Vol.2010-HPC-126 No.2
2010/8/3

非対称固有値計算におけるヘッセンベルグ化のGPU による高速化

村松淳一[†1]　山本有作[†2]　張紹良[†1]

近年，非対称行列の固有値問題に対する高速化の需要がある．非対称固有値問題の応用例として，電子回折像の解析や自動車の振動解析などがあり，これらの応用例では，すべての固有値・固有ベクトルを求める必要があり，計算をより高速化することが求められている．本研究では，GPU の特長を生かし，非対称行列の固有値計算，特にその中のヘッセンベルグ化に注目し，GPU を用いることで高速化を図ることを目的とする．

Acceleration of the Hessenberg Reduction for Nonsymmetric Eigenvalue Problems using GPU

Junichi Muramatsu,[†1] Yusaku Yamamoto[†2]
and Shao-Liang Zhang[†1]

Recently, there is a growing demand for solving nonsymmetric eigenvalue problems. Applications of nonsymmetric eigenvalue problems include vibration analysis of cars and analysis of electronic diffraction images. In these applications, all the eigenvalues and eigenvectors of a matrix are needed and speedup of the computation is strongly required. In this study, we focus on the Hessenberg reduction step and show how this step can be accelerated using GPU.

†1 名古屋大学大学院工学研究科計算理工学専攻
Department of Computational Science and Engineering, Graduate School of Engineering, Nagoya University
†2 神戸大学大学院システム情報学研究科計算科学専攻
Department of Computational Science, Graduate School of System Informatics, Kobe University

1. はじめに

近年，非対称行列の固有値問題 $A\boldsymbol{x} = \lambda\boldsymbol{x}$ に対する高速化の需要がある．非対称固有値問題の応用例として，電子回折像の解析や自動車の振動解析などがあり，これらの応用例では，すべての固有値・固有ベクトルを求める必要がある．自動車の振動解析の問題では，12000 元の問題で 8PU を用いて解いても 7 時間程度の時間がかかってしまう．そのため，計算をより高速化することが求められている．

GPU(Graphics Processing Unit) の高速な演算性能とメモリ転送速度を，グラフィックス演算に限らず汎用の数値計算に活用する GPGPU(General Purpose GPU) が，近年注目を集めている．従来 GPU を用いたプログラミングには，グラフィックス API やストリーム言語など特殊なプログラミングが必要であり，一般のプログラム開発者には敷居の高いものとなっていたが，2006 年に NVIDIA 社から発表された，GPGPU のための統合開発環境 CUDA により，標準の C 言語の簡単な拡張によって NVIDIA 社の GPU を汎用計算に利用できるようになった．また，チューニング済みの BLAS ライブラリなども提供されているため，GPU を用いた行列計算のプログラミングが非常に容易になった．

本研究では，GPU の特長を生かし，非対称行列の固有値計算に GPU を用いることで高速化を図ることを目的とする．

2. 非対称行列の固有値計算アルゴリズム

2.1 非対称行列の固有値計算の手順

非対称行列の固有値計算は，一般的に以下のステップに従って行われる．

Step1: ヘッセンベルグ化　$Q_{n-2}^T \cdots Q_2^T Q_1^T A Q_1 Q_2 \cdots Q_{n-2} = H$
　(ここで，H:ヘッセンベルグ行列，Q:直交行列)

Step2: 直交変換の蓄積　$Q_1 \cdots Q_{n-1} Q_{n-2} = Q$

Step3: Schur 分解　$P_n^T \cdots P_2^T P_1^T H P_1 P_2 \cdots P_n = T$ $P_1 P_2 \cdots P_n = P$
　(ここで，T: 上三角行列，P: 直交行列)

Step4: 固有ベクトル計算 $T \rightarrow \boldsymbol{x}$(T の固有値の計算), $\boldsymbol{x} \leftarrow QP\boldsymbol{x}$(逆変換)

ここで，ヘッセンベルグ行列とは，上三角行列の副対角要素が非零となっている行列である．

また，Step3 で上三角行列への相似変換を行うと，得られた T の対角要素が A の固有値となる．

実行時間の例として，$n = 4800$ の場合，計算時間は図 1 のようになる．

Vol.2010-HPC-126 No.2
2010/8/3

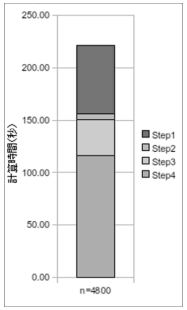

図 1 計算時間のグラフ

から，Step1 のヘッセンベルグ化と Step4 の固有ベクトル計算が多くの時間を占めている．ただ，Step4 に関しては並列化など他の研究により，高速化の見込みが立っている．また，Step1 は計算の構造が単純であるため，単純な演算を得意とする GPU に演算させるメリットが大きい．本研究では，Step1 の高速化を目的とする．

以下では，Step1 のヘッセンベルグ化について詳しく記述する．

2.2 ヘッセンベルグ化

与えられた $n \times n$ 非対称行列 A に，左右から適当な相似変換 $Q_i (i = 1, \cdots, n-2)$ を順に左右に作用させ，行列 A を

$$Q_{n-2}^T Q_{n-1}^T \cdots Q_1^T A Q_1 \cdots Q_{n-1} Q_{n-2} = \begin{pmatrix} * & * & * & \cdots & * & * & * \\ * & * & * & \cdots & * & * & * \\ 0 & * & * & \cdots & * & * & * \\ 0 & 0 & * & \cdots & * & * & * \\ 0 & 0 & 0 & \cdots & * & * & * \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & * & * \end{pmatrix}$$

で表わされるヘッセンベルグ行列にする．A の第 $k (k = 1, \cdots, n-2)$ 列目から，適当な $\boldsymbol{u}$ を生成して $Q_i = I - \boldsymbol{u}\boldsymbol{u}^T$ というハウスホルダー変換を生成した後，A に左右から Q_i を作用させる．これを第 1 列目から順に第 $n-2$ 列まで順に行う．

以下では，まず相似変換として用いるハウスホルダー変換について述べた後，ハウスホルダー変換による列の消去，ヘッセンベルグ化のアルゴリズムについて述べる．

2.3 ハウスホルダー変換

$\boldsymbol{x}$ と $\boldsymbol{y}$ を相異なる与えられたベクトルとする．これらは

$$||\boldsymbol{x}||_2 = ||\boldsymbol{y}||_2$$

を満たしているものとする．このとき，

$$(I - \boldsymbol{u}\boldsymbol{u}^T)\boldsymbol{x} = \boldsymbol{y}, ||\boldsymbol{u}||_2 = \sqrt{2}$$

を満足するベクトル $\boldsymbol{u}$ が存在して，それは符号は別にして一意的に

$$\boldsymbol{u} = \frac{\boldsymbol{x} - \boldsymbol{y}}{||\boldsymbol{x} - \boldsymbol{y}||_2} \times \sqrt{2}$$

によって与えられる．

2.4 ハウスホルダー変換を用いたヘッセンベルグ化

ハウスホルダー変換では，

$$Q\boldsymbol{x} = (I - \boldsymbol{u}\boldsymbol{u}^T) \begin{pmatrix} * \\ * \\ \vdots \\ * \end{pmatrix} = \begin{pmatrix} * \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

となるような，行列 Q（実際に計算する際は $||\boldsymbol{u}||_2 = \sqrt{2}$ を満たすベクトル $\boldsymbol{u}$）を求め，これを行列 A に左右から繰り返しかけることで，順番に列を消去していく．

2.5 ヘッセンベルグ化のアルゴリズム

ヘッセンベルグ化のアルゴリズムは以下のようになっている

for $k = 1$ to n-2 **do**
  (A の第 k 列から $\boldsymbol{u}$ を作成)
    ↓
  (左から Q_i を作用)
  $\boldsymbol{v}^T := \boldsymbol{u}^T * A[k+1:n, k+1:n]$
  $A[k+1:n, k+1, n] := A[k+1:n, k+1, n] - \boldsymbol{u} * \boldsymbol{v}^T$
  (右から Q_i を作用)
  $\boldsymbol{v} := \boldsymbol{u} * A[1:n, k+1:n]$
  $A[1:n, k+1, n] := A[1:n, k+1, n] - \boldsymbol{v} * \boldsymbol{u}^T$
end for

この計算の演算パターンは，左右からハウスホルダー変換を作用させる計算が行列ベクトル積と行列の rank-1 更新で行われる．行列ベクトル積や行列の rank-1 更新などの計算は，

データアクセス回数 (行列データ) が $O(n)$，演算量が $O(n)$ と，データアクセス回数と演算量が同程度のオーダーになる．一方，最近の計算機では演算速度よりメモリアクセス速度がはるかに遅いため，メモリアクセスが性能ネックになってしまう．

2.6 ヘッセンベルグ化のブロック化

ブロック化により，複数のハウスホルダー変換を

$$H_l \cdots H_2 H_1 = (I - 2\boldsymbol{u}_l \boldsymbol{u}_l^T) \cdots (I - 2\boldsymbol{u}_2 \boldsymbol{u}_2^T)(I - 2\boldsymbol{u}_l \boldsymbol{u}_l^T)$$
$$= I - VTV^T$$

と合成することができハウスホルダー変換の作用を行列ベクトル積 (データ:$O(n^2)$，演算量:$O(n^2)$) の演算をより演算効率のいい行列積 (データ:$O(n^2)$，演算量:$O(n^3)$) で行うことができる．実際にヘッセンベルグ化を行う LAPACK のルーチン DGEHRD ではブロック化で計算を行っている．

for $k = 1$ to N, NB **do**
(A の第 k 列〜第 $(k + NB)$ 列から V, T を作成)
↓
(左から Q_i を作用)
$Y := T * V' * A$
$A := A - V * Y$
(右から Q_i を作用)
$Y := A * V * T$
$A := A - Y * V'$
end for

以上がブロック化のアルゴリズムの概略である．

ブロック化したアルゴリズムでは，左右からハウスホルダー変換を作用させる部分に関しては，行列乗算を用いて行うことができる．しかし，A の第 k 列〜第 $(k + NB)$ 列から V, T を作成する部分において行列ベクトル積の計算が多く用いられ，ブロック化しても行列ベクトル積の問題が残ってしまう．

3. GPU におけるプログラム環境

近年，GPU(Graphics Processing Unit) の高速な演算性能とメモリ転送速度を，グラフィックス演算に限らず汎用の数値計算に活用する GPGPU(General Purpose GPU) が注目を集め，HPC（ハイパフォーマンスコンピューティング）のための GPU も製作されるようになった．以下では，GPU を汎用の計算に活用するために用いる環境について述べる．

3.1 CUDA の概要

CUDA（Compute Unified Device Architecture）とは，NVIDIA が提供する GPU 向けの C 言語の統合開発環境であり，コンパイラやライブラリなどから構成されている．CUDA により，汎用の数値計算が C 言語の簡単な拡張を用いて行えるようになっている．

CUDA 環境で GPU を用いて行列演算を行う方法としては，CUDA の BLAS(CUBLAS) を使用する方法と，プログラム全体を移植して nvcc コンパイラを使用する方法の 2 通りが考えられる．nvcc コンパイラは拡張された C 言語のソースコードから CPU と GPU で実行なバイナリファイルを生成する．従来のプログラムを GPU に使用するように拡張することで，複雑な計算にも GPU の使用が可能となる．しかし，GPU の性能を十分に引き出すためには様々な工夫が必要となるため，従来の線形計算プログラムを GPU の性能を引き出せる形で拡張することは容易ではない．

一方，CUBLAS は除算や平方根等一部の演算が行えないものの，BLAS で行える計算に関しては，倍精度実数であればその多くが GPU 向けに最適化されている．

本研究では，CUBLAS を用いて計算を行う．

3.2 CUBLAS の概要

CUBLAS ライブラリとは，BLAS (Basic Linear Algebra Subprograms; ベクトルと行列に関する基本演算ルーチン群) を CUDA 上で動作するように実装したライブラリである．

図**??**に GPU と CPU の概略図を示す．CUBLAS は，まずあらかじめ確保した GPU 上のメモリに計算にメインメモリから必要なデータを転送し，次に計算を GPU を用いて行い，最後に計算結果を GPU 上のメモリからメインメモリに転送するという形で使用する．

基本的に BLAS で構成されているアルゴリズムに関しては，BLAS の部分を CUBLAS に書き換えることにより，プログラムの構造を大幅に変えることなく GPU を使用することができる．

4. GPU を用いたヘッセンベルグ化の実装

本研究では，行列のヘッセンベルグ化に関する部分を GPU で計算することにする．具体的には，ヘッセンベルグ化のサブプログラム (DGEHRD) を CUDA 上で動作するように書き換える．

4.1 実装の方針

CUBLAS では GPU 向けに行列ベクトル積・行列の rank-1 更新・行列席などの行列演算が最適化されており，GPU 上で高速にこれらの計算を行うことができる．しかし，除算・平方根など CUBLAS で定義されていない一部の計算は実行できない．また，CPU-GPU 間のデータ転送速度は，CPU のメモリアクセス速度と比較しても数倍程度遅い．以上のことから，GPU で性能を出すには，次の点に注意して実相を行うことが必要である．

- CPU-GPU 間の転送を最小化する (基本的に GPU 上ですべての演算を行う)
- CUBLAS で実行できない演算に関しては，CPU に一旦データを送り，CPU で演算を行ってから GPU に戻す

以下では，この 2 点を考慮して CUBLAS を用いたヘッセンベルグ化の実装を考える．

4.2 単純な実装方式とその問題点

最も単純な実装方式として，計算の開始時に行列データを CPU から GPU に送り，GPU 上でヘッセンベルグ化を行い，最終的な結果を CPU に戻す方式が考えられる．

この方式の概略図を図 2 に示す．

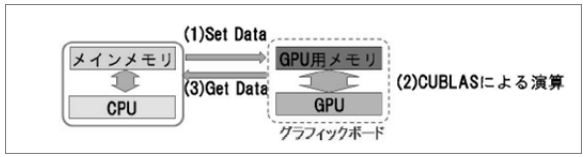

図 2 単純な実装方式

この方法では，ヘッセンベルグ化の演算の大部分を占める，左右からハウスホルダー変換を作用させる部分に関しては，行列乗算を用いて行うことができるため，CUBLAS の DGEMM を用いて高速に計算することができる．しかし，$\boldsymbol{u}$ を計算する部分に関しては，除算や平方根が含まれており，CUBLAS では実行できない．

4.3 本研究で用いる実装方式

前節の問題点から，CUBLAS では行えない演算は，CPU に一旦データを送り，CPU で演算を行ってから GPU に戻すことにする．

この方式の概略図を図 3 に示す．

この方法の特徴として，

- 最初と最後にデータの転送

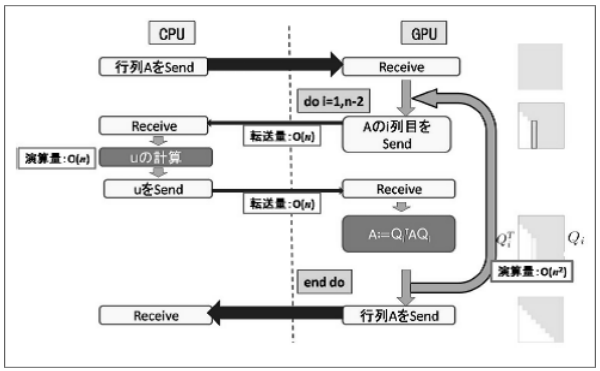

図 3 実際の実装方式

- ループ毎各一回，$\boldsymbol{u}$ の計算のために CPU へデータを転送する必要がある
- 行列を作用させる部分は CUBLAS で計算

という点が挙げられる．

ループ 1 回あたりの，$\boldsymbol{u}$ の計算におけるデータ転送・演算は $O(n)$，行列を左右から作用させる部分の演算量は $O(n^2)$ である．$\boldsymbol{u}$ の計算におけるデータ転送がネックとなるが，n が十分大きくなり，n に比べて n^2 が十分大きくなれば，CUBLAS による行列乗算の占める割合が大きくなり，高速化が見込めると考えられる．

5. 性能評価

5.1 数値実験の概要

前節までの内容に基づいて，CPU と GPU を併用してヘッセンベルグ化を行うプログラムを作成し，性能評価を行った．非対称行列 $A(n = 800)$ の全固有値・固有ベクトルを，LAPACK ルーチンを用いて CPU でヘッセンベルグ化を行うもの (以下，CPU のみ) と，CPU と GPU を併用してヘッセンベルグ化を行うもの (以下，CPU+GPU) の 2 通りで，倍精度実数の精度で計算した．CPU+GPU のプログラムは，LAPACK の DGEHRD をベースとし，これを C 言語で書きなおしてから，CPU-GPU 間の転送を付加し，BLAS を CUBLAS に置き換えて作成した．

評価に用いた行列 A は，各要素 [0,1) の乱数行列，サイズは $n = 800$,1600,2400,3200,4000,4800 の 6 種類，ブロック化におけるブロックサイズ $NB = 32$ で固定とした．

なお，計算機環境は以下の通りである．CPU は 4 コア全て使用して計算を行った．

項目	条件
CPU	Core i7 920 (2.66 GHz) [4 cores]
CPU Memory	6.0GB
コンパイラ	gcc ver.4.1.2 (オプション -O3) Intel Fortran コンパイラ ver.9.1
GPU	Tesla C1060
GPU Memory	4.0GB
CUDA version	2.0

表 1 計算機環境

5.2 固有値計算全体

非対称行列 $A(n = 800$ の倍数) の全固有値・固有ベクトルを，LAPACK ルーチンを用いて CPU でヘッセンベルグ化を行うもの (以下，CPU のみ) と，CPU と GPU を併用してヘッセンベルグ化を行うもの (以下，CPU+GPU) の計算時間を比較した．
なお，

$$(\text{高速化率}) = \frac{(\text{CPU のみの計算時間})}{(\text{CPU+GPU の計算時間})}$$

である．

各 n に関する実行時間 (秒) と高速化率を表 2 に示す．

n	800	1600	2400	3200	4000	4800
CPU 4 cores	0.14	1.53	5.25	12.70	25.07	40.47
CPU+GPU	1.37	2.74	5.08	8.69	13.58	20.12
高速化率	0.1	0.56	1.03	1.46	1.85	2.01

表 2 ヘッセンベルグ化の計算時間

$n = 800$ では性能が出ず，CPU+GPU は CPU のみより遅くなってしまった．これは，n が n^2 に比べて十分に小さくなく，$\boldsymbol{u}$ の計算における CPU への転送コストの全体に占める割合が非常に大きくなってしまったためと考えられる．一方，n が大きくなると高速化率が上がり，$n = 4800$ では 2.01 倍の高速化率を得た．これはデータ転送コスト $O(n)$ が演算コスト $O(n^2)$ に比べて十分に小さくなり，ハウスホルダー変換の作用の行列乗算の部分で得られる利得が，CPU への転送コストと比較して十分大きくなったためと考えられる．

5.3 固有値計算全体の計算時間

固有値計算の各ステップの計算時間を，表 3 に結果を示す．なおここでは Step2〜Step4 は LAPACK のルーチンを用いて，CPU で計算している．

各ステップの計算時間

n	800	1600	2400	3200	4000	4800
Step1(CPU のみ)	0.14	1.53	5.25	12.70	25.07	40.47
Step1(CPU+GPU)	1.37	2.75	5.08	8.70	13.58	20.12
Step2	0.04	0.28	0.79	1.75	3.23	5.34
Step3	0.88	3.13	7.75	14.18	22.54	33.87
Step4	0.32	4.07	14.60	35.70	70.61	116.59

固有値計算全体の計算時間

全体 (CPU のみ)	1.38	9.01	28.4	64.33	121.44	196.28
全体 (CPU+GPU)	2.61	10.23	28.23	60.34	109.96	175.93
高速化率	0.53	0.88	1.01	1.07	1.1	1.12

表 3 固有値計算全体の計算時間

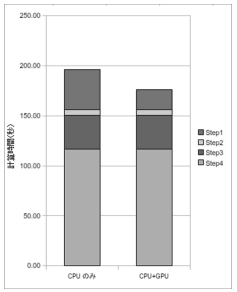

図 4 固有値計算全体の計算時間 ($n = 7200$)

Step1 を GPU で行うことにより，この部分が全体に占める割合は小さくなった．特に，$n = 4800$ では，固有値・固有値計算全体において 1.12 倍の高速化率を得た．

5.4 ルーチンの演算性能

ヘッセンベルグ化を行う LAPACK のコード (DGEHRD) で行われている中で，最も大きな演算である行列積のルーチン (DGEMM) と行列ベクトル積のルーチン (DGEMV) の GPU による高速化の効果を確かめるため，DGEMM,DGEMV 単独での性能と，DGEHRD

の内部の DGEMM,DGEMV の性能を比較した．

なお，パラメータ n に関して，DGEMM,DGEMV 単独に関しては，それぞれ $n \times n$ 行列と $n \times n$ 行列の積，$n \times n$ 行列と n 次元ベクトルの積に対応している．

表 4 に結果を示す．ここで，DGEMM，DGEMV の単独性能とは，それぞれ $n \times n$ 行列と $n \times n$ 行列の積，$n \times n$ 行列と n 次元ベクトルの積を行った場合の性能を意味する．また，ルーチン内性能とは，入力行列のサイズが n であるとき，DGEHRD でコールする DGEMM（あるいは DGEMV）の演算量の総計を求め，それを DGEMM の実行時間の合計で割って得られる性能を意味する．性能の単位は GFLOPS である．

単独性能 (GFLOPS)

DGEMM

n	800	1600	2400	3200	4000	4800
GPU	65.14	74.38	66.51	74.88	66.62	75.11
CPU	16.92	15.64	36.67	36.39	40.04	39.53
高速化率	3.85	4.76	1.81	2.06	1.66	1.90

DGEMV

n	800	1600	2400	3200	4000	4800
GPU	2.77	5.85	8.98	11.78	14.13	16.46
CPU	0.61	1.38	1.86	2.02	2.15	2.35
高速化率	4.55	4.23	4.84	5.82	6.59	7.00

ルーチン内性能 (GFLOPS)

DGEMM

n	800	1600	2400	3200	4000	4800
GPU	36.43	43.87	47.16	47.53	49.30	49.9
CPU	29.49	24.91	24.02	23.84	24.33	24.81
高速化率	1.24	1.76	1.96	1.99	2.03	2.01

DGEMV

n	800	1600	2400	3200	4000	4800
GPU	2.832	5.98	9.08	11.93	14.18	16.52
CPU	5.353	2.80	2.59	2.47	2.42	2.57
高速化率	0.53	2.13	3.50	4.82	5.86	6.43

表 4 ルーチンの性能比較

2 つのルーチンの両方とも，GPU による高速化の効果が見られる．特に DGEMV に関しては，行列・ベクトルのサイズが大きければ大幅な高速化が得られた．しかし，DGEMM に関しては，CPU(4 コア) と比較しても高々2 倍程度の性能差しか得られなかった．そのため，CPU の DGEMM 性能を活かすように負荷分散を見直すことにより，さらに性能向上が期待できる．

6. おわりに

非対称行列の固有値問題の解法のステップの一つであるヘッセンベルグ化に GPU を適用し，高速化を図った．GPGPU のための開発環境 CUDA 及び CUBLAS を用いることで，プログラムの構造を大きく変更することなく，GPU の高い演算性能を利用することができた．

今後の課題としては，ヘッセンベルグ化 (Step1) のプログラムの改良（CPU と GPU の負荷分散を考慮したアルゴリズム），固有ベクトル計算の部分 (Step4) の GPU による高速化，実問題への適用などが挙げられる．

謝　辞

日頃からご指導頂いている京都大学大学院情報学研究科の中村佳正教授に感謝いたします．なお，本研究は科学研究費補助金の補助を受けている．

参 考 文 献

1) CUDA ZONE: http://www.nvidia.co.jp
2) 森正武: 数値解析, 共立出版株式会社, 1973.
3) 深谷猛, 山本有作, 畝山多加志, 中村佳正; 正方行列向け特異値分解の CUDA による高速化, 情報処理学会論文誌. ACS, Vol. 2, No. 1
4) G.W.Stewart: *Matrix Algorithms Volume II: Eigensystems*, SIAM, Philadelphia, 1998