國立臺灣大學理學院應用數學科學研究所
碩士論文
Institute of Applied Mathematical Sciences
College of Science
National Taiwan University
Master Thesis

在叢集伺服器上高效能多維奇異值分解及基於閉曲線積分的特徵值分解

Efficient Higher-Order Singular Value Decomposition and Contour Integral Based Eigen Decomposition on CPU/GPU Cluster

蔡宇翔
Yu-Hsiang Tsai

指導教授：王偉仲博士
Advisor: Weichung Wang, Ph.D.

中華民國 107 年 6 月
June, 2018

doi:10.6342/NTU201803532

doi:10.6342/NTU201803532

國立臺灣大學碩士學位論文
口試委員會審定書

在叢集伺服器上高效能多維奇異值分解及基於閉曲線積分的特徵值分解

Efficient Higher-Order Singular Value Decomposition and Contour Integral Based Eigen Decomposition on CPU/GPU Cluster

本論文係蔡宇翔君 (R05246003) 在國立臺灣大學應用數學科學研究所完成之碩士學位論文，於民國 107 年 6 月 12 日承下列考試委員審查通過及口試及格，特此證明

口試委員：

_____

_____ _____

_____ _____

_____ _____

所　長：_____

doi:10.6342/NTU201803532

doi:10.6342/NTU201803532

誌謝

首先要感謝我的指導教授王偉仲教授，在做研究的途中，老師給予我許多的幫助，例如：協助找到經費讓我能夠出國至國外一流的機構，去 IBM Thomas J. Watson Research Center 拜訪，去東京大學與那邊的教授和學生互相交流，去 Karlsruhe Institue of Technology 做研究，也讓我能夠去國際會議 SuperComputing 2017 (SC17) 進行報告；另外也帶給我許多機會，像是之前老師與 IBM 的研究人員談完之後，把這個機會介紹給我，而也因此完成了此碩論的第一部份，而當老師出去聽演講或跟別的教授會談之後，常常會試著把專題與別人的應用做連結，讓所做的專題有更廣的應用，像是第二部分的應用部分就是這樣來的。十分感謝老師提供給我如此許多的資源跟機會學習。感謝 Jeewhan Choi 博士、Xing Liu 博士、Takeo Hoshi 教授，在研究上給了我許多的幫助，讓我在過程中也學習到很多東西。要感謝研究所的同學們，大家時常保持實驗室的氣氛良好，在有遇到困難時也會願意協助，讓許多事情都能夠順順利利完成，在討論時大家也會勇於提出自己的意見與他人交流，保持著很好的學習風氣。要特別感謝林東成和陳彥禎的協助，常常被我拖著檢查我英文的問題，也常常在溝通之後才發現原來我寫的跟我想表達的常常是兩回事，感謝他們的協助讓我這篇說論能夠順利在時限前完成。最後，謝謝我的家人與朋友的協助，讓我可以無後顧之憂的投入

doi:10.6342/NTU201803532

doi:10.6342/NTU201803532

Acknowledgements

First, I am glad to thank my advisor Prof. Weichung Wang. He always gives me a lot of resources and connections. He helps me to find some funding to support me to go aboard to visit the top research place. For example, visit IBM Thomas J. Watson Research Center, discuss projects with the students and professors of Tokyo University, do some researches in Karlsruhe Institue of Technology, and attend conference SuperComputing 2017 (SC17) to present the student poster. When he attends the speeches and meetings, he always gives me a chance to work with the top researcher or connect my project to different sides. I work with Dr. Jeewhan Choi and Dr. Xing Liu in IBM to finish the first part of my thesis, and also present it on SC17. Prof. Wang also brings the application of the second part of my thesis. He connects my algorithm and Prof. Takeo Hoshi's application. Thanks a lot for the massive help of Prof. Wang. Thanks to Dr. Choi, Dr. Liu and Prof. Hoshi. They also give many advisements to me, and I also learn a lot from them. Thanks to classmates of the same laboratory. When I face some problem, they are always enthusiastic to help me. When discussing the paper or something else, they give their comment. I would like to thank Dung-Cheng Lin and Yen-Chen Chen. They help a lot with my English problem. After discussing some paragraph of the thesis, I just discovered the different meaning between what I think and what I write. Due to their help, I can finish the thesis on time. Finally, I'm glad to thank my family and friends. They allow I can involve the thesis without any problems.

doi:10.6342/NTU201803532

doi:10.6342/NTU201803532

摘要

我們改進了“多維奇異值分解”及“基於閉曲線積分的特徵值分解”。

在“多維奇異值分解”中，我們使用不同的方式實作兩個演算法，並提升解決巨量資料的能力。隨著資料越來越多樣，找到方法去快速且有效得壓縮和分析資料中的多角關係對於高效能計算中是非常重要的。我們實做了兩個已經存在的演算法：多維奇異值分解及逐步降維多維奇異值分解將多維矩陣分解，利用 GPU 來加速他們是非常困難的，因為我們幾乎無法把資料一次放進 GPU 的記憶體當中。所以我們利用 QR 和 Gram 來協助我們縮減問題的大小，而我們實作 QR 和 Gram 是將資料分成一部份一部份來解決，這樣不但能讓 GPU 有能力處理，還能利用計算的部分遮蓋掉資料搬移的時間，最後我們相對於原先利用 cuda 函式庫時做的能加速 163.21 倍，在未來希望能夠將這個方法使用在實際問題上。

在“基於閉曲線積分的特徵值分解”中，我們藉由基於閉曲線積分的特徵值分解，展示了一個分治法的處理方式去解決區域中有太多的特徵對的問題，並應用它在有機材料模擬。在許多應用上，解特徵值分解扮演了重要的角色，這些矩陣通常都是稀疏的大矩陣，但通常實際上我們只需要一小部分的特徵對，現在有 FEAST 或 CIRR 能夠幫忙解決這類的問題，然而當區域中有許多特徵對時，會變得非常慢，所以必須將問題切成許多小問題。決定分區雖然困難但非常重要，要讓每個小問題都能被輕鬆解決，另外當有些特徵對收料的比較早時，原

doi:10.6342/NTU201803532

先的方法還是會花時間繼續迭代他們。我們介紹了兩種分區方式：藉由預測特徵對的數量來分解、藉由先備知識來分解。我們提升解決區域中有過多的特徵對問題，且藉由恰當的分區，分治法會比較原先的還快，也利用凍結技巧去讓程式不要花費時間在已經收斂的特徵對上。之後想設計一個方法能夠讓各個分區解決的時間差不多。

關鍵字： 多維陣列、多維奇異值分解、特徵值問題、凍結、閉曲線積分、加速、巨量資料

doi:10.6342/NTU201803532

Abstract

We implement, accelerate, and improve “Higher-Order Singular Value Decomposition” and “Contour Integral based Eigen Decomposition”.

In “Higher-Order Singular Value Decomposition”, we implemented two methods with different strategies and improved the ability to solve the large tensor problem. With the explosion of big data, finding ways of compressing and analyzing large data sets with the multi-way relationship - i.e., tensors - quickly and efficiently have become critical in High-Performance Computing. We implement two existed methods which are Higher-Order Singular Value Decomposition and Sequential Truncated Higher-Order Singular Value Decomposition to achieve Tucker Decomposition. Implementing them with GPU is very difficult because we usually can not store the whole tensor into GPU memory. We use QR method and Gram method to reduce the problem size to make its size allowed by GPU memory. We also implemented QR method and Gram by part-by-part. It can help us to solve the large data problem and use computing to cover data transferring. Finally, We achieve 163.21x speedup over a CUDA library-based solution. In the future, we want to apply it to the real application.

In “Contour Integral based Eigen Decomposition”, we proposed a divide-and-conquer flow to solving the certain eigenpairs in the specific region containing many eigenpairs with eigensolver based on contour integral with the

doi:10.6342/NTU201803532

locking technique, and use it to solve the generalized eigenvalue problem from the organic material simulation. Solving eigenvalue problems is an essential part of many applications. Those matrices are often large and sparse, but the eigenpairs only are required in the region of interest. Several solvers can solve the eigenpairs in the selected region such as FEAST and CIRR. When there are many eigenpairs in the selected region, the performance is slow, so the partition of the region is needed. Deciding the partition is very difficult but critical such that solving each sub-region should be efficient. When some eigenvector is converged early, the solver still spends time on them. We introduce the two partition method, uniform dividing by the estimated eigenvalue number and dividing by domain acknowledgment. We increase the eigensolver ability to solve the region containing many eigenpairs and get better performance with the proper partition. We also use the locking technique to avoid spending the time on converged eigenpairs. In the future, we would like to design an automatic flow to generate the partition whose sub-region spends almost the same executing time.

Keywords: Tensor, Higher-Order Singular Value Decomposition, Generalized Eigenvalue Problem, Locking, Contour Integral, Acceralating, Big Data

doi:10.6342/NTU201803532

Contents

doi:10.6342/NTU201803532

doi:10.6342/NTU201803532

doi:10.6342/NTU201803532

doi:10.6342/NTU201803532

List of Figures

doi:10.6342/NTU201803532

doi:10.6342/NTU201803532

List of Tables

doi:10.6342/NTU201803532

doi:10.6342/NTU201803532

Part I

Higher-Order Singular Value Decomposition

doi:10.6342/NTU201803532

doi:10.6342/NTU201803532

Chapter 1

Introduction

Singular Value Decomposition displays a critical role nowadays. We can use it to extract the critical feature and to compress the data. Singular Value Decomposition can find the best rank-n approximation of the matrix. However, we do not just want to analysis the bi-relationship of the data. Thus, we need to explore some ways to find the multi-way relationship.

With the explosion of big data, finding ways of compressing and analyzing large data sets with multi-way relationship - i.e. tensors - quickly and efficiently have become critical in HPC. For example, we want to compress the image with RGB channels. We can use Singular Value Decomposition on each channel before. The image structure is similar in each channel, but applying SVD on each channel is to use same feature to explain it. It leads to the performance is not good. If we use tensor decomposition to compress the image, it also take the relationship of color channel. It can compress the image more efficiently than SVD.

We will introduce Tucker decomposition, which is a kind of tensor decomposition. They are Higher-Order Singular Value Decomposition (HOSVD) and Sequential-Truncated Higher-Order Singular Value Decomposition (STHOSVD). They are easier to implement and do not need much additional knowledge about tensor.

While Higher-Order Singular Value Decomposition (HOSVD) and Sequential-Truncated Higher-Order Singular Value Decomposition (STHOSVD) provide us with the means to attain both extremely high compression ratio and low error rate though low-rank approx-

doi:10.6342/NTU201803532

imation, optimizing them on accelerators with limited memory is difficult.
We proposed some methods to overcome the two problems of GPU, slow performance in SVD and the memory problem. We use QR methods to make the program not to store whole data in the memory such that GPU can handle much larger tensor than directly implementation. We try different QR method to achieve the better performance in GPU such that the program in GPU have the competitive performance with MKL implementation. We share our experience and findings on optimizing these algorithms on a node with multiple GPUs, and demonstrate up to 163.21× speedup over a CUDA library-based solution.

The notations used in this part of this thesis are as follows. The regular letters or Latin letters such as a, α, denotes the scalar. The bold lower case letters, such as **v**, denotes the vectors. The bold uppercase letters, such as $\boldsymbol{A}$, denotes the matrix. The bold special letters, such as $\mathcal{T}$, denotes the tensor, We show the detail in the Table 1.1

a, α	scaler
$\mathbf{v}, \mathbf{w}$	vector
$\boldsymbol{A}, \boldsymbol{U}$	matrix
$\mathcal{T}, \mathcal{G}$	tensor
$\mathbf{v}_i$	the i-th element of vector
$\boldsymbol{A}_{ij}$ or $\boldsymbol{A}_{i,j}$	the (i, j) elemtent of matrix
$\boldsymbol{A}_i$	the i-th column vector of matrix
$\mathcal{T}_{i_1 i_2 \cdots i_N}$	the $(i_1, i_2, \cdots, i_N)$ element of tensor
$\mathcal{T}_{(i)}$	the mode-i of tensor
$\boldsymbol{A}^{(i)}$	the i-th matrix $\boldsymbol{A}$

Table 1.1: Notation in HOSVD part of this thesis

doi:10.6342/NTU201803532

Chapter 2

Preliminary

Tensor is a multi-dimension array. An N-way or Nth-order tensor is an element of N vector space. For example, a vector is a first-order/1-way tensor, a matrix is a second-order/2-way tensor, and the image with RGB color channels is a third-order/3-way tensor. A third-order tensor whose dimension is $I \times J \times K$ and elements are real number can be express $\mathcal{T} \in \mathbb{R}^{I \times J \times K}$. $\mathcal{T}(i, j, k)$ or $\mathcal{T}_{ijk}$ is the element (i, j, k) of $\mathcal{T}$. We show some definition and formula of tensors, and [21] shows more detail.

2.1 Tensor Formula

Definition 2.1.1. The norm of a tensor $\mathcal{T} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ is

$$\|\mathcal{T}\| = \sqrt{\sum_{i_1=1}^{I_1} \sum_{i_2=1}^{I_2} \cdots \sum_{i_N=1}^{I_N} \mathcal{T}^2_{i_1 i_2 \ldots i_N}}$$

This is analogous the matrix Frobenius norm.

Definition 2.1.2. Matricization: transforming a tensor into a matrix.

Matricization is known as unfolding or flattening. Use $\mathcal{T}_{(n)}$ to express the mode-n matricization of a tensor $\mathcal{T} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ and the leading dimension of $\mathcal{T}_n$ is I_n. We map the

doi:10.6342/NTU201803532

tensor element $(i_1, i_2, ..., i_N)$ to matrix element (i_n, j) where

$$j = 1 + \sum_{k=1, k\neq n}^{N} (i_k - 1)J_k, \text{ with } J_k = \prod_{m=1, m\neq n}^{k-1} I_m$$

The dimension of resulting matrix is $I_n \times \prod_{i=1, i\neq n}^{N} I_i$

Definition 2.1.3. Tensor multiplication: the n-mode product.

Tensors can be multiplied together, but it is more complex than the matrix multiplication. The more detail of tensor multiplication is shown in [1]. We focus the tensor n-mode product, that is, multiplying a tensor by a matrix in mode n.

The n-mode product of a tensor $\mathcal{T} \in \mathbb{R}^{I_1 \times I_2 \times ... \times I_N}$ and a matrix $\mathcal{A} \in \mathbb{R}^{J \times I_n}$ is a tensor whose dimension is $I_1 \times ... \times I_{n-1} \times J \times I_{n+1} \times ... \times I_n$. we use operation $\times_n$ to express n-mode product.

$$(\mathcal{T} \times_n \boldsymbol{U})_{i_1 \cdots i_{n-1} j i_{n+1} \cdots i_N} = \sum_{i_n=1}^{I_n} \mathcal{T}_{i_1 i_2 \cdots i_N} \boldsymbol{U}_{j i_n}$$

The formula can be expressed in terms of flatten tensor.

$$\mathcal{S} = \mathcal{T} \times_n \boldsymbol{A} \iff \mathcal{S}_{(n)} = \boldsymbol{A}\mathcal{T}_{(n)}$$

Definition 2.1.4. Vector outer product $\circ$:

$$\mathcal{T} = \mathbf{v}^{(1)} \circ \mathbf{v}^{(2)} \circ \cdots \circ \mathbf{v}^{(N)}$$

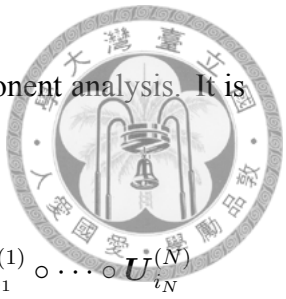
For element,

$$\mathcal{T}_{i_1 i_2 ... i_N} = \mathbf{v}^{(1)}_{i_1} \mathbf{v}^{(2)}_{i_2} \cdots \mathbf{v}^{(N)}_{i_N} \; \forall 1 \leq i_n \leq I_N$$

We also call the tensor $\mathcal{T}$ as a rank-one tensor.

doi:10.6342/NTU201803532

2.2 Tucker Decomposition



The Tucker Decomposition is a form of higher order principal component analysis. It is composed of a core n-way tensor and n matrices.

$$\mathcal{T} \approx \mathcal{G} \times_1 \boldsymbol{U}^{(1)} \times_2 \boldsymbol{U}^{(2)} \times_3 \cdots \times_N \boldsymbol{U}^{(N)} = \sum_{i_1}^{I_1} \cdots \sum_{i_N=1}^{I_N} \mathcal{G}_{i_1 \ldots i_N} U_{i_1}^{(1)} \circ \cdots \circ U_{i_N}^{(N)}$$

The tucker decomposition is also denoted by $[\![\mathcal{G}; \boldsymbol{U}^{(1)}, \cdots, \boldsymbol{U}^{(N)}]\!]$, and we call $\mathcal{G}$ as a core tensor and $\boldsymbol{U}^{(n)}$ as factor matrices.

The Tucker decomposition goes by many names mentioned in [21], and they summarized in Table 2.1

Name	Proposed by
Three-mode factor analysis (3MFA/Tucker3)	Tucker, 1966[35]
Three-mode principal component analysis (3MPCA)	Kroonenberg and De Leeuw, 1980[22]
N-mode principal components analysis	Kapteyn et al., 1986[17]
Higher-order SVD (HOSVD)	De Lathauwer et al., 2000 [2]
N-mode SVD	Vasilescu and Terzopoulos, 2002[37]

Table 2.1: Names for the Tucker decomposition from [21]

Remark. Tucker decomposition are not unique.

Let $[\![\mathcal{G}; \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}]\!]$ is a Tucker decomposition, and $\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{W}$ are not singular matrices. Then,

$$[\![\mathcal{G}; \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}]\!] = [\![\mathcal{G} \times_1 \boldsymbol{U} \times_2 \boldsymbol{V} \times_3 \boldsymbol{W}; \boldsymbol{A}\boldsymbol{U}^{-1}, \boldsymbol{B}\boldsymbol{V}^{-1}, \boldsymbol{C}\boldsymbol{W}^{-1}]\!]$$

doi:10.6342/NTU201803532

doi:10.6342/NTU201803532

Chapter 3

Methods and Results

If we wish to find a low-rank approximation to compress or extract significant features from a large data set with the multi-way relationship, Higher-Order Singular Value decomposition allows us to represent the original data with a *core* tensor, $\mathcal{G}$, and a set of factor matrices along each mode $\{\boldsymbol{A}^{(i)}\}_{i=1}^{N}$.

For example, if we want to store a tensor with dimension $1024 \times 1024 \times 1024$, we need $1024^3 \times 8$ bytes which are approximate 8 GBs of memory. If we apply HOSVD on this data with the rank of $256, 256, 256$, we can reduce its size to a mere ≈ 0.13 GB, while retaining most of its properties like Figure 3.1.

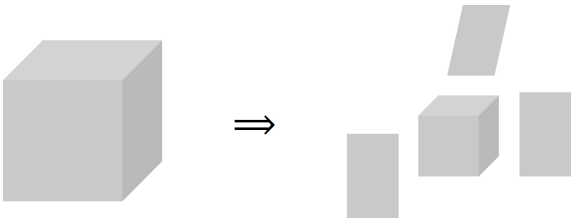


Figure 3.1: $\mathcal{X} \approx \mathcal{G} \times_{i=1}^{N} \boldsymbol{A}^{(i)}$

For a general tensor $\mathcal{X}$ with dimensions $d_1 \times d_2 \times \cdots \times d_n$, we can calculate a low-rank approximation using the following two algorithms: Algorithm 1: Higher Order Singular Value Decomposition (HOSVD) [2] and Algorithm 2: Sequential Truncated Higher Order Singular Value Decomposition (STHOSVD) [36]

doi:10.6342/NTU201803532

3.1 Algorithm

We show the algorithms in Algorithm 1 and Algorithm 2.

Algorithm 1 Higher Order Singular Value Decomposition (HOSVD)

Require: $\mathcal{X}, \{r_i\}$
Ensure: $\mathcal{G}, A^{(1)}, ..., A^{(N)}$
  for $i = 1$ to N **do**
    $\boldsymbol{A}^{(i)} \leftarrow r_i$ leading left singular vectors of $\mathcal{X}_{(i)}$ ▷ $\mathcal{X}_{(i)}$ is mode-i of $\mathcal{X}$
  end for
  $\mathcal{G} = \mathcal{X} \times_1 \boldsymbol{A}^{(1)^\top} \times_2 \boldsymbol{A}^{(2)^\top} \times_3 ... \times_N \boldsymbol{A}^{(n)^\top}$
  return $\mathcal{G}, \boldsymbol{A}^{(i)}$ ▷ tensor $\mathcal{G}$, matrices $A^{(1)}, ..., A^{(N)}$

Algorithm 2 Sequential Truncated Higher Order Singular Value Decomposition (STHOSVD)

Require: $\mathcal{X}, \{r_i\}$
Ensure: $\mathcal{G}, A^{(1)}, ..., A^{(N)}$
  $\mathcal{G} = \mathcal{X}$
  for $i = 1$ to N **do**
    $\boldsymbol{A}^{(i)} \leftarrow r_i$ leading left singular vectors of $\mathcal{G}_{(i)}$ ▷ $\mathcal{G}_{(i)}$ is mode-i of $\mathcal{G}$
    $\mathcal{G} \leftarrow \mathcal{G} \times_i \boldsymbol{A}^{(i)^\top}$
  end for
  return $\mathcal{G}, \boldsymbol{A}^{(i)}$ ▷ tensor $\mathcal{G}$, matrices $\boldsymbol{A}^{(1)}, ..., \boldsymbol{A}^{(N)}$

The difference between two algorithms is what tensor they use in each iteration. STHOSVD applies SVD on the truncated tensor, so the tensor dimension is reduced in each iteration. We get the benefit of solving a smaller tensor in each iteration than HOSVD's. In [36], they show the detail performance and error between two algorithms. We also show the different performance in Figure 3.8

First, we implement these two methods directly in CPU and GPU by MKL and CUDA as our baseline. The implementation by MKL is fast, but we have some troubles in implementation by CUDA.

Remark. Unlike Singular Value Decomposition, the Tucker Decomposition from Algorithm 1 and Algorithm 2 is not the best approximation. There are several algorithms to improve the accuracy such as higher-order orthogonal iteration (HOOI) based on HOSVD in [20].

doi:10.6342/NTU201803532

3.2 cuSOLVER SVD

When we implement the HOSVD/STHOSVD by CUDA, one of the critical bottlenecks is calculating the r_n left singular vectors of the matricized tensor. For dense tensors with large mode *lengths* or large *number* of modes, these are extremely large matrices that will not fit on the GPU memory. Unfortunately, SVD library provided by NVIDIA's cuSolver library requires that the entire data set is in memory before it can be factorized, limiting the range of tensors that can be decomposed, and a more scalable solution is required. we show the memory usage of data in Figure 3.2

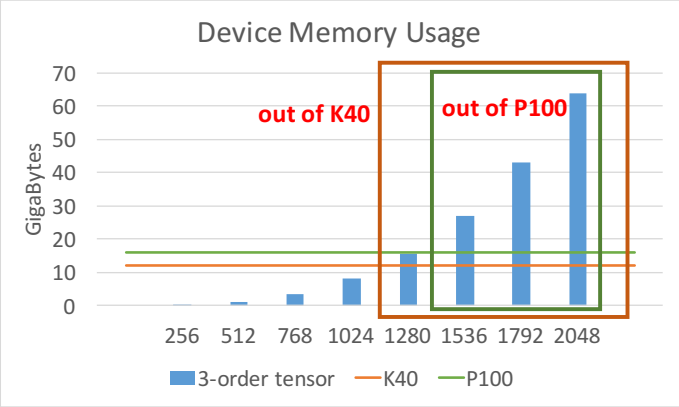


Figure 3.2: CUDA SVD is failed when tensor is large

Thus, we need to find some methods on GPU to solve the memory problem of GPU.

3.3 QR method

By Definition 2.1.2, we can know the resulting matrix of matricization is usually wide and short when the largest dimension is less than the sqrt of the total number of elements. Even if the largest dimension is larger than the sqrt of the total number of elements, it only leads to one mode matrix is tall-and-skinny and others still are wide-and-short. Thus, we only focus on how to solve this problem here.

One first solution is to use QR decomposition to reduce the matrix size before finding its singular vectors. We show the flow in Figure 3.3. The matrix is wide-and-short, so we need to transpose it. We do not need to implement transpose if we use row-major to store

doi:10.6342/NTU201803532

the matrix in CPU. We can do SVD on the upper triangular matrix of QR factorization. The matrix is much smaller than the original matrix so that GPU can solve its SVD. Another thing we need to consider, that is, how to solve the QR factorization. If we still use CUDA to solve it entirely in GPU, it still needs whole data in GPU memory, and we face the same problem again. Thus, we need some vector-wise or block-wise QR factorization methods.

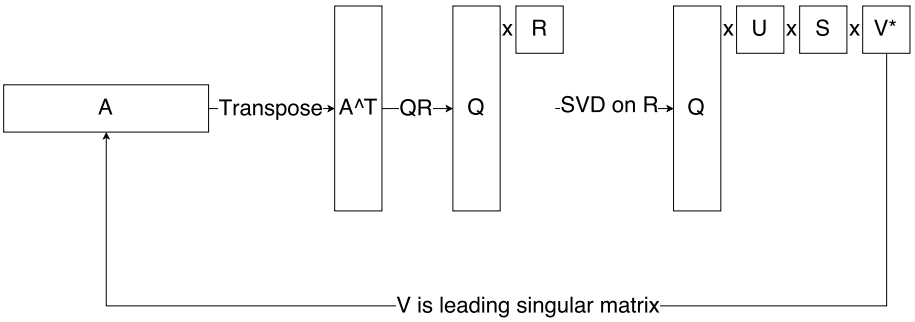
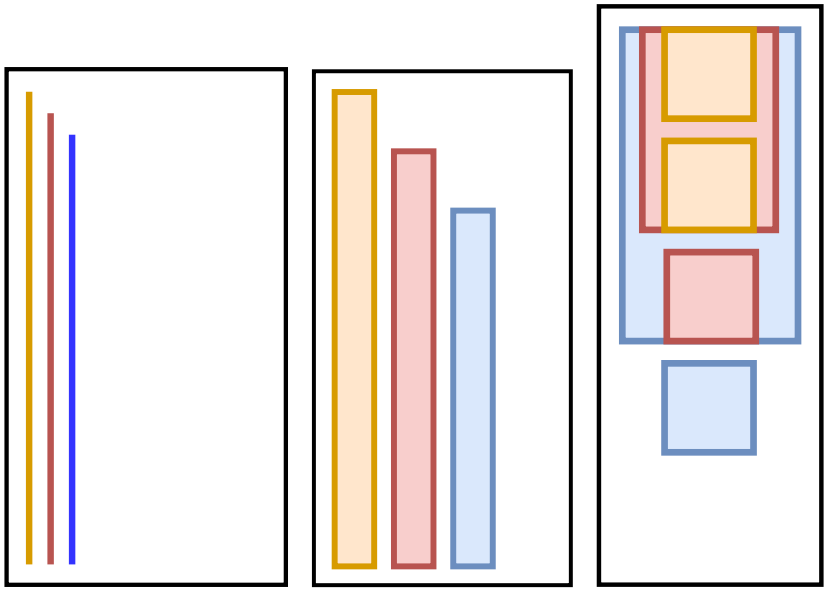


Figure 3.3: QR-based method



(a) HouseholderQR (b) BlockQR (c) TSQR

Figure 3.4: Three QR method overview

doi:10.6342/NTU201803532

3.4 Householder QR

We implemented the commonly used Householder QR method in [4] which allows us to stream the data vector-by-vector Figure 3.4(a), thereby requiring very little device memory. We show the algorithm in Algorithm 3

However, Householder QR has limitation. First, transferring the data vector-by-vector reduces bandwidth utilization. Second, the calculation is composed mostly of BLAS-2 operations, leading to reduced compute utilization

Householder QR is good at calculating HOSVD for large tensors but is slower than cuSolver SVD due to lower bandwidth and computing utilization in Figure 3.5.

Algorithm 3 Householder QR

Require: $\boldsymbol{A} \in \mathbb{R}^{m \times n}$
Ensure: $\boldsymbol{U}, \boldsymbol{R}$
  $\boldsymbol{R} = \boldsymbol{A}$
  $\boldsymbol{U} = zeros(m, n)$
  for $i = 1$ to n **do**
    $\mathbf{x} = \boldsymbol{R}(i:m, n)$
    $\mathbf{u} = \text{sign}(\mathbf{x}(1))\text{norm}(\mathbf{x})\mathbf{e}_1 + \mathbf{x}$
    $\mathbf{u} = \mathbf{u}/\text{norm}(\mathbf{u})$
    $\boldsymbol{R}(i:m, i:n) = R(i:m, i:n) - 2\mathbf{u}\mathbf{u}^\top \boldsymbol{R}(i:m, i:n)$
    $\boldsymbol{U}(i:m, i) = \mathbf{u}$
  end for
  return $\boldsymbol{U}, \boldsymbol{R}$ ▷ $\boldsymbol{U}$ are householder factors and $\boldsymbol{R}$ is the upper triangular matrix of QR

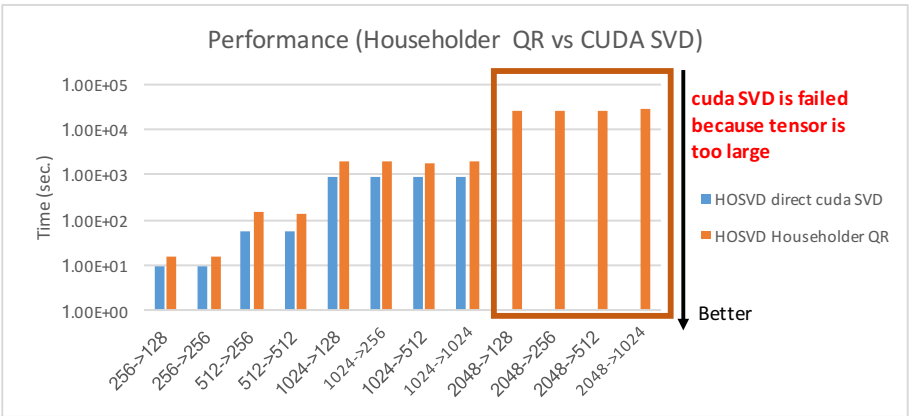


Figure 3.5: Householder QR method can solve large tensor

doi:10.6342/NTU201803532

3.5 Modified Block QR

The main idea of Block QR is to compute the block-wise householder factors. We can update the matrix by block-wise factors, and it is more efficient in GPU.

Theorem 3.5.1. *A series householder matrix can be a form of* $(\boldsymbol{I} + \boldsymbol{W}\boldsymbol{Y}^\top)$

Proof. $\boldsymbol{P}^{(i)} = \boldsymbol{I} + \beta_i \mathbf{u}^{(i)} \mathbf{u}^{(i)\top} = I + \mathbf{v}^{(i)} \mathbf{u}^{(i)\top}$ is the matrix from one householder factor. P is composed of a series $\boldsymbol{P}^{(i)}$

$$\boldsymbol{P} = \boldsymbol{P}^{(1)}\boldsymbol{P}^{(2)}...\boldsymbol{P}^{(r)} = (\boldsymbol{I} + \mathbf{v}^{(1)}\mathbf{u}^{(1)\top})(\boldsymbol{I} + \mathbf{v}^{(2)}\boldsymbol{U}^{(2)\top}) \cdots (I + \mathbf{v}^{(r)}\mathbf{u}^{(r)\top})$$

We show that $(\boldsymbol{I} + \boldsymbol{W}\boldsymbol{Y}^\top)(\boldsymbol{I} + \mathbf{u}\mathbf{v}^\top) = (\boldsymbol{I} + \hat{\boldsymbol{W}}\hat{\boldsymbol{Y}}^\top)$, for $\boldsymbol{Y}, \boldsymbol{W} \in \mathbb{R}^{n \times k}$ and $\mathbf{u}, \mathbf{v} \in \mathbb{R}^{n \times 1}$

$$\begin{aligned}
(\boldsymbol{I} + \boldsymbol{W}\boldsymbol{Y}^\top)(\boldsymbol{I} + \mathbf{v}\mathbf{u}^\top) &= \boldsymbol{I} + \boldsymbol{W}\boldsymbol{Y}^\top + (\boldsymbol{I} + \boldsymbol{W}\boldsymbol{Y}^\top)(\mathbf{v}\mathbf{u}^\top) \\
&= \boldsymbol{I} + [\boldsymbol{W},\ (\boldsymbol{I} + \boldsymbol{W}\boldsymbol{Y}^\top\mathbf{v})][\boldsymbol{Y},\ \mathbf{u}]^\top \\
&= \boldsymbol{I} + \hat{\boldsymbol{W}}\hat{\boldsymbol{Y}}^\top
\end{aligned}$$

where $\hat{\boldsymbol{W}} = [\boldsymbol{W},\ (\boldsymbol{I} + \boldsymbol{W}\boldsymbol{Y}^\top\mathbf{v})] \in \mathbb{R}^{n \times (k+1)}$ and $\hat{\boldsymbol{Y}} = [\boldsymbol{Y},\ \mathbf{u}] \in \mathbb{R}^{n \times (k+1)}$

We do it recursively on the series of $\boldsymbol{P}^{(i)}$, so we can get $\boldsymbol{P} = (\boldsymbol{I} + \boldsymbol{W}\boldsymbol{Y})$ for some $\boldsymbol{W}, \boldsymbol{Y}$ □

In algorithm 4 line 13, there are two Mat-vec operations. However, the Blas-2 operations are slow in GPU, so we modify a little part of these codes to make it more powerful.

In algorithm 5, there are only one Mat-Mat operation (line 10) out of for-loop and one Mat-Vec operation (line 14) in for-loop.

We replace r level-2 operation with one level 3 operation. It is more suitable in GPU than the original one.

By using extra device memory, modified Block QR (Figure 3.4(b)) combines several Householder factors into two matrices to increase the memory throughput. Modified

doi:10.6342/NTU201803532

Algorithm 4 Block QR

Require: $\boldsymbol{A}, r$
Ensure: $\boldsymbol{Q}, \boldsymbol{R}$

$\boldsymbol{Q} = \boldsymbol{I}$
for k=1:n/r, s=(k-1)r+1 **do**
  for j=1:r **do**
    $u = s + j - 1$
    $[\boldsymbol{V}, \beta] = house(\boldsymbol{A}(u : m, u))$
    $\boldsymbol{A}(u : m, u : s + r - 1) = (\boldsymbol{I} + \beta \mathbf{v}\mathbf{v}^\top)\boldsymbol{A}(u : m, u : s + r - 1)$
    $\boldsymbol{V}(:, j) = [zeros(j - 1, 1);\ \mathbf{v}], \mathbf{b}(j) = \beta$
  end for
  $\boldsymbol{Y} = \boldsymbol{V}(1 : end, 1)$
  $\boldsymbol{W} = \mathbf{b}(1)\boldsymbol{V}(1 : end, 1)$
  for j=2:r **do**
    $\mathbf{v} = \boldsymbol{V}(:, j)$
    $\mathbf{z} = (\boldsymbol{I} + \boldsymbol{W}\boldsymbol{Y}^\top)\mathbf{v}$
    $\boldsymbol{W} = [\boldsymbol{W},\ \mathbf{b}(j)\mathbf{z}], \boldsymbol{Y} = [\boldsymbol{Y},\ \mathbf{v}]$
  end for
  $\boldsymbol{A}(s : m, s + r : n) = (\boldsymbol{I} + \boldsymbol{Y}\boldsymbol{W}^\top)\boldsymbol{A}(s : m, s + r : n)$
  $\boldsymbol{Q}(1 : m, s : m) = \boldsymbol{Q}(1 : m, s : m)(\boldsymbol{I} + \boldsymbol{W}\boldsymbol{Y}^\top)$
end for

Algorithm 5 Modified Block QR

Require: $\boldsymbol{A}, \boldsymbol{R}$
Ensure: $\boldsymbol{Q}, \boldsymbol{R}$

$\boldsymbol{Q} = \boldsymbol{I}$
for k=1:n/r, s=(k-1)r+1 **do**
  for j=1:r **do**
    $u = s + j - 1$
    $[\mathbf{v}, \beta] = house(\boldsymbol{A}(u : m, u))$
    $\boldsymbol{A}(u : m, u : s + r - 1) = (\boldsymbol{I} + \beta \mathbf{v}\mathbf{v}^\top)\boldsymbol{A}(u : m, u : s + r - 1)$
    $\boldsymbol{V}(:, j) = [zeros(j - 1, 1);\ \mathbf{v}], \mathbf{b}(j) = \beta$
  end for
  $\boldsymbol{Y} = \boldsymbol{V}$
  $\boldsymbol{C} = \boldsymbol{V}^\top \boldsymbol{V}$
  $\boldsymbol{W} = \mathbf{b}(1)\boldsymbol{V}(1 : end, 1)$
  for j=2:r **do**
    $\mathbf{v} = \boldsymbol{V}(:, j)$
    $\mathbf{z} = \mathbf{v} + \boldsymbol{W} * \boldsymbol{C}(1 : j - 1, j)$
    $\boldsymbol{W} = [\boldsymbol{W},\ \mathbf{b}(j)\mathbf{z}], \boldsymbol{Y} = [\boldsymbol{Y},\ \mathbf{v}]$
  end for
  $\boldsymbol{A}(s : m, s + r : n) = (\boldsymbol{I} + \boldsymbol{Y}\boldsymbol{W}^\top)\boldsymbol{A}(s : m, s + r : n)$
  $\boldsymbol{Q}(1 : m, s : m) = \boldsymbol{Q}(1 : m, s : m)(\boldsymbol{I} + \boldsymbol{W}\boldsymbol{Y}^\top)$
end for

doi:10.6342/NTU201803532

Block QR has the more blas-3 operation than Householder QR. Not surprisingly, Modified Block QR is faster than Householder QR. Also, we only need some column blocks of the matrix in GPU memory, so we can also solve larger data. However, when the matrix is tall, the corresponding block used in modified Block QR is too skinny to utilize the compute units fully and reduces the overall performance (much like the Household QR case).

3.6 Tall Skinny QR

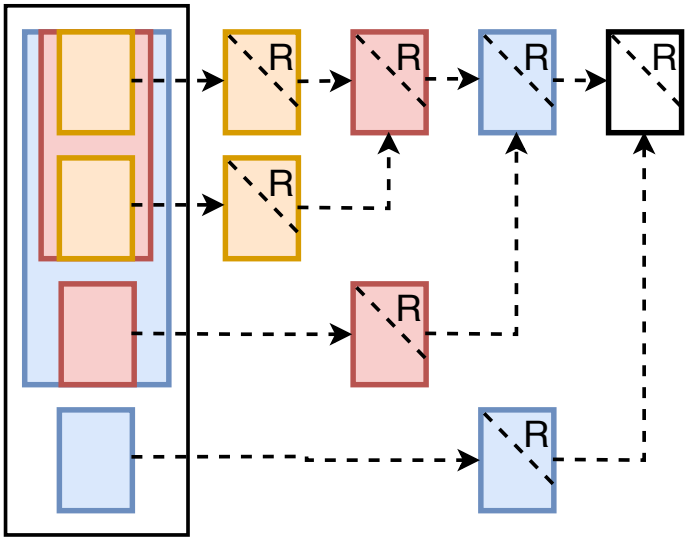


Figure 3.6: solve upper triangular matrix by TSQR

We can only compute the upper triangular matrix of QR factorization in Figure 3.3, so we show how to solve it by tall skinny QR in Figure 3.6. How to solve Q matrix by TSQR and more details are shown in [3].

TSQR is more suitable for this problem than Block QR because it splits the rows to avoid the above situation. It divides the matrix into several square blocks in a column. To solve the QR problem, we solve the QR factorization for two adjacent blocks independently and combine them. Inductively, it forms the whole process.

We explored using Tall Skinny QR - TSQR (Figure 3.4(c)) to overcome the throughput issues. Moreover, we do not need to solve Q of QR by TSQR, so we reduce many operations

doi:10.6342/NTU201803532

to achieve better performance. Modified Block QR and TSQR significantly improve the performance of the HOSVD algorithm. The performance of them are shown in Figure 3.7

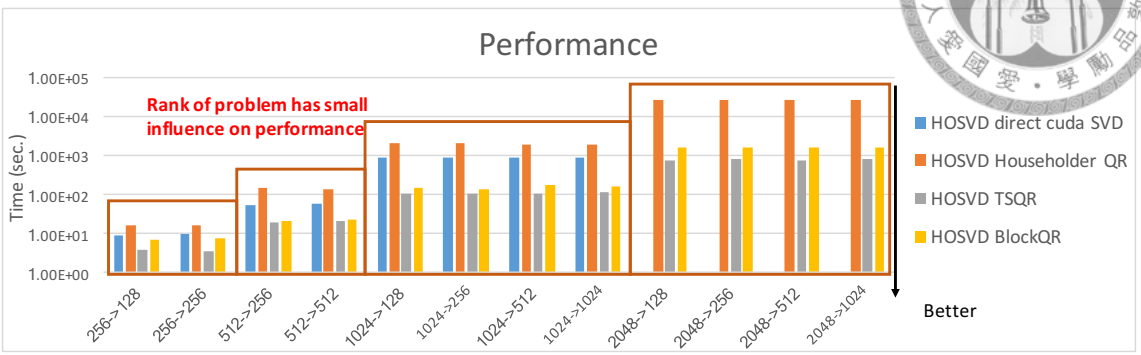


Figure 3.7: Performance of HOSVD with QR methods

Modified Block QR and TSQR improve performance over the Householder QR and cuSolver SVD methods. However, HOSVD's performance is independent of the rank. Even if we want just a small rank approximation of the tensor, we need to spend similar time computing it like computing full rank approximation.

Therefore, we introduce the STHOSVD method that can further improve the performance, and reduce the overall work when the rank is smaller by calculating the Tensor-Times Matrix (TTM) step within the inner loop.

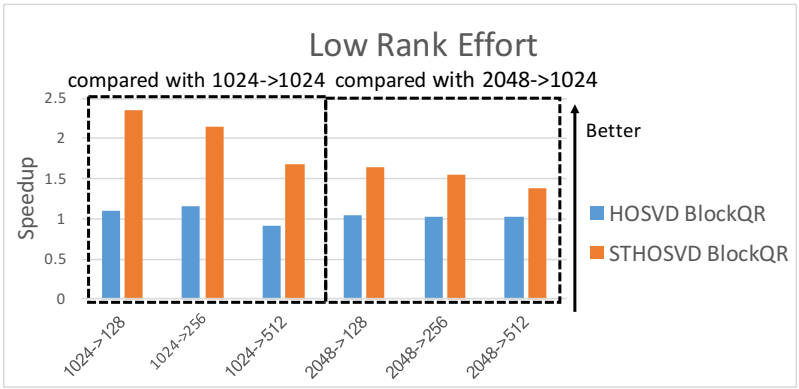
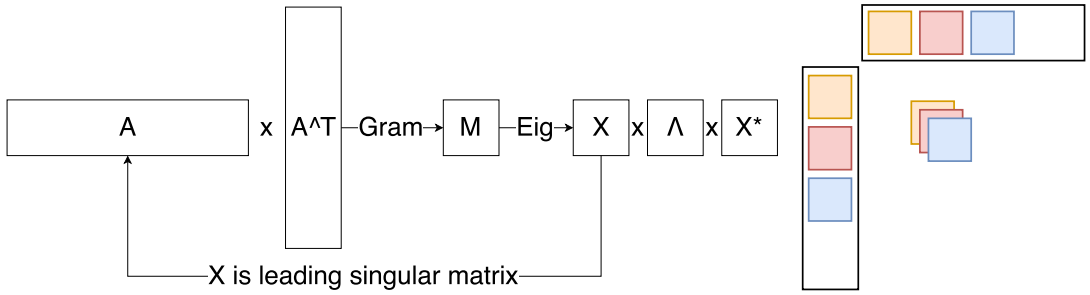


Figure 3.8: STHOSVD vs HOSVD with different rank

doi:10.6342/NTU201803532

3.7 Gram Method

Besides QR methods we also use Gram method followed by eigenvalue decomposition to solve the problem (Figure 3.9(a)). In this process, we multiply the matricized tensor by its transpose and regard the initial SVD problem as an eigenvalue problem of Gram matrix with smaller problem size (Figure 3.9(b)).



(a) Gram flow (b) Gram

Figure 3.9: Gram method

The block multiplications used in Gram method are independent, so we can interleave the matrix transfer with computation to increase efficiency, and then combine it with the STHOSVD method to increase performance. The Performance of STHOSVD is shown in Figure 3.10

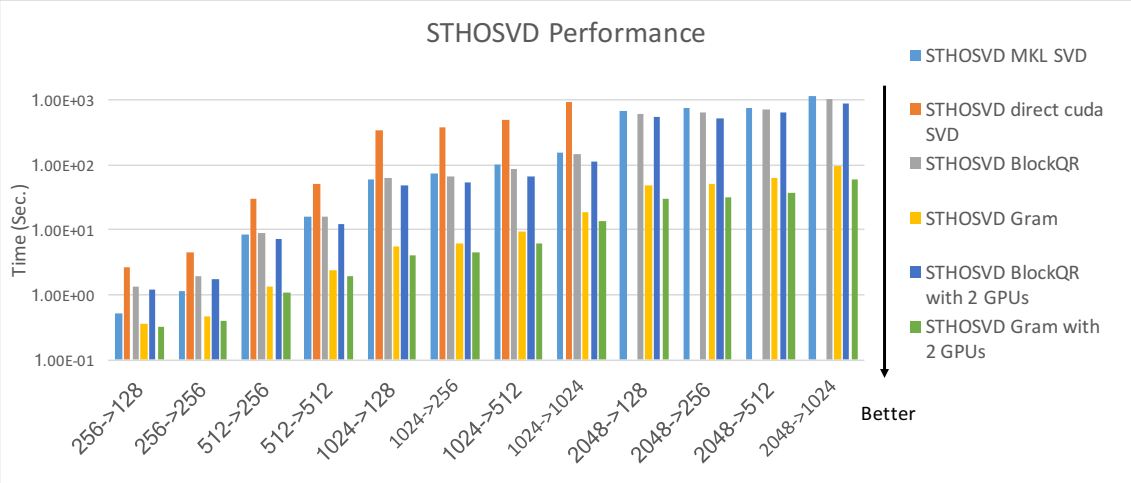


Figure 3.10: STHOSVD: Gram method is the fastest algorithm

doi:10.6342/NTU201803532

3.8 Error

We generate the dense tensor with random numbers whose condition number is small, so the accuracy of QR method and Gram method are competitive in Figure 3.11. The random generating tensor is not main structure inside itself, so the non-full rank approximation is lower accuracy.

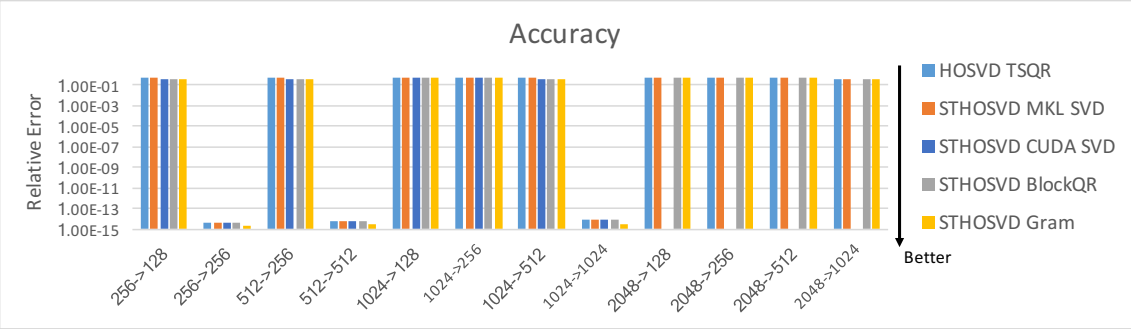


Figure 3.11: Gram and QR methods have similar accuracy

doi:10.6342/NTU201803532

doi:10.6342/NTU201803532

Chapter 4

Implementation

4.1 Transpose

For the QR methods, the transpose of the matrix we need to consider. However, when we store the matrix row-major in CPU, we can do the transpose implicitly. In the memory, the data is stored as sequential, that is, $\boldsymbol{A}$ is stored as $\boldsymbol{A}_{11}\boldsymbol{A}_{12}\boldsymbol{A}_{13}\cdots\boldsymbol{A}_{1M}\boldsymbol{A}_{21}\cdots\boldsymbol{A}_{NM}$ (row-major) in CPU memory. When we move the data to GPU sequential, the ordering of elements is still the same. The function provided by NVIDIA is for column-major so that the function will see the transpose of $\boldsymbol{A}$

In GPU view, such as Figure 4.1,

$$\begin{bmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{21} & \cdots & \boldsymbol{A}_{M1} \\ \boldsymbol{A}_{12} & \boldsymbol{A}_{22} & \cdots & \boldsymbol{A}_{M2} \\ \vdots & \ddots & \ddots & \vdots \\ \boldsymbol{A}_{1N} & \boldsymbol{A}_{2N} & \cdots & \boldsymbol{A}_{MN} \end{bmatrix} = \begin{bmatrix} \boldsymbol{A}_{11}^\top & \boldsymbol{A}_{12}^\top & \cdots & \boldsymbol{A}_{1M}^\top \\ \boldsymbol{A}_{21}^\top & \boldsymbol{A}_{22}^\top & \cdots & \boldsymbol{A}_{2M}^\top \\ \vdots & \ddots & \ddots & \vdots \\ \boldsymbol{A}_{N1}^\top & \boldsymbol{A}_{N2}^\top & \cdots & \boldsymbol{A}_{NM}^\top \end{bmatrix} = \boldsymbol{A}^\top$$

doi:10.6342/NTU201803532

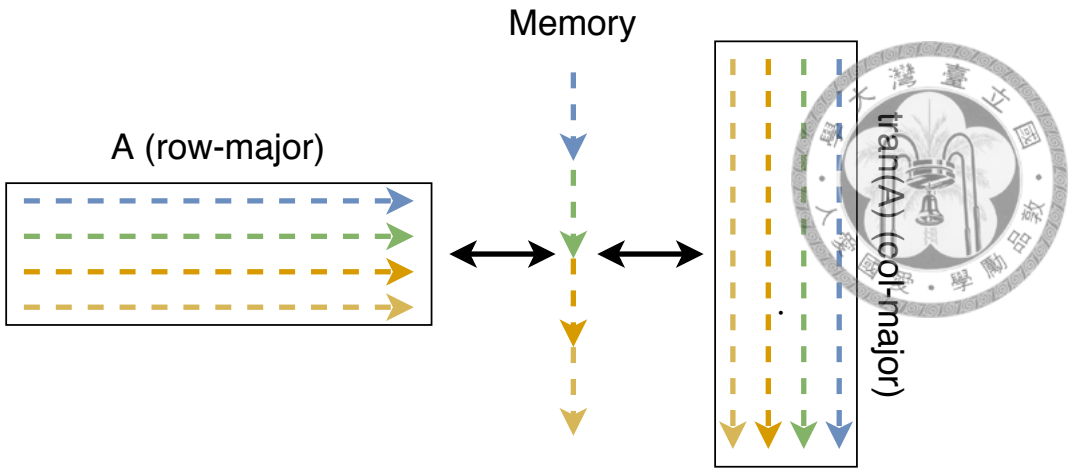


Figure 4.1: the view on different major

Similarly, we can get back the $\boldsymbol{V}^\top$ back to GPU without transpose.

Remark. In cuSolver 8, 9.0, 9.1, 9.2, the routine cusolverDn<t>gesvd returns $\boldsymbol{V}^\top$ not $\boldsymbol{V}$ for real number.

4.2 CUDA Stream

We use CUDA stream on GPU to schedule the works. The different streams are 'almost' independent, so the GPU can do several small tasks at the same time if they can. The stream thought is also between different GPU. If there is no constraint, the streams in different GPUs are really independent. In Section 4.3, we use two GPU to do the updating step at the same time. For BlockQR, Housholder QR Blas-2 operation, we also use several streams to do it.

Remark. The streams mentioned here are not the default stream.
Whether the function can be run simultaneously in several streams of the same GPU depends on the number of SM function using, or data movement, etc.

doi:10.6342/NTU201803532

4.3 Multiple GPUs

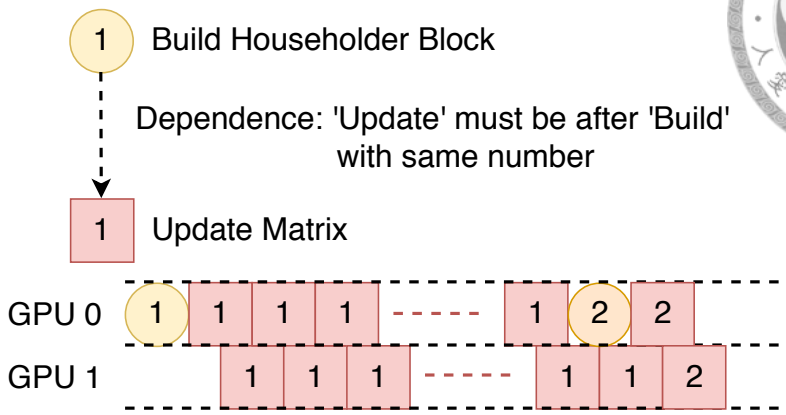


Figure 4.2: the idea of blockQR multigpu

Householder QR (Figure 3.4(a)) and Block QR (Figure 3.4(b)):

We implement Householder QR and Block QR block-wise. The information also updates block-wise. These methods update simultaneously on multiple GPUs. We make one GPU update fewer blocks than the other GPUs and solve QR factorization of the next block when other GPUs still update the remaining blocks. We show the idea in Figure 4.2. With this schedule, the next round updating step does not need to wait for the QR factorization.

- TSQR (Figure 3.4(c)):

  We split a matrix into several tall and skinny blocks by the number of GPUs. We use TSQR to solve each block, so it is an independent process. And then we combine the results together in a single matrix and apply TSQR method on this matrix.

- Gram method (Figure 3.9(b)):

  We split the matrix into several small blocks. For each block, we only need to calculate the product of its transpose and itself. The operations in each block are independent so that we can assign those works to multiple GPUs equally.

doi:10.6342/NTU201803532

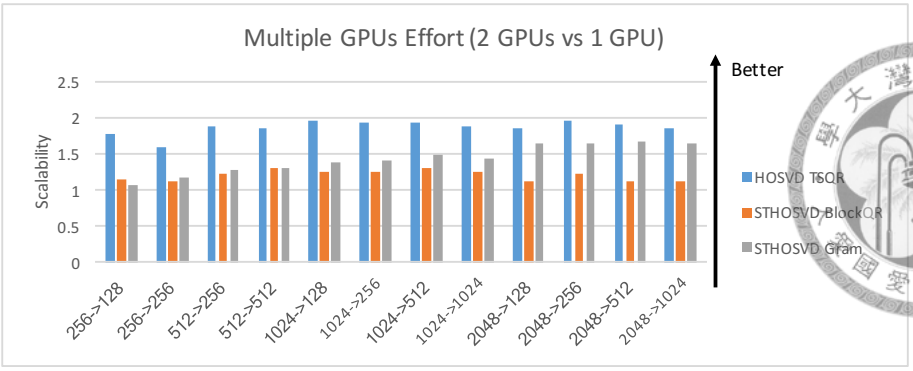


Figure 4.3: Scalability of multiple GPUs

doi:10.6342/NTU201803532

Chapter 5

Conclusion

We study and optimize four methods - Householder QR, Modified Block QR, TSQR, and Gram method - to solve the SVD step in HOSVD and STHOSVD. QR methods implemented by the vector-wise or the block-wise algorithm can solve the GPU memory problems. Among these QR methods, TSQR is the fastest one. TSQR can ignore the computing Q step, and it uses row block which is more suitable for our problems.

We also use another algorithm, Gram method. Among them, Gram method is the fastest algorithm and the Simplest to implement, and provides comparable accuracy in less time when the condition number of the tensor is small.

Although QR methods are slower than Gram method, in the case that the condition number of the original tensor is large, it may provide higher accuracy.

For the overall performance, we show it in Figure 5.1 or figure in https://goo.gl/QsovD1.

doi:10.6342/NTU201803532

doi:10.6342/NTU201803532

Figure 5.1: All Performance

Part II

Contour Integral based Eigenvalue Decomposition

doi:10.6342/NTU201803532

doi:10.6342/NTU201803532

Chapter 6

Introduction

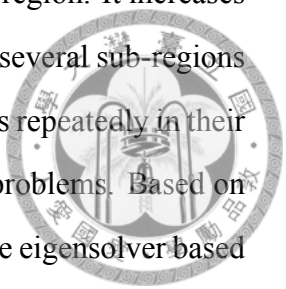
Solving eigenvalue problems is an important part of many applications. Those matrices are often large and sparse but the eigenpairs only are required in the region of interest. Several solvers can solve the eigenpairs in selected regions such as FEAST and CIRR. FEAST is a density-matrix-based algorithm proposed by Polizzi [27] [26], [19]. CIRR is the Raleigh-Ritz-type approach of the contour integral eigensolver proposed by Ikegami and Sakurai [29], [12], [13], [16]. These eigensolvers are extended to solve non-Hermitian eigenvalue problems. The numerical analysis are shown in [16] and [26]

These eigensolvers require inputs of columns of the initial subspace. If the initial subspace is too small, it can not solve the eigenpairs in the region of interest. There are some algorithms for estimating the number of eigenpairs in certain region in [5], [6], and [24]. We use the estimated number of the eigenpairs as the number of columns of initial subspace with the scale (we use 1.5 as the scale). That is, $\#\{\text{cols of subspace}\} = \lceil \text{scale}(1.5) \times \#\{\text{estimation}\} \rceil$. The estimated number is only used to build the initial subspace. Moreover, FEAST also estimates the number of the eigenpairs in second loop [26], and it uses its estimated result in stopping criterion.

Partition of the region is a crucial part of solving generalized eigenvalue problems by the eigensolver based on contour integral. The eigensolver based on contour integral is a powerful tool to solve the whole eigenpairs in a given region for generalized eigen-

doi:10.6342/NTU201803532

value problem. In some applications, there are many eigenpairs in the region. It increases the difficulty of the problem. We can separate the original region into several sub-regions whose union is the original one. However, it will solve some eigenpairs repeatedly in their intersections. Thus, the deflating technique is needed to avoid these problems. Based on deflating technique, we introduced a divide-and-conquer method on the eigensolver based on contour integral.



The deflating technique can remove those repeated eigenpairs we have solved. The matrices are sparse, so we can explicitly form the deflated matrices. It will turn the matrices into dense matrices. We can use Woodbury Identity Theorem to solve the linear systems without building matrices explicitly, but it will increase the number of linear systems [7]. Locking technique [31], [28] is known as an implicit deflating technique. It may lead the convergence problem [31], but it can decrease the number of linear systems by removing solved vector out of the searching base. In [39], they show how to apply locking technique in FEAST for solving Hermitian standard eigenvalue problems.

This paper discusses the dividing-and-conquer method with locking technique based on FEAST for solving Hermitian generalized eigenvalue problems and shows how to apply the locking technique to generalized eigenvalue problems with some condition and discuss the performance between different kinds of partition, such as auto-partition and pre-knowledge partition. We choose the PENTF cases from [9] as our testing data. We show the benefits of locking, and the fact that partition is helpful when solving much larger problems and sometimes accelerate the solving process.

The notations used in this part of this thesis are as follows. The regular letters or Latin letters such as a, α, denotes the scalar. The bold lower case letters, such as $\mathbf{v}$, denotes the vectors. The bold uppercase letters, such as $\boldsymbol{A}$, denotes the matrix. We show the detail in the Table 6.1

doi:10.6342/NTU201803532

a, α	scaler
$\mathbf{v}$	vector
$\boldsymbol{A}$	matrix
$\mathbf{v}_i$	the i-th element of vector
$\boldsymbol{A}_{ij}$ or $\boldsymbol{A}_{i,j}$	the (i, j) elemtent of matrix
$\boldsymbol{A}_i$	the i-th column vector of matrix
$\boldsymbol{A}^{(i)}$	the i-th matrix $\boldsymbol{A}$
$\boldsymbol{A}_{[k]}$	specified the column of matrix.

Table 6.1: Notation in CI part of this thesis

doi:10.6342/NTU201803532

doi:10.6342/NTU201803532

Chapter 7

Preliminary

7.1 Eigensolver based on Contour Integral

We consider a circle in the complex plane, and we call the boundary Γ and the interior Ω such as Figure 7.1. Eigensolver based on Contour Integral computes some equations on the quadrature points to get the basis of the eigenpairs in the Ω. After building the basis, we project the original matrix pairs on this basis to form a smaller generalized eigenvalue problem than the original one. Then we solve its eigenpairs, and we can reconstruct the eigenpairs in the interior Ω of the original matrix pair $\boldsymbol{A}$, $\boldsymbol{B}$. We will introduce some algorithm in later.

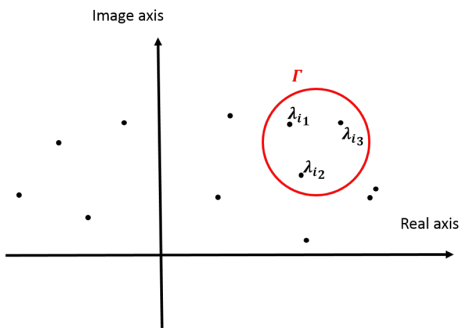


Figure 7.1: Circle on complex plane

doi:10.6342/NTU201803532

7.2 Quadrature rules

We need to calculate the equations of the contour integral numerically. Thus, we introduce some quadrature rules show how to build up **x**, **w** of P quadratic points for $k = 1 \cdots P$. It is gotten from [39]

- midpoint rule:

$$\begin{cases} \mathbf{x}_k & = \dfrac{2k-1}{2P} \\ \mathbf{w}_k & = \dfrac{1}{P} \end{cases}$$

- Gauss-Chebyshev rule for the first kind:

$$\begin{cases} \mathbf{x}_k & = \dfrac{1}{2}(1 + \cos(\dfrac{(2k-1)\pi}{2P})) \\ \mathbf{w}_k & = \dfrac{2\pi}{P} \sin(\dfrac{(2k-1)\pi}{2P}) \end{cases}$$

- Gauss-Chebyshev rule for the second kind:

$$\begin{cases} \mathbf{x}_k & = \dfrac{1}{2}(1 + \cos(\dfrac{k\pi}{P+1})) \\ \mathbf{w}_k & = \dfrac{2\pi}{P} \sin(\dfrac{k\pi}{P+1}) \end{cases}$$

- Gauss-Legendre rule:

$$\begin{cases} \mathbf{x}_k & = \dfrac{t_k + 1}{2} \\ \mathbf{w}_k & = \dfrac{1}{(1-t_k^2)(L'_P(t_k))^2} \end{cases}$$

  where t_k is the k-th root of the pth Legendre polynomial $L_P(x)$

Among these quadratic rule, the Gauss-Legendre rule (Section 7.2) is popular. MKL FEAST and our implementation use this rule.

Remark. For using Section 7.2, we need to compute the point **z** on the circle boundary Γ

doi:10.6342/NTU201803532

$$\theta_k = -\frac{\pi}{2}(x_k - 1)$$

$$\mathbf{z}_k = c + re^{i\theta_i}$$

$$\mathbf{w}_k \leftarrow -\frac{w_k}{2}$$

By the way, we implement the complex Hermitian version on the eigensolver. The quadratic point of the lower part is conjugate with those point of the upper part, and weight is not changed.

doi:10.6342/NTU201803532

doi:10.6342/NTU201803532

Chapter 8

Theorem

8.1 Deflation

The original generalized eigenvalue problem is given by

$$\boldsymbol{A}\mathbf{x} = \lambda \boldsymbol{B}\mathbf{x}$$

If all eigenvectors are B-orthonormal, we can use the following deflation technique.

Let a set $\{y\}_j$ is B-orthonormal, i.e. ,

$$\mathbf{y}_i^* \boldsymbol{B} \mathbf{y}_j = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases}$$

Collect those eigenvectors we have solved in S.

$\mathbf{y}$ is an eigenvector. $\boldsymbol{A}\mathbf{y} = \lambda \boldsymbol{B}\mathbf{y}$

Consider $\tilde{\boldsymbol{A}} = \boldsymbol{A} + \sigma \boldsymbol{B}(\sum_{\mathbf{v}\in S} \mathbf{v}\mathbf{v}^*)\boldsymbol{B}^*$.

If $\mathbf{y} \in S$, then $\tilde{\boldsymbol{A}}\mathbf{y} = \boldsymbol{A}\mathbf{y} + \sigma \boldsymbol{B}(\sum_{\mathbf{y}\in S} \mathbf{v}\mathbf{v}^*)\boldsymbol{B}^*\mathbf{y} = \lambda \boldsymbol{B}\mathbf{y} + \sigma \boldsymbol{B}\mathbf{y} = (\lambda + \sigma)\boldsymbol{B}\mathbf{y}$.

If $\boldsymbol{Y} \notin S$, then $\tilde{\boldsymbol{A}}\mathbf{y} = \boldsymbol{A}\mathbf{y} + \sigma \boldsymbol{B}(\sum_{\mathbf{v}\in S} \mathbf{v}\mathbf{v}^*)\boldsymbol{B}^*\mathbf{y} = \lambda \boldsymbol{B}\mathbf{y} + 0 = \lambda \boldsymbol{B}\mathbf{y}$.

By applying such deflation technique, the eigenvectors belonging to S can be removed from S

doi:10.6342/NTU201803532

8.2 Theorem

The method can be applied to these problems which satisfy the following conditions.

Condition 8.2.1.

$$\forall\{\lambda, \mathbf{x}\} \textit{ eigenpairs, } \boldsymbol{A}\mathbf{x} = \lambda\boldsymbol{B}\mathbf{x} \iff \boldsymbol{A}^*\mathbf{x} = \lambda^*\boldsymbol{B}^*\mathbf{x} \tag{8.1}$$

$$\forall\mathbf{x}, \mathbf{y}, \langle\mathbf{x}, \boldsymbol{B}\mathbf{y}\rangle = \langle\boldsymbol{B}\mathbf{x}, \mathbf{y}\rangle \tag{8.2}$$

Theorem 8.2.2. *If $\boldsymbol{A}\mathbf{x} = \lambda\boldsymbol{B}\mathbf{x}$ and $\boldsymbol{A}^*\mathbf{x} = \lambda^*\boldsymbol{B}^*\mathbf{x}$, then the eigenvectors with distinct eigenvalues are B-orthogonal.*

Proof. Write $\boldsymbol{A}\mathbf{x} = \lambda\boldsymbol{B}\mathbf{x}$ and $\boldsymbol{A}\mathbf{y} = \mu\boldsymbol{B}\mathbf{y}$, which $\lambda \neq \mu$

$$\langle\mathbf{y}, \boldsymbol{A}\mathbf{x}\rangle = \mathbf{y}^*\lambda\boldsymbol{B}\mathbf{x} = \lambda\mathbf{y}^*\boldsymbol{B}\mathbf{x}$$

$$\langle\mathbf{y}, \boldsymbol{A}\mathbf{x}\rangle = \langle\boldsymbol{A}^*\mathbf{y}, \mathbf{x}\rangle$$

$$\langle\boldsymbol{A}^*\mathbf{y}, \mathbf{x}\rangle = \langle\mu^*\boldsymbol{B}^*\mathbf{y}, \mathbf{x}\rangle = \mathbf{y}^*\boldsymbol{B}\mu\mathbf{x} = \mu\mathbf{y}^*\boldsymbol{B}\mathbf{x}$$

$$0 = \langle\mathbf{y}, \boldsymbol{A}\mathbf{x}\rangle - \langle\boldsymbol{A}^*\mathbf{y}, \mathbf{x}\rangle = (\lambda - \mu)\mathbf{y}^*\boldsymbol{B}\mathbf{x}$$

$\lambda \neq \mu$, so $\mathbf{y}^*\boldsymbol{B}\mathbf{x} = 0$. i.e. they are B-orthogonal.

Note that, in this argument, condition(2) are not required. □

Theorem 8.2.3. *$\boldsymbol{A}\mathbf{x} = \lambda\boldsymbol{B}\mathbf{x}$, $\boldsymbol{A}^*\mathbf{x} = \lambda^*\boldsymbol{B}^*\mathbf{x}$, and $\langle\mathbf{x}, \boldsymbol{B}\mathbf{y}\rangle = \langle\boldsymbol{B}\mathbf{x}, \mathbf{y}\rangle$. The eigenvectors with identical eigenvalues can be reformulated as B-orthogonal eigenvectors with the same eigenvalues.*

Proof. By collecting independent eigenvectors with identical eigenvalues, we can reconstruct an orthonormal eigenvector with the same eigenvalue via linear combination.

$$\mathbf{y} = \sum_{i=1}^{r} a_i\mathbf{x}^{(i)}$$

$$\boldsymbol{A}\mathbf{y} = \boldsymbol{A}\sum_{i=1}^{r} a_i\mathbf{x}^{(i)} = \sum_{i=1}^{r} a_i\boldsymbol{A}\mathbf{x}^{(i)} = \sum_{i=1}^{r} a_i\lambda\boldsymbol{B}\mathbf{x}^{(i)} = \lambda\boldsymbol{B}\sum_{i=1}^{r} a_i\mathbf{x}^{(i)} = \lambda\boldsymbol{B}\mathbf{y}$$

doi:10.6342/NTU201803532

Thus, the eigenvalue of a linear combination of eigenvectors remains the same

One can re-scale them to make it B-orthonormal. For example, **x** and **y** are eigenvectors with same eigenvalue. Assume $\langle \mathbf{x}, \boldsymbol{B}\mathbf{y} \rangle = k$ is not B-orthonormal. we denote $\hat{\mathbf{y}} = \mathbf{y} - k\mathbf{x}$. Then,

$\langle \mathbf{x}, \boldsymbol{B}\hat{\mathbf{y}} \rangle = \langle \mathbf{x}, \boldsymbol{B}(\mathbf{y} - k\mathbf{x}) \rangle = \langle \mathbf{x}, \boldsymbol{B}\mathbf{y} \rangle - \langle \mathbf{x}, k\mathbf{x} \rangle = k - k = 0$

$\langle \hat{\mathbf{y}}, \boldsymbol{B}\mathbf{x} \rangle = \langle (\mathbf{y} - k\mathbf{x}), \boldsymbol{B}\mathbf{x} \rangle = \langle \mathbf{y}, \boldsymbol{B}\mathbf{x} \rangle - \langle k\mathbf{x}, \mathbf{x} \rangle = \overline{\langle atB\mathbf{x}, \mathbf{y} \rangle} - \overline{k} = \overline{k} - \overline{k} = 0$ □

Theorem 8.2.4. *$\boldsymbol{A}\mathbf{x} = \lambda \boldsymbol{B}\mathbf{x}$, $\boldsymbol{A}^*\mathbf{x} = \lambda^* \boldsymbol{B}^*\mathbf{x}$ and $\langle \mathbf{x}, \boldsymbol{B}\mathbf{y} \rangle = \langle \boldsymbol{B}\mathbf{x}, \mathbf{y} \rangle$. One can construct a collection of B-orthogonal eigenvectors.*

Proof. Under the Condition 8.2.1, this is the direct conclusion from previous theorems. □

Theorem 8.2.5. *$\boldsymbol{B}$ is and Hermitian matrix and $\boldsymbol{B} = \boldsymbol{L}\boldsymbol{L}^*$. If $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{U}\boldsymbol{D}\boldsymbol{U}^*\boldsymbol{L}^*$, where $\boldsymbol{U}$ is a unitary matrix and $\boldsymbol{D}$ is a diagnol matrix, this matrix pair ($\boldsymbol{A}$, (B)) satisfied our conditions. That is, $\boldsymbol{L}^{-1}\boldsymbol{A}\boldsymbol{L}^{*-1}$ is a normal matrix.*

Proof.

$$\boldsymbol{A}\mathbf{x} = \lambda \boldsymbol{B}\mathbf{x}$$

$$\boldsymbol{L}\boldsymbol{M}\boldsymbol{D}\boldsymbol{M}^*\boldsymbol{L}^*\mathbf{x} = \lambda \boldsymbol{L}\boldsymbol{L}^* x$$

$$(\boldsymbol{L}\boldsymbol{M})\boldsymbol{D}(\boldsymbol{M}^*\boldsymbol{L}^*)\mathbf{x} = \lambda(\boldsymbol{L}\boldsymbol{M})(\boldsymbol{M}^*\boldsymbol{L}^*)x$$

$$\boldsymbol{D} is a diagnal matrix, so \lambda are the diagnal value$$

$$\boldsymbol{A}^*\mathbf{x} = \mu \boldsymbol{B}^*\mathbf{x}$$

$$\boldsymbol{L}\boldsymbol{M}\boldsymbol{D}^*\boldsymbol{M}^*\boldsymbol{L}^*\mathbf{x} = \mu \boldsymbol{L}\boldsymbol{L}^* x$$

$$\boldsymbol{D}^* is a diagnal matrix, so \mu are the diagnal value$$

Thus, μ is conjugate of λ □

8.3 Cases

1. A is a symmetric matrix, and B is a symmetric positive definite matrix.

doi:10.6342/NTU201803532

2. A is a Hermitian matrix, and B is a Hermitian positive definite matrix.

3. B is a Hermitian positive definite matrix, and $L^{-1}AL^{*-1}$ is a normal matrix

These cases satisfy our condition. The eigenvalues can be complex in some cases.

doi:10.6342/NTU201803532

Chapter 9

Algorithm

Denote

$$\rho(\boldsymbol{A}, \boldsymbol{B}, \mathbf{z}, \mathbf{w})Q = \sum_{i=1}^{N} \mathbf{w}_i (\mathbf{z}_i \boldsymbol{B} - \boldsymbol{A})^{-1} \boldsymbol{B}\mathbf{q}$$

,where **z** and **w** are quadratic points shown in Section 7.2

9.1 Estimation

We choose the estimation method shown in [6] becasue its linear systems are similar with those of FEAST(Algorithm 7). Thus, we can reuse the implementation of solving the linear systems.

Algorithm 6 estimate the number of eigenvalue in the circle in [6]

Require: $\boldsymbol{A}, \boldsymbol{B}, \mathbf{z}, \mathbf{w}$
Ensure: the estimated number of eigenvalue in the circle
1: Initial the random matrix: $\boldsymbol{Q}$
2: Approximate subspace projection: $\boldsymbol{Y} \leftarrow \rho(\boldsymbol{A}, \boldsymbol{B}, \mathbf{z}, \mathbf{w})\boldsymbol{Q}$
3: Calculate the trace: $\frac{1}{P} Q^\top Y$

9.2 FEAST

The Algorithm 7 is declared in [26]. We use this algorithm as our main eigensolver.

doi:10.6342/NTU201803532

Algorithm 7 FEAST

Require: $\boldsymbol{Q}^{(0)}, \boldsymbol{A}, \boldsymbol{B}, \mathbf{z}, \mathbf{w}$
Ensure: $\boldsymbol{Q}^{(k)}, \boldsymbol{\Lambda}^{(k)}$
1: Set $k \leftarrow 1$
2: Approximate subspace projection: $\boldsymbol{Y}^{(k)} \leftarrow \rho(\boldsymbol{A}, \boldsymbol{B}, \mathbf{z}, \mathbf{w})\boldsymbol{Q}^{(k-1)}$
3: Form reduced system: $\hat{\boldsymbol{A}}^{(k)} \leftarrow \boldsymbol{Y}^{(k)*}\boldsymbol{A}\boldsymbol{Y}^{(k)}$, $\hat{\boldsymbol{B}}^{(k)} \leftarrow \boldsymbol{Y}^{(k)*}\boldsymbol{B}\boldsymbol{Y}^{(k)}$
4: Solve eigenproblem: $\hat{\boldsymbol{A}}^{(k)}\hat{\boldsymbol{X}}^{(k)} = \hat{\boldsymbol{B}}^{(k)}\hat{\boldsymbol{X}}^{(k)}\hat{\boldsymbol{\Lambda}}^{(k)}$ for $\hat{\boldsymbol{\Lambda}}^{(k)}$, $\hat{\boldsymbol{X}}^{(k)}$
5: Set $\boldsymbol{Q}^{(k)} \leftarrow \boldsymbol{Y}^{(k)}\hat{\boldsymbol{X}}^{(k)}$
6: Set $\boldsymbol{\Lambda}^{(k)} = \hat{\boldsymbol{\Lambda}}^{(k)}$
7: Set $k \leftarrow k + 1$

9.3 Locking Technique

The locking technique is shown in [28] and the subspace iteration with locking is discussed in [39], suggesting the case of the algorithm when A is a Hermitian matrix and B is identity. Here, we generalize such algorithm for generalized eigenvalue problem. It can apply to the

Algorithm 8 Subspace iteration with locking method

Require: $\boldsymbol{Q}^{(0)}, \boldsymbol{A}, \mathbf{z}, \mathbf{w}$
Ensure: $\boldsymbol{Q}^{(k)}, \boldsymbol{\Lambda}^{(k)}$
1: Set $k \leftarrow 1, j \leftarrow 0$
2: Approximate subspace projection: $\hat{\boldsymbol{Y}}^{(k)} \leftarrow \left[\boldsymbol{Q}_j^{(k-1)}, \rho(\boldsymbol{A}, \boldsymbol{I}, \mathbf{z}, \mathbf{w})\boldsymbol{Q}_{m-j}^{(k-1)}\right]$
3: Orthonormalize the column vectors of $\hat{\boldsymbol{Y}}^{(k)} = \left[\hat{\boldsymbol{Y}}_j^{(k)}, \hat{\boldsymbol{Y}}_{m-j}^{(k)}\right]$ into $\boldsymbol{Y}^{(k)}$ (first j columns will be invariant such that $\boldsymbol{Y}^{(k)} = \left[\boldsymbol{Y}_j^{(k)}, \boldsymbol{Y}_{m-j}^{(k)}\right]$, which $\boldsymbol{Y}_j^{(k)} = \boldsymbol{Q}_j^{(k-1)}$, $\boldsymbol{Y}_{m-j}^{(k)} = \hat{\boldsymbol{Y}}_{m-j}^{(k)} - \boldsymbol{Q}_j^{(k-1)}(\boldsymbol{Q}_j^{(k-1)})^*\hat{\boldsymbol{Y}}_{m-j}^{(k)}$
4: Form reduced system: $\hat{\boldsymbol{A}}^{(k)} \leftarrow (\boldsymbol{Y}_{m-j}^{(k)})^*\boldsymbol{A}\boldsymbol{Y}_{m-j}^{(k)}$
5: Solve eigenproblem:$\hat{\boldsymbol{A}}^{(k)}\hat{\boldsymbol{X}}^{(k)} = \hat{\boldsymbol{X}}^{(k)}\hat{\boldsymbol{\Lambda}}^{(k)}$ for $\hat{\boldsymbol{\Lambda}}^{(k)}$, $\hat{\boldsymbol{X}}^{(k)}$
6: Set $\boldsymbol{Q}^{(k)} \leftarrow \left[\boldsymbol{Q}_j^{(k-1)}, \boldsymbol{Y}_{m-j}\hat{\boldsymbol{X}}^{(k)}\right]$
7: Set $\boldsymbol{\Lambda}^{(k)} = \hat{\boldsymbol{\Lambda}}^{(k)}$
8: Test the eigenvalues for convergence. Let i_{conv} be the number of newly converged eigenvalues. Set $j \leftarrow j + i_{conv}$.
9: Change the eigenvectors ordering in $\boldsymbol{Q}^{(k)}$ to make $\boldsymbol{Q}_j^{(k)}$ are the converged eignevectors.
10: Set $k \leftarrow k + 1$

problem with the condition Condition 8.2.1. The first j_0 column of $\boldsymbol{Q}^{(0)}$ is the eigenvectors solved in the previous sub-regions.

doi:10.6342/NTU201803532

9.4 FEAST with locking on general propose

Algorithm 9 FEAST with locking

Require: $\boldsymbol{Q}^{(0)}, \boldsymbol{A}, \boldsymbol{B}, \mathbf{z}, \mathbf{w}, j_0$
Ensure: $\boldsymbol{Q}^{(k)}, \boldsymbol{\Lambda}^{(k)}$

1: Set $k \leftarrow 1, j \leftarrow j_0$
2: Approximate subspace projection: $\hat{\boldsymbol{Y}}^{(k)} \leftarrow \left[\boldsymbol{Q}_j^{(k-1)}, \rho(\boldsymbol{A}, \boldsymbol{B}, \mathbf{z}, \mathbf{w})\boldsymbol{Q}_{m-j}^{(k-1)}\right]$
3: B-orthonormalize the column vectors of $\hat{\boldsymbol{Y}}^{(k)} = \left[\hat{\boldsymbol{Y}}_j^{(k)}, \hat{\boldsymbol{Y}}_{m-j}^{(k)}\right]$ into $\boldsymbol{Y}^{(k)}$ (first j columns will be invariant such that $\boldsymbol{Y}^{(k)} = \left[\boldsymbol{Y}_j^{(k)}, \boldsymbol{Y}_{m-j}^{(k)}\right]$, which $\boldsymbol{Y}_j^{(k)} = \boldsymbol{Q}_j^{(k-1)}$, $\boldsymbol{Y}_{m-j}^{(k)} = \hat{\boldsymbol{Y}}_{m-j}^{(k)} - \boldsymbol{Q}_j^{(k-1)}(\boldsymbol{B}\boldsymbol{Q}_j^{(k-1)})^* \hat{\boldsymbol{Y}}_{m-j}^{(k)}$
4: Form reduced system: $\hat{\boldsymbol{A}}^{(k)} \leftarrow (\boldsymbol{Y}_{m-j}^{(k)})^* \boldsymbol{A}\boldsymbol{Y}_{m-j}^{(k)}$, $\hat{\boldsymbol{B}}^{(k)} \leftarrow (\boldsymbol{Y}_{m-j}^{(k)})^* \boldsymbol{B}\boldsymbol{Y}_{m-j}^{(k)}$
5: Solve eigenproblem:$\hat{\boldsymbol{A}}^{(k)}\hat{\boldsymbol{X}}^{(k)} = \hat{\boldsymbol{B}}^{(k)}\hat{\boldsymbol{X}}^{(k)}\hat{\boldsymbol{\Lambda}}^{(k)}$ for $\hat{\boldsymbol{\Lambda}}^{(k)}$, $\hat{\boldsymbol{X}}^{(k)}$
6: Set $\boldsymbol{Q}^{(k)} \leftarrow \left[\boldsymbol{Q}_j^{(k-1)}, \boldsymbol{Y}_{m-j}\hat{\boldsymbol{X}}^{(k)}\right]$
7: Set $\boldsymbol{\Lambda}^{(k)} = \hat{\boldsymbol{\Lambda}}^{(k)}$
8: Test the eigenvalues for convergence. Let i_{conv} be the number of newly converged eigenvalues Set $j \leftarrow j + i_{conv}$.
9: Change the eigenvectors ordering in $\boldsymbol{Q}^{(k)}$ to make $\boldsymbol{Q}_j^{(k)}$ are the converged eigenvectors.
10: Set $k \leftarrow k + 1$

When the matrices pairs $\boldsymbol{A}, \boldsymbol{B}$ satisfied our condition shown before, we can use Algorithm 9 to solve its generalized eigenvalue problems.

9.5 Processing Flow

We show a flexible flow in Figure 9.1, we use this flow to solve the pentacene application problem.

doi:10.6342/NTU201803532

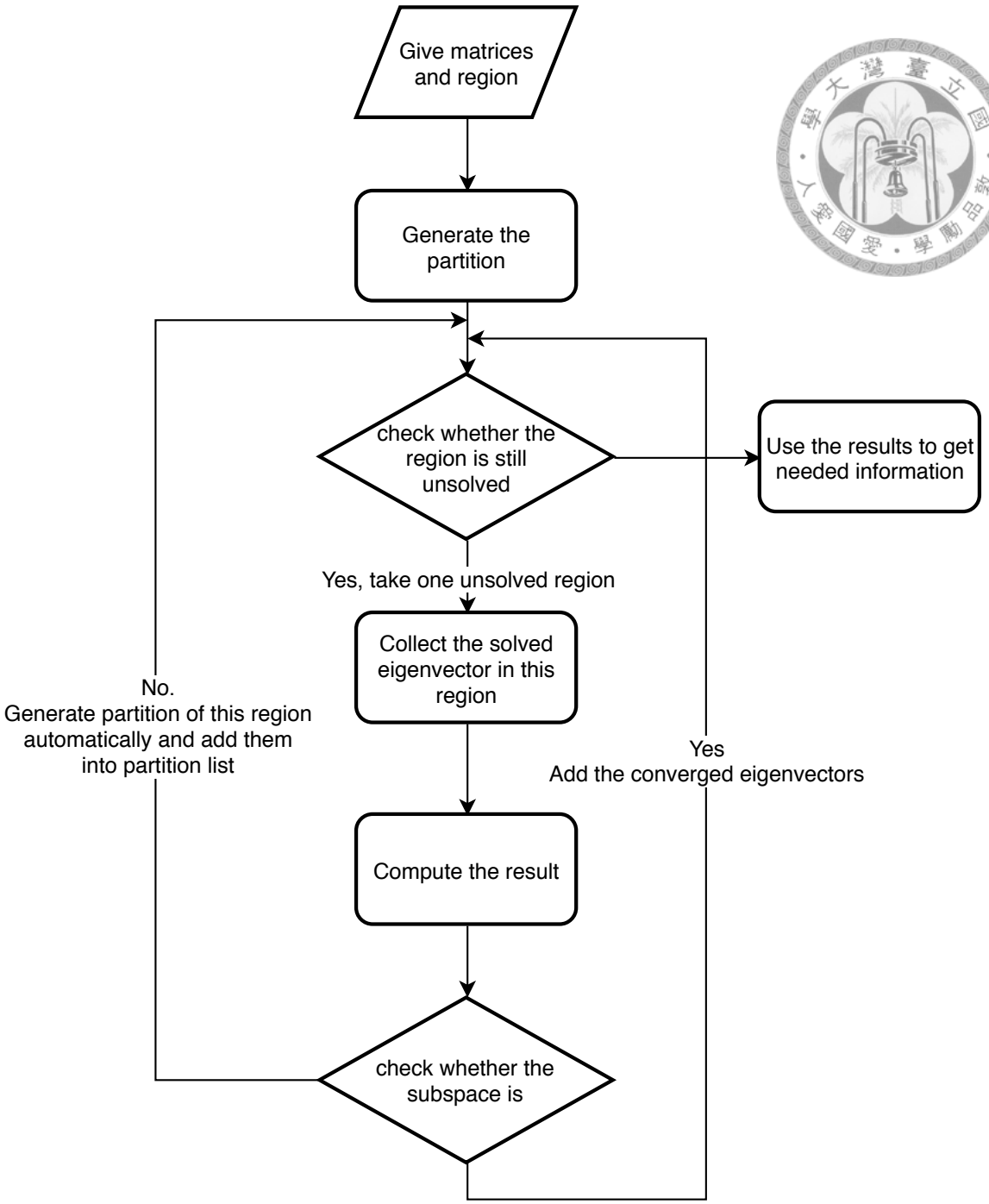


Figure 9.1: the processing flow

doi:10.6342/NTU201803532

Chapter 10

Implementation

10.1 Linear Solver

The critical part is the step for solving the linear systems, so the fast linear system is helpful for the eigensolver based on contour integral. There are many libraries to solve linear systems. For sparse iterative solver, MAGMA provides the solution on the GPU, Paralution, and Petsc provides the solution on the CPU. For sparse direct solver, MUMPS, PARDISO also provide the solution on CPU. Among them, we choose MKL Pardiso as our solver because MKL Pardiso is robust direct solver and it is faster than other methods in this application.

MKL Pardiso (Parallel Direct Sparse Solver) has three part to solve linear systems.

1. Analysis: Fill-reduction analysis and symbolic factorization
2. Factorize: Numerical factorization
3. Solve: Forward and Backward solve including optional iterative refinement

First, in the solve part, MKL pardiso allows us to solve transpose/hermitian part without analyzing and factorization. It is very suitable for our application because the matrices of the upper part and lower part are Hermitian. It can reduce almost half of solving time.

Second, because MKL pardiso is a direct sparse solver, we can store the factorization in the first loop. Then, we can only do solving part in later loops. The factorization needs much

doi:10.6342/NTU201803532

memory, so it can not be used in huge matrices. For example, storing the factorization is fine in PENTF20400, PENTF98736, and PENTF 183600, but it fails in PENTF734400. MKL pardiso reduce the solving time by storing the factorization can make us focus on the partition.

10.2 Data Structure

In the beginning time, we implement the eigensolver on GPU by MAGMA. However, we have tried different iterative solvers for PENTF application, but their performance is not faster than MKL pardiso due to the matrix structure. For now, we focus on the version on CPU which is done on MKL. We write some wrapper to handle the data structure by MAGMA. If we find some suitable iterative solver or fast direct solver on GPU, we can move on the GPU version very quickly.

We use the CSR to store the sparse data such as $\boldsymbol{A}$, $\boldsymbol{B}$, and column-major to store the dense data such as random matrices, eigenvectors. Because the function provided by NVIDIA almost only support the column-major function, so we use column-major to reduce the work which moving to GPU version.

10.3 Partition List

We will introduce partition in Chapter 11, we have several partition methods. Some of them depend on the pre-knowledge, and others depend on the estimation. For more quickly developing, we use MATLAB to estimate the partition from the previous result, so we need to use files to store the information. We store the center, radius, estimated number and whether the circle is split. Storing whether the circle is split is important because one of the algorithms needs the information to decide the residual setting. We can use it repeatedly and keep that the program result comes from the same list. Another benefit of the partition list, we can draw the circle picture easily in MATLAB not C++ like Figure 11.1

doi:10.6342/NTU201803532

and Figure 11.2

doi:10.6342/NTU201803532

doi:10.6342/NTU201803532

Chapter 11

Partition

We can use some small sub-regions to cover the given region and solve these small sub-regions directly. A pitfall of the deflation technique is that eigenpairs in the intersection will repeatedly be computed. To avoid such problem, one can move those eigenpairs outside these regions.

11.1 Partition and Estimation

We use two strategies, which are auto-partition and pre-knowledge partition, to divide the region into sub-regions. Auto partition only depends on the estimated number of eigenpairs, so the partition maybe is not very good. Pre-knowledge partition depends on the eigenvalue distribution of smaller cases. We estimate the number of eigenpairs in each sub-region. The sub-regions are not overlapped in Figure 11.1, but the sub-regions are overlapped in Figure 11.2.

The strategies for partition:

1. Auto partition:

   Calculate the estimation of the number of the eigenpairs in the region. When the estimated number of eigenpairs in the region is larger than the allowance, divide the region into sub-region uniformly. Check whether the estimated number of eigenpairs in each sub-regions are smaller than the allowance. Repeat this process until the estimated number of eigenpairs in all sub-regions are smaller than the allowance.

doi:10.6342/NTU201803532

2. Pre-knowlege partition:

   These cases of different sizes have similar eigenvalue distribution. Thus, we use the distribution of smaller cases to estimate the distribution of larger ones. Moreover, we can know the exact number of eigenpairs in [-0.42, -0.4]. There is a gap in [-0.43, -0.42]. We only use the estimated number of eigenpairs in [-0.48, -0.43]. Combine the information of the number of eigenpairs in each interval and the eigenvalue distribution of smaller cases to get a proper partition.

11.2 Conquering Partition

Figure 11.1 shows the partition we compute, the number after method is the max estimated number in each circle. The legends in Figure 11.1 shows '(center) radius estimated_number (actual_number)'. The strategies for overlapped partition:

1. scale(1.1):

   When we use auto partition strategy to compute the partition, we scale the radius of the middle circles by 1.1 when dividing the circle. It causes each circle will overlap with its neighborhood.

2. conquer_s:

   We found that FEAST can not solve every eigenpair in some regions with the small number of eigenpairs. We delete the region with the smallest number of eigenpairs and add the whole region to get the lost eigenpairs.

3. conquer_m:

   First, we order the sub-regions from the least to the largest according to the estimate of eigenpairs. Then we delete the first n-th sub-regions such that the sum of the deleted is the largest number that smaller than the allowance.

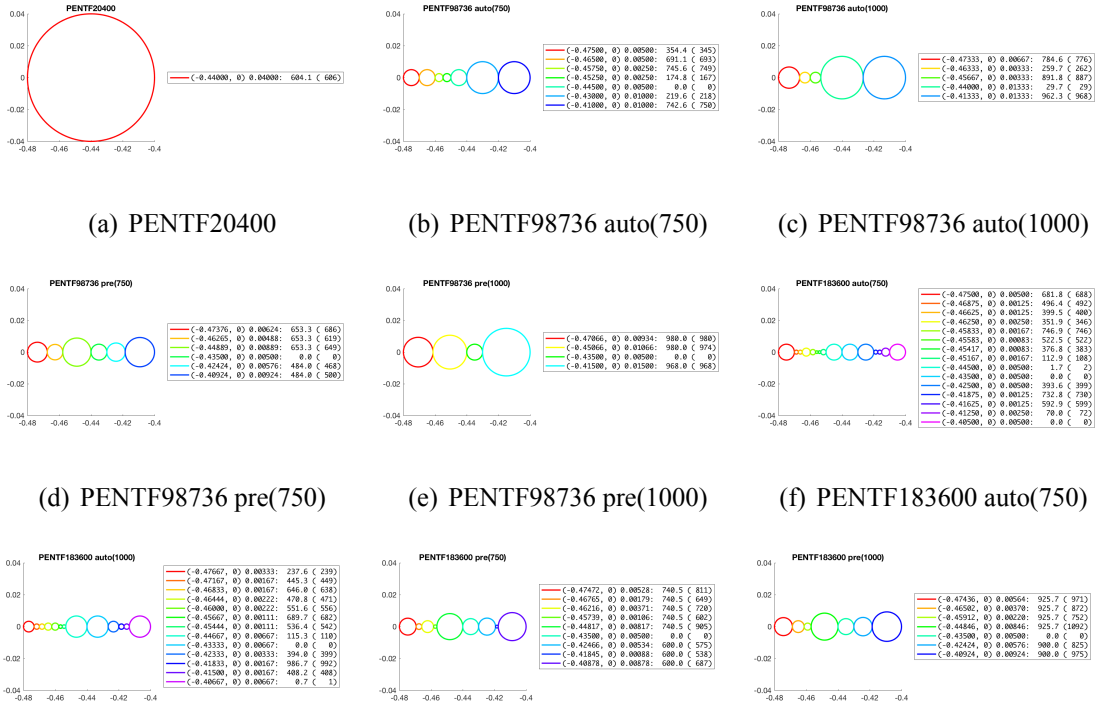
In Figure 11.2, we show the overlapping partition. In scale(1.1) cases, we use the arrow following by the text to specify the number in the intersection. The conquer_s and conquer_m partitions come from the previous no-overlapping partition with deleting some circles and add the original circle to conquer all sub-circles.

doi:10.6342/NTU201803532

Partition	Time (sec)
20400	23.4
98736: one	176.7
98736: auto(750)	1966.8
98736: auto(1000)	1427.2
183600: one	396.2
183600: auto(750)	9457.3
183600: auto(1000)	8150.8
734400: one	2880.2

Table 11.1: the computing time of generating list

When we use the pre-knowledge partition, we only compute the estimation of the whole region. It is another benefit in the pre-knowledge partition.



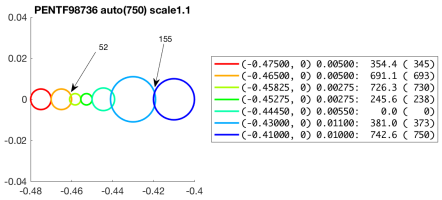
(a) PENTF20400 (b) PENTF98736 auto(750) (c) PENTF98736 auto(1000)

(d) PENTF98736 pre(750) (e) PENTF98736 pre(1000) (f) PENTF183600 auto(750)

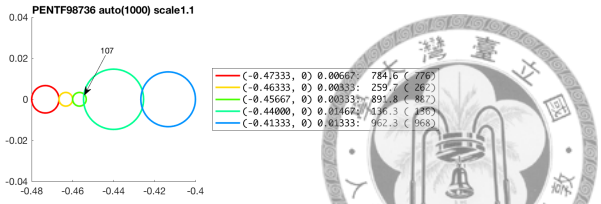
(g) PENTF183600 auto(1000) (h) PENTF183600 pre(750) (i) PENTF183600 pre(1000)

Figure 11.1: Partition

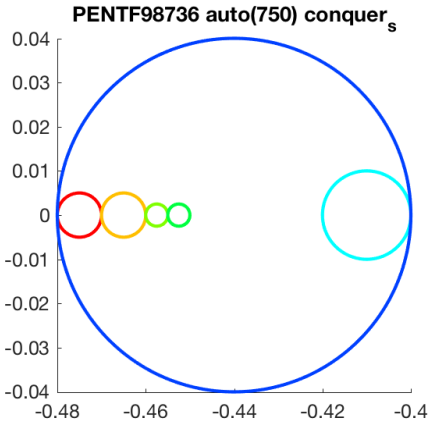
doi:10.6342/NTU201803532



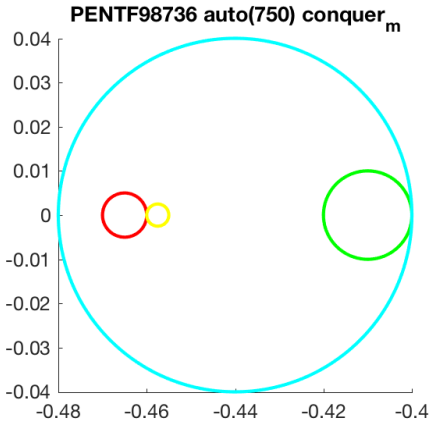
(a) PENTF98736 auto(750) scale1.1



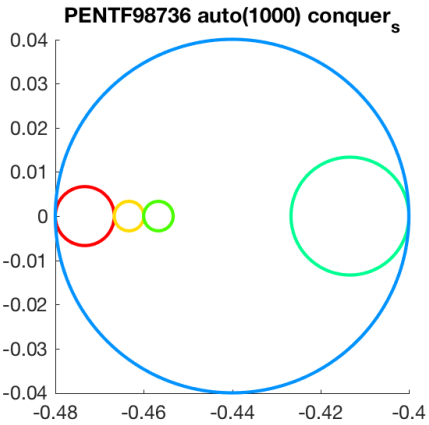
(b) PENTF98736 auto(1000) scale1.1



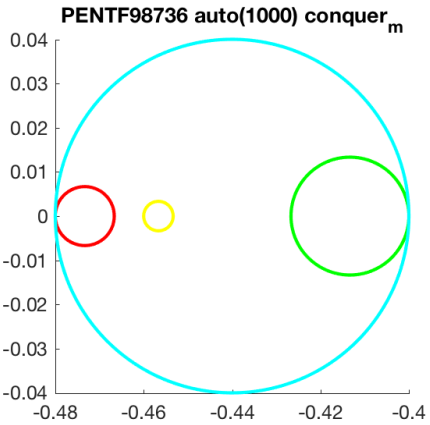
(c) PENTF98736 auto(750) conquer_s



(d) PENTF98736 auto(750) conquer_m



(e) PENTF98736 auto(1000) conquer_s



(f) PENTF98736 auto(1000) conquer_m

Figure 11.2: Overlapped Partition

doi:10.6342/NTU201803532

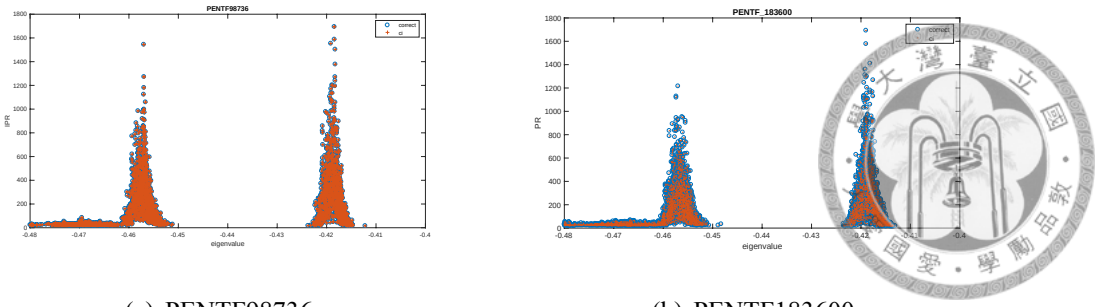
Chapter 12

Results

12.1 Application

We use PENTF matrices as our testing cases. PENTF stems from a two-dimensional thin-film (TF) of disordered pentacene (PEN), and it is a famous organic molecule. A typical target range of eigenvalues is $[-0.4, -0.48]$. Pentacene is one of most famous organic molecules, in particular, for the organic transistor, and can be found in [8]. In [25], they show the shows the experimental measurement of the participation ratio (PR) value, and it is important for device property. The all eigenpairs of PENTF cases are computed by EigenKernel [10, 15, 32] and the results are also shown in [9]. We know the exact number of eigenpairs in $[-0.43, -0.4]$, and it is $\frac{\text{the size of matrices}}{102}$.
All the matrix data stem from the electronic state calculations of pentacene thin films. The difference between them is the number of molecules in the calculated system. The eigenvalue distribution is similar to among the matrix data when the value is normalized by the matrix size. Consequently, the result of a smaller system can be used as the pre-knowledge for a larger system. Since systematic research from small systems into large systems is general in computational material science, the use of the preknowledge for large systems is fruitful among many researches.

doi:10.6342/NTU201803532



(a) PENTF98736 (b) PENTF183600

Figure 12.1: participation ratio

12.2 Environment

We run this code on the Reedbush whose CPUs are Intel Xeon E5-2695v4 (Broadwell-EP) with 36 threads and 2.1 GHz, and the ram is 256GB. We run the code with full threads (36) on one CPU.

The testing matrices can be found in [9], and we choose the PENTF cases of the sizes 20400, 98736, 183600.

The given region is $[-0.48, -0.4]$

There are 606, 2922 and 5487 eigenpairs in $[-0.48, -0.4]$ of PENTF 20400, PENTF 98736, and PENTF 183600. We set the residual is 10^{-12} in the eigensolver based on contour integral. The correct answers are computed by the mini-application of EigenKernel (https://github.com/eigenkernel)[10, 15, 32] that uses the dense-matrix solver algorithm in ScaLAPACK. We plot the eigenvalue - participation ratio by correct answer and our solver.

12.3 MKL FEAST vs FEAST vs FEAST with locking

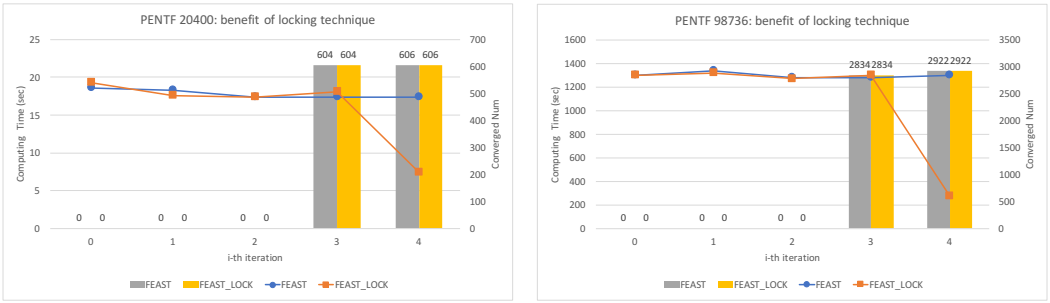
Our implementation is similar to the algorithm of MKL FEAST. We compare the time and the number of iteration of solving the problem in [-0.48, -0.4]. We do not know the detail of the implementation of MKL, so the time performance of ours and MKL's are not comparable. We show our base implementation is viable. We also compare with FEAST with locking to show that locking technique provides some benefits when those eigenpairs

doi:10.6342/NTU201803532

have different convergence rate.

time (sec)	PENTF 20400	PENTF 98736	PENTF 183600
MKL FEAST	534.7	16023.6	failed
FEAST	196.8	7215.3	failed
FEAST_LOCK	102.2	5650.9	failed

Table 12.1: Performance in one region



(a) PENTF20400 (b) PENTF98736

Figure 12.2: the benefit of locking technique

i-th iteration	MKL FEAST	FEAST	FEAST_LOCK
0	2.51E-03	1.99E-03	1.99E-03
1	2.02E-06	1.78E-06	1.78E-06
2	1.48E-09	1.42E-09	1.42E-09
3	1.82E-12	1.59E-12	1.59E-12
4	3.32E-14	2.69E-14	1.93E-14 (1E-12)

Table 12.2: PENTF 20400: max-residual

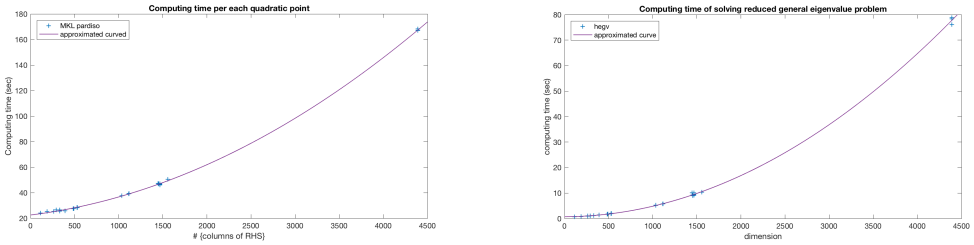
doi:10.6342/NTU201803532

i-th iteration	MKL FEAST	FEAST	FEAST_LOCK
0	5.78E-03	4.93E-03	4.93E-03
1	4.28E-06	3.78E-06	3.78E-06
2	3.27E-09	3.19E-09	3.19E-09
3	4.82E-12	4.92E-12	4.92E-12
4	1.53E-13	1.09E-13	1.07E-13 (1E-12)

Table 12.3: PENTF 98736: max-residual

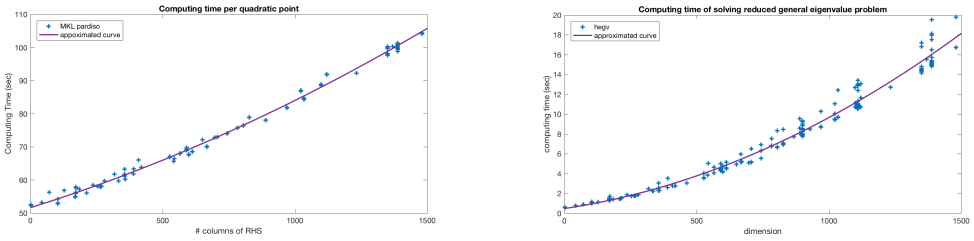
The maximum residual of each iteration step is similar. All of them solve the same number of converged eigenpairs in [-0.48, -0.4]. Although our implementation is faster than MKL's, it shows that our implementation can be a baseline.

12.4 Locking Effect



(a) linear systems (b) reduced eigenvalue problem

Figure 12.3: PENTF98736: detail performance



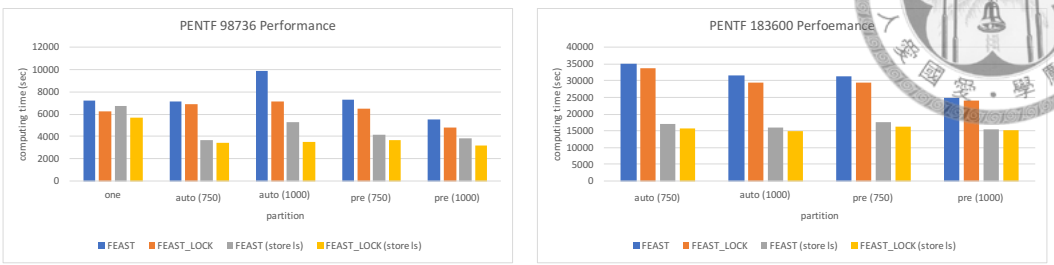
(a) linear systems (b) reduced eigenvalue problem

Figure 12.4: PENTF183600: detail performance

doi:10.6342/NTU201803532

12.5 Dividing Partition

Figure 12.5: Performance



(a) PENTF 98736 (b) PENTF 183600

Note. EigenKernel spends 8.7 hours(31320 secs) solving PENTG183600 with 32 nodes of Reedbush.

It fails when we use one interval [-0.48, -0.4] to solve PENTF 183600. However, when we divide the interval into several sub-regions, we can solve this problem easily. In PENTF 98736 case, the computing time of pre-knowledge partition with four methods is shorter than computing time of one partition, but the auto partition the two methods which store the result of matrix factorization by MKL pardiso are faster than one partition's.

The factorization is a heavy overhead in each iteration, so we also implement the algorithms which store the matrix factorization by MKL pardiso. Although it needs more memory to store the information, it can reduce the repeating work in each iteration. The speed of FEAST and FEAST_LOCK provided in the different partition are faster than solving one region.

12.6 Conquering Partition

Locking technique also works successfully on this overlapping cases. That is, we scale the radius of the middle circle to make each sub-regions has an intersection with its neighborhood. We do not solve the eigenvector done before again. This method can be used when we doubt the eigenpairs near the boundary of the circle. Furthermore, we can use a bigger circle to cover the smaller circle for picking up the lost eigenpairs.

doi:10.6342/NTU201803532

Dividing can make us solve the larger problem and sometimes accelerate the solving process, but it sometimes fails in sub-regions. For example, in the fourth circle of auto partition (750) of PENTF98736, FEAST run it in 10 iterations, but there are no eigenpairs converged.

Conquering can help us to solve those lost eigenpairs. We introduce the conquer method in the previous section. The conquer_s can get the lost eigenpairs successfully. The conquer_m get the issue of locking technique, it can not solve all lost eigenpairs due to the locking eigenpairs numerical error. We can solve the sub-region with higher precision (10^{-13}), and then we solve the conquering part, which is the whole region, with 10^{-12}. It can avoid the locking technique convergence problem if the process can arrive the desire residual in each circle.

PENTF 98736 Partition	Converged eigenpairs
auto(750)	2922
auto(1000)	2893
conquer_s(1000)	2922
conquer_m(750)	2192
conquer_s(750)	2922
conquer_m(1000)	2631
scale(1.1)(750)	3040
scale(1.1)(1000)	2893
conquer_s(750)(different error)	2922
conquer_m(750)(different error)	2922
conquer_s(1000)(different error)	2922
conquer_m(1000)(different error)	2922

In auto(1000) method of PENTF98736, the eigensolver based on Contour Integral does not solve all eigenpairs(2922). However, we can use the conquer method to pick up the missing eigenpairs.

doi:10.6342/NTU201803532

Chapter 13

Conclusion

We declare FEAST with the locking technique on the generalized eigenvalue problems under the conditions. The locking technique is helpful inside the FEAST, and it can reduce the computing time of solving linear systems and compute the reduced eigenproblem when some eigenpairs arrive the required residual early.

Dividing the partition can solve the large problem which fails on solving in only one region and sometimes can accelerate the whole process. Due to the smaller number of the eigenpairs in each circle than the given region, the solving time in each circle is less than one in the given region. Dividing do not ensure that the FEAST can solve all eigenpairs in sub-regions.

Conquering is based on the locking technique, and we use it to pick up the lost eigenpairs. FEAST might not find all eigenpairs in each sub-region by dividing, so we use a bigger circle to cover the given region and compute the lost eigenpairs.

Pre-knowledge is important for solving the eigenvalue problems. Pre-knowledge can allow us only to estimate the number of eigenpairs in the given region, but we get the excellent partition. Moreover, pre-knowledge partition only contains the smallest number of sub-regions, so we do not compute the factorization many times. Thus, we get the benefits of pre-knowledge on both of estimation and solving part.

We introduce the strategies for dividing and conquer with locking technique, and they work on PENTF cases well. Although the convergence problems in locking technique or FEAST, we can use different regions to make the results better based on those eigenpairs

doi:10.6342/NTU201803532

we solved without the expensive cost.

doi:10.6342/NTU201803532

Bibliography

[1] B. W. Bader and T. G. Kolda. Algorithm 862: Matlab tensor classes for fast algorithm prototyping. *ACM Transactions on Mathematical Software (TOMS)*, 32(4):635–653, 2006.

[2] L. De Lathauwer, B. De Moor, and J. Vandewalle. A multilinear singular value decomposition. *SIAM journal on Matrix Analysis and Applications*, 21(4):1253–1278, 2000.

[3] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-optimal parallel and sequential qr and lu factorizations. *SIAM Journal on Scientific Computing*, 34(1):A206–A239, 2012.

[4] J. W. Demmel. *Applied numerical linear algebra*, volume 56. Siam, 1997.

[5] E. Di Napoli, E. Polizzi, and Y. Saad. Efficient estimation of eigenvalue counts in an interval. *Numerical Linear Algebra with Applications*, 23(4):674–692, 2016.

[6] Y. Futamura, H. Tadano, and T. Sakurai. Parallel stochastic estimation method of eigenvalue distribution. *JSIAM Letters*, 2:127–130, 2010.

[7] G. H. Golub and C. F. Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

[8] M. L. Hammock, A. Chortos, B. C.-K. Tee, J. B.-H. Tok, and Z. Bao. 25th anniversary article: the evolution of electronic skin (e-skin): a brief history, design considerations, and recent progress. *Advanced materials*, 25(42):5997–6038, 2013.

[9] T. Hoshi. *ELSES matrix library*. http://www.elses.jp/matrix/.

doi:10.6342/NTU201803532

[10] T. Hoshi, H. Imachi, K. Kumahata, M. Terai, K. Miyamoto, K. Minami, and F. Shoji. Extremely scalable algorithm for 108-atom quantum material simulation on the full system of the k computer. In *Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA), 2016 7th Workshop on*, pages 33–40. IEEE, 2016.

[11] T. Hoshi, H. Imachi, S. Yokoyama, and T. Kaji. Numerical linear algebraic problems in large-scale massively parallel electronic state calculations on the k computer. In *International Workshop on Eigenvalue Problems: Algorithms; Software and Applications, in Petascale Computing*, 2015.

[12] T. Ikegami and T. Sakurai. Contour integral eigensolver for non-hermitian systems: a rayleigh-ritz-type approach. *Taiwanese Journal of Mathematics*, pages 825–837, 2010.

[13] T. Ikegami, T. Sakurai, and U. Nagashima. A filter diagonalization for generalized eigenvalue problems based on the sakurai–sugiura projection method. *Journal of Computational and Applied Mathematics*, 233(8):1927–1936, 2010.

[14] H. Imachi. *Numerical Methods for Large-scale Quantum Material Simulations*. PhD thesis, Tottori University, 2017.

[15] H. Imachi and T. Hoshi. Hybrid numerical solvers for massively parallel eigenvalue computations and their benchmark with electronic structure calculations. *Journal of Information Processing*, 24(1):164–172, 2016.

[16] A. Imakura, L. Du, and T. Sakurai. Accuracy analysis on the rayleigh-ritz type of the contour integral based eigensolver for solving generalized eigenvalue problems. Technical report, Citeseer, 2014.

[17] A. Kapteyn, H. Neudecker, and T. Wansbeek. An approach ton-mode components analysis. *Psychometrika*, 51(2):269–275, 1986.

[18] A. Kerr, D. Campbell, and M. Richards. Qr decomposition on gpus. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 71–78. ACM, 2009.

doi:10.6342/NTU201803532

[19] J. Kestyn, E. Polizzi, and P. T. Peter Tang. Feast eigensolver for non-hermitian problems. *SIAM Journal on Scientific Computing*, 38(5):S772–S799, 2016.

[20] E. Kofidis and P. A. Regalia. On the best rank-1 approximation of higher-order super-symmetric tensors. *SIAM Journal on Matrix Analysis and Applications*, 23(3):863–884, 2002.

[21] T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

[22] P. M. Kroonenberg and J. De Leeuw. Principal component analysis of three-mode data by means of alternating least squares algorithms. *Psychometrika*, 45(1):69–97, 1980.

[23] J. Levin. Three-mode factor analysis. *Psychological Bulletin*, 64(6):442, 1965.

[24] Y. Maeda, Y. Futamura, and T. Sakurai. Stochastic estimation method of eigenvalue density for nonlinear eigenvalue problem on the complex plane. *JSIAM letters*, 3:61–64, 2011.

[25] H. Matsui, A. S. Mishchenko, T. Hasegawa, et al. Distribution of localized states from fine analysis of electron spin resonance spectra in organic transistors. *Physical review letters*, 104(5):056602, 2010.

[26] P. T. Peter Tang and E. Polizzi. Feast as a subspace iteration eigensolver accelerated by approximate spectral projection. *SIAM Journal on Matrix Analysis and Applications*, 35(2):354–390, 2014.

[27] E. Polizzi. Density-matrix-based algorithm for solving eigenvalue problems. *Physical Review B*, 79(11):115112, 2009.

[28] Y. Saad. *Numerical Methods for Large Eigenvalue Problems: Revised Edition*. SIAM, 2011.

doi:10.6342/NTU201803532

[29] T. Sakurai, H. Tadano, et al. Cirr: a rayleigh-ritz type method with contour integral for generalized eigenvalue problems. *Hokkaido mathematical journal*, 36(4):745–757, 2007.

[30] O. Schenk and K. Gärtner. Solving unsymmetric sparse systems of linear equations with pardiso. *Future Generation Computer Systems*, 20(3):475–487, 2004.

[31] A. Stathopoulos. Locking issues for finding a large number of eigenvectors of hermitian matrices. Technical report, Tech Report WM-CS-2005-09, Computer Science, The College of William & Mary, 2005.

[32] K. Tanaka, H. Imachi, T. Fukumoto, T. Fukaya, Y. Yamamoto, and T. Hoshi. Eigenkernel-a middleware for parallel generalized eigenvalue solvers to attain high scalability and usability. *arXiv preprint arXiv:1806.00741*, 2018.

[33] L. R. Tucker. Implications of factor analysis of three-way matrices for measurement of change. *Problems in measuring change*, 15:122–137, 1963.

[34] L. R. Tucker. The extension of factor analysis to three-dimensional matrices. *Contributions to mathematical psychology*, 110119, 1964.

[35] L. R. Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31(3):279–311, 1966.

[36] N. Vannieuwenhoven, R. Vandebril, and K. Meerbergen. A new truncation strategy for the higher-order singular value decomposition. *SIAM Journal on Scientific Computing*, 34(2):A1027–A1052, 2012.

[37] M. A. O. Vasilescu and D. Terzopoulos. Multilinear analysis of image ensembles: Tensorfaces. In *European Conference on Computer Vision*, pages 447–460. Springer, 2002.

[38] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*, pages 167–188. Springer, 2014.

doi:10.6342/NTU201803532

[39] Y. Xi and Y. Saad. Computing partial spectra with least-squares rational filters. *SIAM Journal on Scientific Computing*, 38(5):A3020–A3045, 2016.

doi:10.6342/NTU201803532