processing Society of Japan