Preprocessing Algorithm for the Generalized SVD on the Graphics Processing Units

Flegar, Goran

Master's thesis / Diplomski rad

2016

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Science / Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:217:008225

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2023-02-09**

Repository / Repozitorij:

Repository of the Faculty of Science - University of Zagreb

UNIVERSITY OF ZAGREB

FACULTY OF SCIENCE

DEPARTMENT OF MATHEMATICS

Goran Flegar

PREPROCESSING ALGORITHM FOR THE GENERALIZED SVD ON THE GRAPHICS PROCESSING UNITS

Diploma Thesis

Advisor:
prof. dr. sc. Sanja Singer

Zagreb, 2016

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

To Jelena

Contents

Introduction

Generalized singular value decomposition (GSVD) has proved to be a useful tool in many areas, some of them being comparative analysis of the genome-scale expression data sets, incomplete singular boundary element method and ionospheric tomography. It has also found it's use in other problems from the linear algebra domain, such as the generalized eigenvalue problem, the generalized total least squares and the linear equality or inequality constrained least squares.

Paige's algorithm [12] is the most widely used method for computing the GSVD. The method consists of two steps: a preprocessing step where each of the matrices from the matrix pair (A, B) is reduced to triangular form, followed by an iterative method based on the implicit Kogbetliantz SVD procedure. This method is implemented in LAPACK's xGGSVD routine, with the preprocessing step implemented in xGGSVP. The same preprocessing step has been proposed in [11] in case B is not of full column rank. In this thesis we will study a parallel modification of this algorithm.

Around 2006 Dennard scaling [7], which roughly states that smaller transistors can operate on higher frequencies using the same amount of power, started to fail [8] and prevented further increases in CPU frequencies. This forced the CPU manufacturers to explore new venues in processor design. Graphics processing units (GPUs) became one of those venues resulting in the release of the CUDA programming platform in 2007, which enabled general-purpose processing on GPUs (GPGPU) manufactured by Nvidia. Since then the GPGPU popularity increased and is now widely used in scientific computing and industry with libraries available in a wide spectrum of domains including linear algebra, signal processing, data mining and many others. The wide-spread and efficiency of GPUs is precisely the reason for studying the GSVD preprocessing algorithm in the context of GPGPU and the CUDA platform.

The rest of this thesis is organized into three chapters. In Chapter 1 we examine the generalized SVD decomposition and the upper triangular decomposition of a matrix pair resulting from the preprocessing algorithm. Chapter 2 deals with the implementation of this algorithm using the CUDA programming platform. Finally, the algorithm is tested on a variety of matrix pairs of different dimensions and the results of these tests are presented in Chapter 3.

Chapter 1

Preprocessing Algorithm for the Generalized SVD

In this chapter we define the generalized singular value decomposition and the upper triangular decomposition resulting from the preprocessing algorithm. We also describe the steps required to obtain this upper triangular decomposition.

1.1 The Generalized SVD and the Preprocessing Decomposition

Before diving into the preprocessing step of the GSVD algorithm, the following theorem and remark state some facts about the generalized SVD and show its connection to the standard singular value decomposition.

Theorem 1.1 (Generalized SVD [1])**.** *Let $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{p\times n}$ be matrices and let $r = \operatorname{rank}([A^T\ B^T]^T)$. Then there exist orthogonal $U \in \mathbb{R}^{m\times m}$, $V \in \mathbb{R}^{p\times p}$ and $Q \in \mathbb{R}^{n\times n}$ and nonsingular upper triangular $R \in \mathbb{R}^{r\times r}$ such that*

$$U^T A Q = \Sigma_A \begin{bmatrix} 0 & R \end{bmatrix}, \quad V^T B Q = \Sigma_B \begin{bmatrix} 0 & R \end{bmatrix},$$

where Σ_A and Σ_B are of the following form:

1. *If* $m \geq r$*:*

$$\Sigma_A = \begin{array}{c} \\ k \\ l \\ m-r \end{array} \begin{array}{c} \begin{array}{cc} k & l \end{array} \\ \begin{bmatrix} I & 0 \\ 0 & C \\ 0 & 0 \end{bmatrix} \end{array},$$

$$\Sigma_B = \begin{array}{c} \\ l \\ p-l \end{array} \begin{array}{c} \begin{array}{cc} k & l \end{array} \\ \begin{bmatrix} 0 & S \\ 0 & 0 \end{bmatrix} \end{array}.$$

2. *If* $m < r$*:*

$$\Sigma_A = \begin{array}{c} \\ k \\ m-k \end{array} \begin{array}{c} \begin{array}{ccc} k & m-k & r-m \end{array} \\ \begin{bmatrix} I & 0 & 0 \\ 0 & C & 0 \end{bmatrix} \end{array}$$

$$\Sigma_B = \begin{array}{c} \\ m-k \\ r-m \\ p-l \end{array} \begin{array}{c} \begin{array}{ccc} k & m-k & r-m \end{array} \\ \begin{bmatrix} 0 & S & 0 \\ 0 & 0 & I \\ 0 & 0 & 0 \end{bmatrix} \end{array}$$

Here $l = \operatorname{rank}(B)$*,* $k = r - l$ *and* C *and* S *are nonnegative diagonal matrices such that* $C^2 + S^2 = I$*.*

Remark 1.2. *The decomposition from Theorem 1.1 is sometimes referred to as the "Triangular Form" of the GSVD [3]. There is also an alternative decomposition called the "Diagonal Form":*

$$U^T A X = \Sigma_A \begin{bmatrix} 0 & I_{r\times r} \end{bmatrix}, \quad V^T B X = \Sigma_B \begin{bmatrix} 0 & I_{r\times r} \end{bmatrix},$$

where U*,* V*,* Σ_A *and* Σ_B *are as in Theorem 1.1 and* X *is a nonsingular matrix of order* n*. The diagonal form can be obtained from the triangular form by defining* X *as*

$$X = \begin{bmatrix} I & 0 \\ 0 & R^{-1} \end{bmatrix}.$$

Let $\Sigma_A^T \Sigma_A = \operatorname{diag}(\alpha_1^2, \ldots, \alpha_r^2)$ *and* $\Sigma_B^T \Sigma_B = \operatorname{diag}(\beta_1^2, \ldots, \beta_r^2)$ *where* $0 \leq \alpha_i, \beta_i \leq 1$ *for* $i = 1, \ldots, r$*. The pairs* (α_i, β_i) *are called the* ***generalized singular value pairs*** *and the*

quotients $\sigma_i = \alpha_i/\beta_i$, which may be infinite if $\beta_i = 0$, are called the ***generalized singular values****.*

The final remark regarding the GSVD is the justification of its name. Suppose that B is square and nonsingular. Then we have

$$U^T(AB^{-1})V = (U^TAQ)(V^TBQ)^{-1} = (\Sigma_A R)(\Sigma_B R)^{-1} = \Sigma_A\Sigma_B^{-1}$$

so the SVD of the matrix AB^{-1} can be computed from the GSVD of the pair (A, B). The generalized singular values of the pair (A, B) are the singular values of AB^{-1}. In particular, if $B = I$ then the GSVD of (A, B) is equal to the SVD of A.

Now we turn to the preprocessing step for the GSVD. The following theorem introduces the goal of the preprocessing step, the triangular decomposition of the matrix pair (A, B).

Theorem 1.3 (GSVD preprocessing step [5])**.** *Let $A \in \mathbb{R}^{m\times n}$, $B \in \mathbb{R}^{p\times n}$ and r, k and l be as in Theorem 1.1. Then there exist orthogonal $U \in \mathbb{R}^{m\times m}$, $V \in \mathbb{R}^{p\times p}$ and $Q \in \mathbb{R}^{n\times n}$ such that*

$$U^TAQ = \tilde{A}, \quad V^TBQ = \tilde{B},$$

where

$$\tilde{B} = \begin{array}{c} \\ l \\ p-l \end{array} \begin{array}{c} \begin{array}{ccc} n-r & k & l \end{array} \\ \begin{bmatrix} 0 & 0 & B_{13} \\ 0 & 0 & 0 \end{bmatrix} \end{array},$$

and $\tilde{A}$ is of one of the following forms:

1. *If $m \geq r$:*

$$\tilde{A} = \begin{array}{c} \\ k \\ l \\ m-r \end{array} \begin{array}{c} \begin{array}{ccc} n-r & k & l \end{array} \\ \begin{bmatrix} 0 & A_{12} & A_{13} \\ 0 & 0 & A_{23} \\ 0 & 0 & 0 \end{bmatrix} \end{array}.$$

2. *If $m < r$:*

$$\tilde{A} = \begin{array}{c} \\ k \\ m-k \end{array} \begin{array}{c} \begin{array}{ccc} n-r & k & l \end{array} \\ \begin{bmatrix} 0 & A_{12} & A_{13} \\ 0 & 0 & A_{23} \end{bmatrix} \end{array}.$$

B_{13} and A_{12} are nonsingular upper triangular and A_{23} is upper trapezoidal.

From this theorem it is obvious that after finding the GSVD of the triangular pair $(\tilde{A}, \tilde{B})$ we also get the GSVD of (A, B). Furthermore, if we have a GSVD of (A_{23}, B_{13})

$$\tilde{U}A_{23}\tilde{Q}^T = \Sigma_{A_{23}}\tilde{R}, \quad \tilde{V}B_{13}\tilde{Q}^T = \Sigma_{B_{13}}\tilde{R},$$

we can also get the GSVD of $(\tilde{A}, \tilde{B})$:

$$\begin{bmatrix} I & 0 \\ 0 & \tilde{U} \end{bmatrix} \tilde{A} \begin{bmatrix} I & 0 \\ 0 & \tilde{Q}^T \end{bmatrix} = \begin{bmatrix} 0 & A_{12} & A_{23}\tilde{Q}^T \\ 0 & 0 & \tilde{U}A_{23}\tilde{Q}^T \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0 & \Sigma_{A_{23}} \end{bmatrix} \begin{bmatrix} 0 & A_{12} & A_{23}\tilde{Q}^T \\ 0 & 0 & \tilde{R} \end{bmatrix},$$
$$\begin{bmatrix} \tilde{V} & 0 \\ 0 & I \end{bmatrix} \tilde{B} \begin{bmatrix} I & 0 \\ 0 & \tilde{Q}^T \end{bmatrix} = \begin{bmatrix} 0 & 0 & \tilde{V}B_{13}\tilde{Q}^T \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & \Sigma_{B_{13}} \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & A_{12} & A_{23}\tilde{Q}^T \\ 0 & 0 & \tilde{R} \end{bmatrix}.$$

Thus, we have reduced the GSVD of a general matrix pair to the GSVD of an upper triangular pair with nonsingular B, which can be computed using the implicit Kogbetliantz [12, 4] or the implicit Hari–Zimmerman [11] algorithm.

1.2 Finding the Preprocessing Decomposition

In this section we will prove Theorem 1.3 by constructing the method to compute the triangular decomposition. The building blocks of this algorithm will be the well-known QR decomposition and QR decomposition with column pivoting as well as the complete orthogonal decomposition (URV), whose existence is proved below.

Proposition 1.4 (Complete orthogonal decomposition)**.** *Given a matrix $A \in \mathbb{R}^{m\times n}$ of rank r there exist orthogonal matrices $U \in \mathbb{R}^{m\times m}$ and $V \in \mathbb{R}^{n\times n}$ such that*

$$U^T A V = \begin{array}{c} \\ r \\ m-r \end{array} \begin{array}{c} \begin{array}{cc} n-r & r \end{array} \\ \begin{bmatrix} 0 & R \\ 0 & 0 \end{bmatrix} \end{array},$$

where R is nonsingular upper triangular.

Proof. In order to obtain the complete orthogonal decomposition it is necessary to find the rank of the matrix A. An efficient method of doing that, which also brings the structure of the resulting matrix closer to our goal, is the QR decomposition with column pivoting

$$Q^T A P = \tilde{R},$$

where Q is orthogonal, P is a permutation matrix (which is also orthogonal) and $\tilde{R}$ is upper triangular with the additional property that the absolute value of each diagonal element $\tilde{R}_{ii}$ is larger than the Euclidian norm of each of the columns of $\tilde{R}_{(i+1):m,(i+1):n}$. Here, $\tilde{R}_{(i+1):m,(i+1):n}$ is the lower right submatrix of $\tilde{R}$ starting at position $(i+1, i+1)$. This means that if a diagonal entry of R is zero, then the whole trailing matrix is also a zero matrix, so $\tilde{R}$ is of the form

$$\tilde{R} = \begin{array}{c} \\ r \\ m-r \end{array} \begin{array}{c} \begin{array}{cc} r & n-r \end{array} \\ \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix} \end{array}.$$

where $r \leq \min\{m, n\}$ and R_{11} is upper triangular with nonzero diagonal entries. From this it is obvious that $\tilde{R}$ has exactly r linearly independent columns, i.e. $\operatorname{rank}(\tilde{R}) = r$. Since orthogonal transformations preserve rank, the rank of the matrix A is also r. Let $\tilde{R} = [\tilde{r}_1, \ldots, \tilde{r}_m]^T$, where $\tilde{r}_i$ are column vectors, i.e. $\tilde{r}_i^T$ are the rows of $\tilde{R}$. The next step is to find an orthogonal matrix V_1 such that $V_1^T \tilde{r}_r = \|\tilde{r}_r\|_2 e_n$. This can be achieved by a series of Givens rotations or by a Householder reflection. Then we apply V_1 to $\tilde{R}$ from the right side obtaining

$$\tilde{R}V_1 = \begin{array}{c} \\ r-1 \\ 1 \\ m-r \end{array} \begin{array}{c} \begin{array}{cc} n-1 & 1 \end{array} \\ \begin{bmatrix} R_{11}^{(1)} & R_{12}^{(1)} \\ 0 & R_{22}^{(1)} \\ 0 & 0 \end{bmatrix} \end{array},$$

with $R_{22}^{(1)} \neq 0$. Note that we can chose V_1 in such a way that $R_{11}^{(1)}$ keeps its upper trapezoidal structure, but this is not necessary. We continue with the same procedure recursively on $R_{11}^{(1)}$ obtaining

$$R_{11}^{(1)}V_2 = \begin{array}{c} \\ r-1 \\ m-r \end{array} \begin{array}{c} \begin{array}{cc} n-r & r-1 \end{array} \\ \begin{bmatrix} 0 & R_{12}^{(2)} \\ 0 & 0 \end{bmatrix} \end{array},$$

with V_2 orthogonal and $R_{12}^{(2)}$ upper triangular, and

$$\tilde{R}V_1 \begin{bmatrix} V_2 & 0 \\ 0 & 1 \end{bmatrix} = \begin{array}{c} \\ r-1 \\ 1 \\ m-r \end{array} \begin{array}{c} \begin{array}{ccc} n-r & r-1 & 1 \end{array} \\ \begin{bmatrix} 0 & R_{12}^{(2)} & R_{12}^{(1)} \\ 0 & 0 & R_{22}^{(1)} \\ 0 & 0 & 0 \end{bmatrix} \end{array},$$

which gives us the required factorization, where $U = Q$ and $V = PV_1V_2$. □

Note that in some literature the right side of the URV decomposition is written as

$$\begin{array}{c} \\ r \\ m-r \end{array} \begin{array}{c} \begin{array}{cc} r & n-r \end{array} \\ \begin{bmatrix} R & 0 \\ 0 & 0 \end{bmatrix} \end{array},$$

but this is clearly a matter of a simple cyclic column permutation which can be "absorbed" by the matrix V. With this proposition in our arsenal we are ready to prove Theorem 1.3.

Proof of Theorem 1.3. We begin by forming the block-matrix $[A^T \ B^T]^T$ and by doing the URV decomposition of B obtaining

$$\begin{bmatrix} A \\ B \end{bmatrix} = \begin{bmatrix} AQ_1Q_1^T \\ V_1RQ_1^T \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0 & V_1 \end{bmatrix} \begin{bmatrix} AQ_1 \\ R_1 \end{bmatrix} Q_1^T, \quad \begin{bmatrix} AQ_1 \\ R_1 \end{bmatrix} = \begin{array}{c} \\ m \\ l \\ p-l \end{array} \begin{array}{c} \begin{array}{cc} n-l & l \end{array} \\ \begin{bmatrix} A_{11}^{(1)} & A_{12}^{(1)} \\ 0 & B_{12}^{(1)} \\ 0 & 0 \end{bmatrix} \end{array},$$

where $B_{12}^{(1)}$ is nonsingular upper triangular and $l = \operatorname{rank}(B)$. Now we calculate the URV decomposition of $A_{11}^{(1)} = U_2R_2Q_2^T$ and get

$$\begin{bmatrix} A_{11}^{(1)} & A_{12}^{(1)} \\ 0 & B_{12}^{(1)} \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} U_2 & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} R_2 & U_2^TA_{12}^{(1)} \\ 0 & B_{12}^{(1)} \\ 0 & 0 \end{bmatrix} \begin{bmatrix} Q_2^T & 0 \\ 0 & I \end{bmatrix},$$

$$\begin{bmatrix} R_2 & U_2^TA_{12}^{(1)} \\ 0 & B_{12}^{(1)} \\ 0 & 0 \end{bmatrix} = \begin{array}{c} \\ k \\ m-k \\ l \\ p-l \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{bmatrix} 0 & A_{12}^{(2)} & A_{13}^{(2)} \\ 0 & 0 & A_{23}^{(2)} \\ 0 & 0 & B_{13}^{(2)} \\ 0 & 0 & 0 \end{bmatrix} \end{array},$$

with $A_{12}^{(2)}$ and $B_{13}^{(2)}$ nonsingular upper triangular. From that we can see that the last $k+l$ rows of the matrix are linearly independent and since orthogonal transformations preserve the rank of the matrix it follows that $r = \operatorname{rank}([A^T \ B^T]^T) = k + l$. The only step left to do is the QR decomposition $A_{23}^{(2)} = U_3A_{23}^{(3)}$ which gives us the required decomposition if we set

$$U = U_2 \begin{bmatrix} I & 0 \\ 0 & U_3 \end{bmatrix}, \quad V = V_1, \quad Q = Q_1 \begin{bmatrix} Q_2 & 0 \\ 0 & I \end{bmatrix}.$$

□

Chapter 2

Computing the Preprocessing Step on the GPU

After seeing how to compute the preprocessing step for the GSVD in the previous chapter we focus on the details of implementing this algorithm on the GPU. The algorithm consists of the QR and URV decompositions. The URV decomposition can be broken up into QR with column pivoting and a QR-like decomposition with orthogonal transformations applied from the right, giving the decomposition $A = RQ^T$, where R and Q are as in the QR decomposition. We will call this decomposition the RQ decomposition. In the following sections we analyze the computation of each of these steps on a CUDA-enabled GPU in detail.

2.1 The CUDA Programming Model

Before explaining the inner workings of our algorithm we briefly summarize the CUDA programming model to justify the decisions made when designing the algorithm. The system modeled by CUDA is a heterogeneous system comprised of a general purpose processor (called the *host*) and a specialized graphics processor (called the *device*). The program execution is controlled by the host, which groups the computation into a sequence of tasks and instructs the device to compute them.

The device executes multiple threads (sequences of instructions) in parallel. These threads are grouped into *blocks*, which is the largest group of threads that can be synchronized. The entire task is computed by a group of blocks called a *grid*. Since threads belonging to different blocks cannot be synchronized the task should consist of multiple independent subtasks so each of them can be executed by a single block. There is another group of threads in the thread hierarchy called the *warp*, which can be safely ignored from the point of program correctness, but is very important from the perspective of ef-

ficiency. The warp is a group of 32 threads belonging to the same block which always execute the same instruction, but possibly on different data — effectively forming a SIMD multiprocessor. These threads are always synchronized, so there is no need for implicit synchronization, but there is a problem with branching instructions, which can lead to the so called "thread divergence" if after this kind of instruction some of the threads end up on one and the others on another point in the program. This problem is solved by serialization — the first group of threads execute their sequence of instructions while the other group waits, and then the roles are swapped so the other group executes their sequence. The groups are diverged until they again end up on the same instruction in which case they are merged and continue to execute together. Of course, this behavior should be avoided since it results in slowdowns of up to a factor of 32 if every thread in a warp ends up executing a different instruction.

The device and the host have separate memories, so the data has to be copied to the device before the computation. This can be done simultaneously with the computation but since the order of the size of our data is $O(n^2)$ and the order of the required computation $O(n^3)$ the improvements on larger matrices are negligible. Therefore, we first copy all the data to the device, run our algorithm, and finally copy the results back to the host. The device memory hierarchy closely follows the thread hierarchy. It consists of the global memory which is accessible to all threads and persists between different tasks, shared memory which is shared among the threads of a single block, and finally, the local memory and a small amount of registers which are local to a single thread. The registers are the fastest, followed by the shared memory and finally the local and global memory, so the registers and the shared memory should be used in favor of the global and local memory whenever possible. Access to the global memory is cached in blocks of 128 bytes when using the L1 and L2 caches or 32 bytes when using only the L2 cache. The best performance can therefore be achieved if all threads in a warp are accessing memory that is a part of the same 128 byte wide block. There are also more specialized memories such as constant memory which can only be changed by the host and achieves the best performance when all blocks of a warp access the same memory location.

Different GPU architectures have some additional limitations which we should be aware of when developing our algorithm. The GPU that will be used to test our algorithm is a Tesla S2050, of compute capability 2.0. On this device the size of the block is limited to 1024 threads with a maximum of 1536 threads per streaming multiprocessor (SM), which is the basic building block of a GPU. Multiple blocks can be allocated to the SM at the same time, but not partial blocks. This means that by using a block size of 1024 we waste 512 threads. Since the performance of the SM increases by using as many threads as possible, blocks of this size are usually not the best strategy. There is also a limit on the number of blocks per SM, which is 8, so the minimal number of threads per block should be at least 192 in order to use all the available threads. The size of the warp is also a limiting

factor as block size should be a multiple of warp size in order to split it into multiple full warps. Otherwise, some of the warps have less than 32 threads, which degrades performance. From this analysis we can deduce that the candidate block sizes are 256, 512, 192, 384 and 768 threads.

2.2 Computing the QR Decomposition

In this section we describe a way to efficiently compute the required QR decompositions. A simple algorithm would be to mimic the sequential algorithm which computes the first column of the R factor, updates the trailing matrix and then proceeds to the next column, until the entire matrix is transformed to the R factor and the transformations are accumulated to form the Q factor. The problem with this approach is that it uses the device memory in an inefficient way. One step of the update requires the entire matrix to be read from the global memory and written back. This then accumulates to roughly $2mn^2/3$ memory reads and writes for the QR decomposition of an $m \times n$ matrix, which is the same as the number of computational operations required for the decomposition, so we expect this algorithm to be memory-bounded.

A better algorithm can be constructed by calculating the R factor of a block of columns before proceeding with the update of the trailing matrix. This method was used in [2] and significant speedups over other QR implementations were observed for tall and skinny matrices. In the rest of this section we describe a variant of this algorithm using Givens rotations.

Let b be the desired width of the block column and $A \in \mathbb{R}^{m \times n}$ be the input matrix,

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

where A_1 is the first block column, i.e. A_1 has b columns. The overall idea is to compute the decomposition $A_1 = Q_1 R_1$ where $R_1 = [R_{11}^T \; 0]^T$ and R_{11} is $b \times b$ upper triangular obtaining

$$A = Q_1 \begin{bmatrix} R_1 & Q_1^T A_2 \end{bmatrix} = Q_1 \begin{bmatrix} R_{11} & R_{12} \\ 0 & \tilde{A} \end{bmatrix}$$

and then compute the QR decomposition of $\tilde{A}$ in the same manner.

To do this, we need a way to efficiently compute the QR decomposition of A_1 on the GPU, i.e. we need to split the computation to a series of independent subtasks that can be computed by a single thread block. We do this by partitioning the matrix into $b \times b$ blocks and computing the QR decomposition of each of the blocks:

$$A_1 = \begin{bmatrix} A_{11} \\ \vdots \\ A_{k1} \end{bmatrix} = \begin{bmatrix} Q_{11} R_{11}^{(1)} \\ \vdots \\ Q_{k1} R_{k1}^{(1)} \end{bmatrix} = \begin{bmatrix} Q_{11} & & 0 \\ & \ddots & \\ 0 & & Q_{k1} \end{bmatrix} \begin{bmatrix} R_{11}^{(1)} \\ \vdots \\ R_{k1}^{(1)} \end{bmatrix},$$

where $k = \lceil m/b \rceil$. It is obvious that theses are independent tasks, so each of them can be computed by a single thread block. The next step is to group the blocks in pairs and use orthogonal transformations to annihilate one of them using the other. For simplicity suppose k is even, if it is odd, we ignore the last block. We compute the QR decomposition of each of the pairs $[(R^{(1)}_{(2i-1),1})^T \ (R^{(1)}_{2i,1})^T]^T$, $i = 1, \ldots, k/2$, which annihilates the even blocks of the matrix:

$$\begin{bmatrix} R^{(1)}_{11} \\ R^{(1)}_{21} \\ \vdots \\ R^{(1)}_{(k-1),1} \\ R^{(1)}_{k1} \end{bmatrix} = \begin{bmatrix} Q_{11} \begin{bmatrix} R^{(2)}_{11} \\ 0 \end{bmatrix} \\ \vdots \\ Q_{(k/2),1} \begin{bmatrix} R^{(2)}_{(k-1),1} \\ 0 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} Q_{11} & & 0 \\ & \ddots & \\ 0 & & Q_{(k/2),1} \end{bmatrix} \begin{bmatrix} R^{(2)}_{11} \\ 0 \\ \vdots \\ R^{(2)}_{(k-1),1} \\ 0 \end{bmatrix}.$$

We repeat this process on pairs of non-zero blocks until only the first block remains. Again, the QR decompositions of different block pairs are independent operations, so each of them can be computed by a single thread block. By performing the initial QR decomposition on the blocks we have ensured that the matrix pairs used in subsequent QR decompositions are of special form (two triangular matrices stacked on top of each other), which can be used to reduce the number of steps required for the decomposition of a single block pair. Additionally, the trailing matrix update can be computed in parallel with the next step of the block pair reduction, obtaining slightly better GPU utilization in the final stages of the algorithm, when the number of non-zero blocks is small. The overall procedure is described in more detail in Algorithm 2.1 and the QR decomposition of a single block in Algorithm 2.2. The process is also visualized on Figure 2.1.

The benefit of this approach over the naive parallelization is its memory use. The savings come from the fact that each thread block can first read all of its inputs to shared memory, perform the computation using only shared memory and finally write the results back to the global memory. For block size b this method requires roughly n/b steps, since each step computes b columns of the final R factor. One step consists of the initial QR decomposition and the trailing matrix update, which requires mn global memory reads and writes, and $\log_2 k$ steps to get a single triangular block, which requires additional $2mn$ reads and writes (as every step uses half of the matrix used in previous step). The entire step needs $3mn$ global memory reads and writes, which sums up to mn^2/b reads and writes for the whole computation if we take into account that each step operates on a slightly smaller matrix. This reduces the global memory accesses by a factor of $2b/3$.

The only thing remaining to show is how to compute the QR decomposition of a single block and of two triangular blocks. To compute the QR decomposition of a $b \times b$ matrix with a single thread block we split up its threads into $b/2$ groups of $2b$ threads. Each of the groups is instructed to perform one Givens rotation on a pair of matrix rows and to update the Q factor (half of the group updates the R factor and the other half the Q factor).

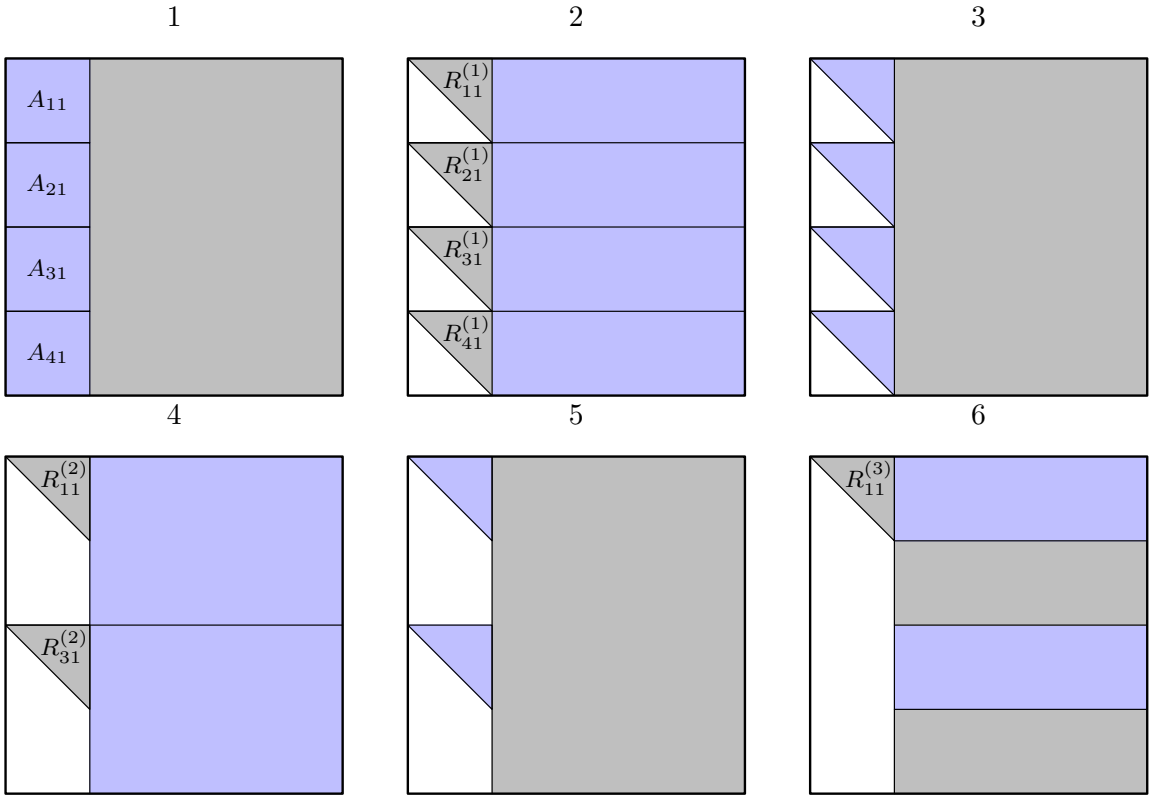

Figure 2.1: QR factorization of a single column block. First, the QR factorization of individual blocks is computed (1) and the trailing matrix is updated (2). Next, pairs of triangular blocks are reduced to a single triangular block (3) and the trailing matrix is updated again (4). The process continues until a single triangular block remains (5, 6).

The order of rotations is chosen to minimize the number of required parallel steps using the procedure introduced in [10]. This procedure uses a greedy algorithm to determine the maximal number of pairs of rows available for computation in each step. The algorithm chooses the bottom half of available rows and annihilates the first non-zero element in that row using the upper half of available rows. This makes sense since more elements of the bottom rows will have to be annihilated in subsequent steps. The algorithm is better illustrated by an example — the matrix displayed below shows the order in which the elements should be annihilated in case of an 8×8 block. The entries in the matrix designate the parallel step in which the annihilation of the corresponding block element should be

Algorithm 2.1: Blocked QR decomposition — host part

`Blocked_QR(`*inout :: A, out :: Q, in :: b*`)`;
Description: Computes the QR decomposition of the matrix $A \in \mathbb{R}^{m \times n}$ using a block size of b. On exit, A is overwritten with the R factor of the decomposition.
begin
  $Q \leftarrow I$;
  $A^{(1)} \leftarrow A, \quad Q^{(1)} \leftarrow Q$;
  for $i \leftarrow 1$ **to** $\lceil n/b \rceil$ **do**
    `Blocked_QR_step(`$A^{(i)}$, $Q^{(i)}$, b`)`;
    partition $A^{(i)} \rightarrow \begin{bmatrix} R_{11} & R_{12} \\ 0 & A^{(i+1)} \end{bmatrix}$, where R_{11} is $b \times b$;
    partition $Q^{(i)} \rightarrow \begin{bmatrix} Q_1 & Q^{(i+1)} \end{bmatrix}$, where Q_1 has b columns ;
  end
end

computed.

$$\begin{bmatrix} * & * & * & * & * & * & * & * \\ 3 & * & * & * & * & * & * & * \\ 2 & 5 & * & * & * & * & * & * \\ 2 & 4 & 7 & * & * & * & * & * \\ 1 & 3 & 6 & 8 & * & * & * & * \\ 1 & 3 & 5 & 7 & 9 & * & * & * \\ 1 & 2 & 4 & 6 & 8 & 10 & * & * \\ 1 & 2 & 3 & 5 & 7 & 9 & 11 & * \end{bmatrix}$$

This Givens ordering can be precomputed once on the host, stored in the constant memory and then read by the device when needed.

Once we obtain a triangular structure of $b \times b$ submatrices we can reduce pairs of those blocks to one triangular block with b groups of $2b$ threads to "peel off" the diagonals of the second matrix using the first matrix. This can be done in b steps as shown in the following

Algorithm 2.2: Single step of blocked QR decomposition — host part

`Blocked_QR_step`(*inout::A, inout::Q, in::b*) ;
begin
  partition $A^{(i)} \rightarrow \begin{bmatrix} A_{11} & A_{12} \\ \vdots & \vdots \\ A_{k1} & A_{k2} \end{bmatrix}$, where A_{i1} are $b \times b$;
  partition $Q^{(i)} \rightarrow \begin{bmatrix} Q_1 & \cdots & Q_k \end{bmatrix}$, where Q_i have b columns;
  device do `Device_QR`$(A_{i1}, \tilde{Q}_i)$, $\quad i = 1, \ldots, k$;
  device do $Q_i \leftarrow Q_i \tilde{Q}_i^T$, $A_{i2} \leftarrow \tilde{Q}_i A_{i2}$, $\quad i = 1, \ldots, k$;
  $l \leftarrow 1$;
  while $l < k$ **do**
    device do `Device_QR_triangular`$([A_{i,1}^T\ A_{i+l,1}^T]^T, \tilde{Q}_i)$,
      $i = 1, 1 + 2l, 1 + 4l, \ldots$;
    device do $\begin{bmatrix} A_{i,2} \\ A_{i+l,2} \end{bmatrix} \leftarrow \tilde{Q}_i \begin{bmatrix} A_{i,2} \\ A_{i+l,2} \end{bmatrix}$, $\begin{bmatrix} Q_{i,2} & Q_{i+l,2} \end{bmatrix} \leftarrow \begin{bmatrix} Q_{i,2} & Q_{i+l,2} \end{bmatrix} \tilde{Q}_i^T$
      $i = 1, 1 + 2l, 1 + 4l, \ldots$;
    $l \leftarrow 2l$;
  end
end

8×8 matrix.

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ & & 1 & 2 & 3 & 4 & 5 & 6 \\ & & & 1 & 2 & 3 & 4 & 5 \\ & & & & 1 & 2 & 3 & 4 \\ & & & & & 1 & 2 & 3 \\ & & & & & & 1 & 2 \\ & & & & & & & 1 \end{bmatrix}$$

The method for computing the QR decomposition of a single $b \times b$ block is outlined in Algorithm 2.3. The method for the QR decomposition of a pair of blocks is analogous, it only uses a different Givens ordering table and a double amount of groups of threads.

Algorithm 2.3: QR decomposition of a single block — device part

`Device_QR`(*inout :: A, out :: Q*);

Description: Computes the QR decomposition of the matrix $A \in \mathbb{R}^{b \times b}$. On exit, A is overwritten with the R factor of the decomposition.

Global : A table T with $steps$ rows and $b/2$ columns where each entry (i, j) consists of a triplet (r_1, r_2, c) which designates that in the i-th step thread group j should apply Givens rotation to rows r_1 and r_2 to annihilate the element at position (r_2, c). In case the group should wait this particular step and do nothing, every element of the triplet is set to -1.

```
begin
    Q ← I ;
    parallel idx ← 1 to b^2 do
        gid ← GroupId(idx);
        for i ← 1 to steps do
            r1, r2, c ← T_{i,gid} ;
            if r1 ≠ −1 then
                Apply rotation to rows r1 and r2 of A and Q to annihilate A_{r2c} ;
            end
            synchronize threads;
        end
    end
end
```

2.3 Computing the Complete Orthogonal Decomposition

To compute the complete orthogonal decomposition we first need an efficient algorithm for the QR decomposition with column pivoting. The classical algorithm performs column pivoting by permuting the column with the largest norm to the beginning of the matrix and then performing one step of the normal QR factorization. This gives a factorization where the absolute value of each diagonal element r_{ii} of matrix R is larger than each of the column norms of the bottom-right block of R starting at position $(i + 1, i + 1)$. The rank is determined by stopping the algorithm once the diagonal element r_{ii} drops under some small threshold as this means that the norm of the trailing matrix is small enough to be discarded and the rank is declared to be $i - 1$. However, this algorithm suffers from bad memory use on the GPU for the same reasons as the classical algorithm for the QR decomposition. The reason for this is the inability to compute the QR decomposition of a column block because the second column cannot be selected before the computation with

the first column is completed and the trailing matrix updated. To alleviate this problem new algorithms have been developed recently, which forfeit the decrease of absolute values of diagonal matrix entries in exchange for the ability to select multiple linearly independent columns at once, while retaining (and some of them even boosting) the ability to detect the rank of the matrix. Some of the strategies used by these algorithms include randomization [9] and tournament selection [6]. We will use the ideas from [6] to develop a variation of the algorithm suitable for the GPU.

Say we want to find $\min\{r, b\}$ linearly independent columns of a matrix $A \in \mathbb{R}^{m\times n}$, where r is the rank of A and b is the block size used in Algorithm 2.1. First define $A^{(0)} = A$. We divide the matrix $A^{(0)}$ into $m \times 2b$ blocks, $A^{(0)} = [A_1 \ \dots \ A_k]$, and compute the QR with column pivoting of each of the tall and skinny matrices $A_i = Q_i R_i P_i^T$. We apply the permutation P_i to the matrix A_i and select the first b blocks of the resulting matrix. This is done to extract the maximal amount of independent columns from the matrix A, as these columns tend to end up in the beginning of the matrix. We do this for each of the matrices A_i and from them create a new matrix $A^{(1)}$, which has roughly $n/2$ columns. We repeat this process, selecting the columns of matrices $A^{(i)}$ and obtaining matrices $A^{(i+1)}$, until we end up with a matrix that has b columns. We use these b columns for the next step of the Algorithm 2.1. This procedure is described by Algorithm 2.4 and illustrated in Figure 2.2. In the absence of round off errors it will always find the largest possible set of independent columns, as shown by Theorem 2.1.

Theorem 2.1. *If $A \in \mathbb{R}^{m\times n}$ has rank r, then the last of the matrices $A^{(i)}$ has rank* $\min\{r, b\}$.

Proof. We prove this theorem by induction on the number of required matrices $A^{(i)}$.

If $n \leq b$ and no steps are required it is obvious that the matrix $A^{(0)} = A$ has $\min\{r, b\}$ independent columns.

Suppose that the theorem holds for all matrices which require at most i steps of Algorithm 2.4 to compute the final matrix $A^{(i)}$. Let A be a matrix such that $i + 1$ steps are required to compute the final matrix $A^{(i+1)}$. $A^{(i+1)}$ is obtained by applying one step of the algorithm on two matrices $A_1^{(i)}$ and $A_2^{(i)}$ ($A^{(i)} = [A_1^{(i)} \ A_2^{(i)}]$) with at most b columns which are obtained from the two "halves" A_1 and A_2 of the original matrix A using at most i steps. From the induction hypothesis it follows that matrices $A_1^{(i)}$ and $A_2^{(i)}$ have ranks $\min\{r_1, b\}$ and $\min\{r_2, b\}$, respectively, where $r_1 = \operatorname{rank}(A_1)$ and $r_2 = \operatorname{rank}(A_2)$, $r_1 + r_2 \geq r$. If either one of r_1 and r_2 is greater or equal to b it is obvious that $r \geq b$ and, since the matrix $A^{(i+1)}$ is constructed by selecting b independent columns from $[A_1^{(i)} \ A_2^{(i)}]$, it has rank $b = \min\{r, b\}$. If both of the ranks r_1 and r_2 are smaller than b, then matrices $A_1^{(i)}$ and $A_2^{(i)}$ contain the maximal amount of linearly independent columns of A_1 and A_2, respectively. This means that the subspace spanned by $A = [A_1 \ A_2]$, is the same subspace as the one spanned by the matrix $[A_1^{(i)} \ A_2^{(i)}]$, so there must be exactly r linearly independent columns in the matrix $[A_1^{(i)} \ A_2^{(i)}]$, which are then sampled into $A^{(i+1)}$. □

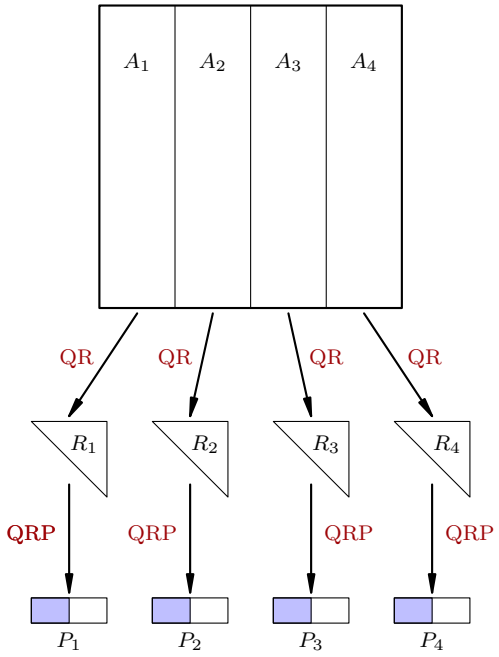

Figure 2.2: One step of the pivoting procedure. The matrix is split into block columns of width $2b$. The QR decomposition of each of the block columns is computed to obtain the triangular factors of each of the blocks. Finally the QR decomposition with column pivoting of each of the triangular factors is performed and first b elements of the resulting permutation are selected. These elements designate the $n/2$ columns which will be used to build the matrix $A^{(i+1)}$ for the next step of the pivoting procedure.

Now we only need to show how to find the QR decomposition with column pivoting of a tall and skinny matrix A_j. First, we compute the QR decomposition of A_j using Algorithm 2.2, which computes the QR decomposition of a tall and skinny matrix, obtaining a $2b \times 2b$ matrix R and then compute the QR with column pivoting of R using a single thread block. As we are only interested in the permutation P we do not need to compute the orthogonal factor Q, which saves threads and allows the efficient computation of a $2b \times 2b$ matrix block using $2b^2$ threads and at most $2b$ steps.

To compute the QR with column pivoting of a single block, in each step of the decomposition we use a method similar to Algorithm 2.3. We divide the threads into b groups of $2b$ threads. First, we find the column with the largest norm — each group of threads is instructed to find the norms of two columns via reduction. Once we have all of the column norms a single group of threads uses reduction on the norms to find the column with the largest norm. Lastly, the column with the largest norm is swapped with the first column.

Algorithm 2.4: Pivoting algorithm for the QR decomposition

`Block_Pivot(`*in :: A, in :: b, out :: P*`);`

Description: Finds at most b linearly independent columns of $A \in \mathbb{R}^{m\times n}$ and returns their indices as elements of the array $P \in \mathbb{R}^b$.

begin
- $A^{(0)} \leftarrow A$;
- $i \leftarrow 0$;
- **while** $A^{(i)}$ *has more than b columns* **do**
  - **partition** $A^{(i)} \rightarrow \begin{bmatrix} A_1 \cdots A_k \end{bmatrix}$, where A_j have $2b$ columns;
  - Use Algorithm 2.2 to obtain the R factor from the QR decomposition of A_j for all $j = 1, \ldots k$ (label the R factors with R_j) ;
  - **device do** compute the QR decompositions with column pivoting of the $2b \times 2b$ matrices R_j and obtain the permutations P_j;
  - Apply P_j to the columns of each of the matrices A_j and obtain A'_j;
  - Take the first b columns of each of the matrices A'_j and from them form $A^{(i+1)}$;
  - $i \leftarrow i + 1$;
- **end**
- Set P to the list of indices of $A^{(i)}$'s columns in the original matrix A;

end

This completes the pivoting of the first column and we are free to use Givens rotations in the same manner as in Algorithm 2.3 and annihilate the subdiagonal elements of the first column. Note that we cannot start to annihilate other columns, as we still do not know which column will be the next pivot. This explains a single step of the QR decomposition with column pivoting. The rest of the steps are done analogously. Again, we are only interested in the permutation matrix P, so we do not need to compute the Q factor.

After finding b independent columns (or less in case the rank of the matrix is less than b) we permute the matrix A so these b columns become the first columns in the matrix, perform the QR decomposition of the first $m\times b$ block of A and update the trailing $m\times(n-b)$ part of A. We continue with the rest of the matrix in the same manner. As soon as we find less than b independent columns, we stop the computation and report the rank of the matrix to be the total number of linearly independent columns found during all of the steps of the algorithm.

After finding the QR decomposition with column pivoting:

$$A = Q \begin{bmatrix} R \\ 0 \end{bmatrix} P^T$$

where Q is orthogonal, $R \in \mathbb{R}^{r\times n}$ upper trapezoidal with full row rank and P is a permutation

matrix, we only need to compute the RQ decomposition of R in order to get the complete orthogonal decomposition. The procedure is analogous to Algorithm 2.1, but instead of operating on blocks of columns, we operate on blocks of rows and apply rotations to the columns to get the decomposition

$$R = \begin{bmatrix} 0 & R' \end{bmatrix} V^T,$$

where R' is upper triangular nonsingular and V orthogonal.

2.4 Putting It All Together

Now that we have all the building blocks for the preprocessing step we describe how to combine them to get the complete algorithm. In order to get the largest gain in memory usage, while retaining efficiency of a single block we need the matrix block size b to be as large as possible. Since we need the thread block size to be equal to $2b^2$ for the pivoting scheme, the only b which gives a candidate block size from Section 2.1 is $b = 16$ ($2b^2 = 512$). We also said that we use *groups* of threads of size $2b = 32$. The most natural choice is to use a single warp as a group, as this way we have the best possible constant memory usage, since the whole group will access the same value of the table T in Algorithm 2.3.

The algorithm proceeds as follows: let (A, B), $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{p \times n}$ be the matrix pair we need to preprocess. First, initialize U, V and Q to identity matrices. We compute the complete orthogonal decomposition of B and update the matrices A, V and Q in parallel with the computation of B, applying the individual $b \times b$ blocks on the block-rows of V and Q. This will increase the utilization levels of the GPU in the final steps of reductions, when there is only a small amount of computation left to be done. Compute the complete orthogonal decomposition of the first $m \times n - l$ block of A and update U, Q and the trailing block of A in parallel. Finally, compute the QR decomposition of the bottom-right $(m-k) \times l$ block of A and update U in parallel.

Chapter 3

Numerical Testing

The algorithm was tested on a machine with a 4-core *Intel Xeon* CPU and an *Nvidia Tesla S2050*. Even though the *Tesla S2050* system consists of 4 GPUs this algorithm uses only one of them, so a version of the algorithm that utilizes all 4 GPUs would achieve additional speedups. We test the algorithm's speed, rank detection, backward error and the effect on the computed generalized singular values. The results are compared with those of LAPACK's and MKL's `xGGSVP` routines.

3.1 Test Matrices

The input data for the algorithm consists of a matrix pair $(A, B) \in \mathbb{R}^{m\times n} \times \mathbb{R}^{p\times n}$. In all of the tests the number of rows of both matrices (m and p) are set to be equal. The algorithm was tested on inputs with different ratios of the number of rows (m) and the number of columns (n). The exact ratios were $1:1$, $2:1$, $4:1$ and $8:1$, and for each of the ratios ten matrices of different sizes were generated. The ranks of the matrices were also set to a fixed fraction of the number of columns. The rank of B (l from Theorem 1.3) was set to $0.5n$ and the rank of $[A^T\ B^T]^T$ ($k + l$ from Theorem 1.3) to $0.8n$.

The matrices were generated using *MATLAB*'s `randn` method to generate matrices $T_1 \in \mathbb{R}^{m\times k+l}$, $T_2 \in \mathbb{R}^{p\times l}$ and $T_3 \in \mathbb{R}^{n\times n}$. Then, matrices U and R_A were obtained as Q and R factors of the QR decomposition of T_1, V and R_B as Q and R factors of the QR decomposition of T_2 and Q as the Q factor from the QR decomposition of T_3. Matrices R_A and R_B were padded with zeros from the left to the width of n columns. Finally, the pair (A, B) was generated by setting $A = UR_AQ$ and $B = VR_BQ$.

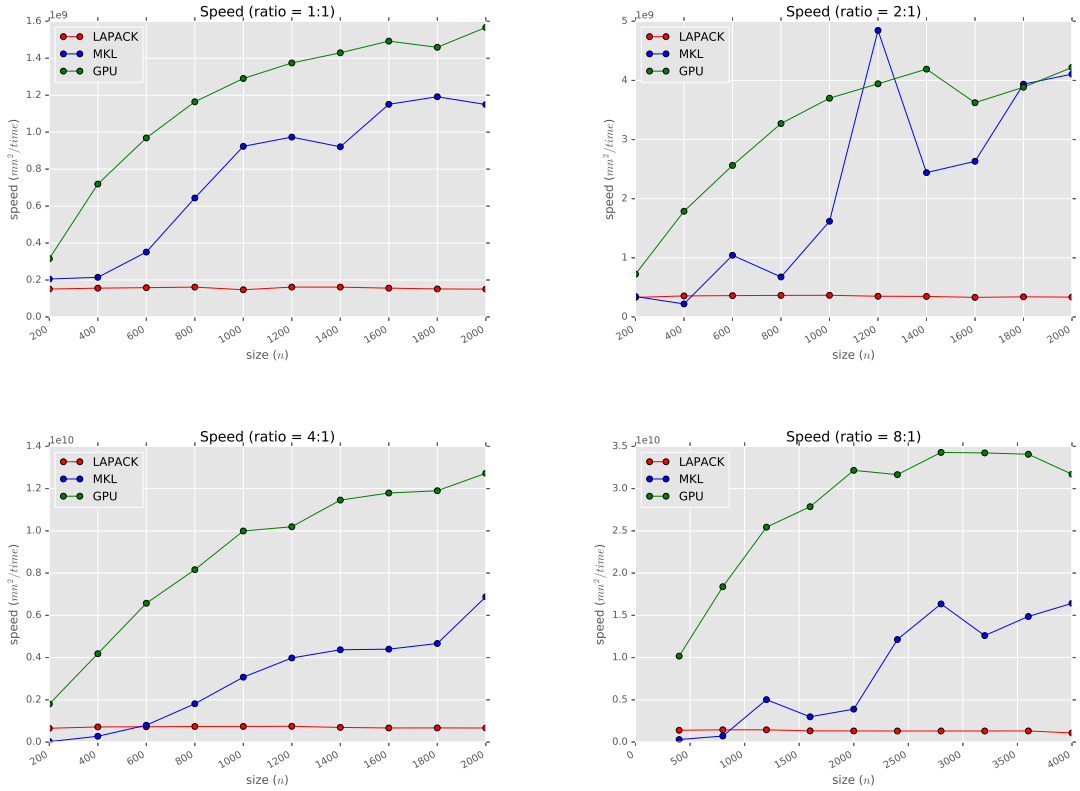

Figure 3.1: The speed of our algorithm (green) compared to the speed of MKL's (blue) and LAPACK's (red) implementations of DGGSVP

3.2 Speed

The speed was tested by running our algorithm and MKL and LAPACK versions of `xGGSVP` using the same matrix pair (A, B) as input. The results are summarized in Figure 3.1. The number of rows (m) of the input matrices is plotted on the x-axis and the speed $mn^2/time$ on the y-axis. The algorithm outperforms single-threaded LAPACK's implementation by roughly a factor of $10x$, with more speedup gained on the tall and skinny matrices. It is slightly faster than the multi-threaded MKL's implementation, up to a factor of about $2x$ on the tall and skinny inputs.

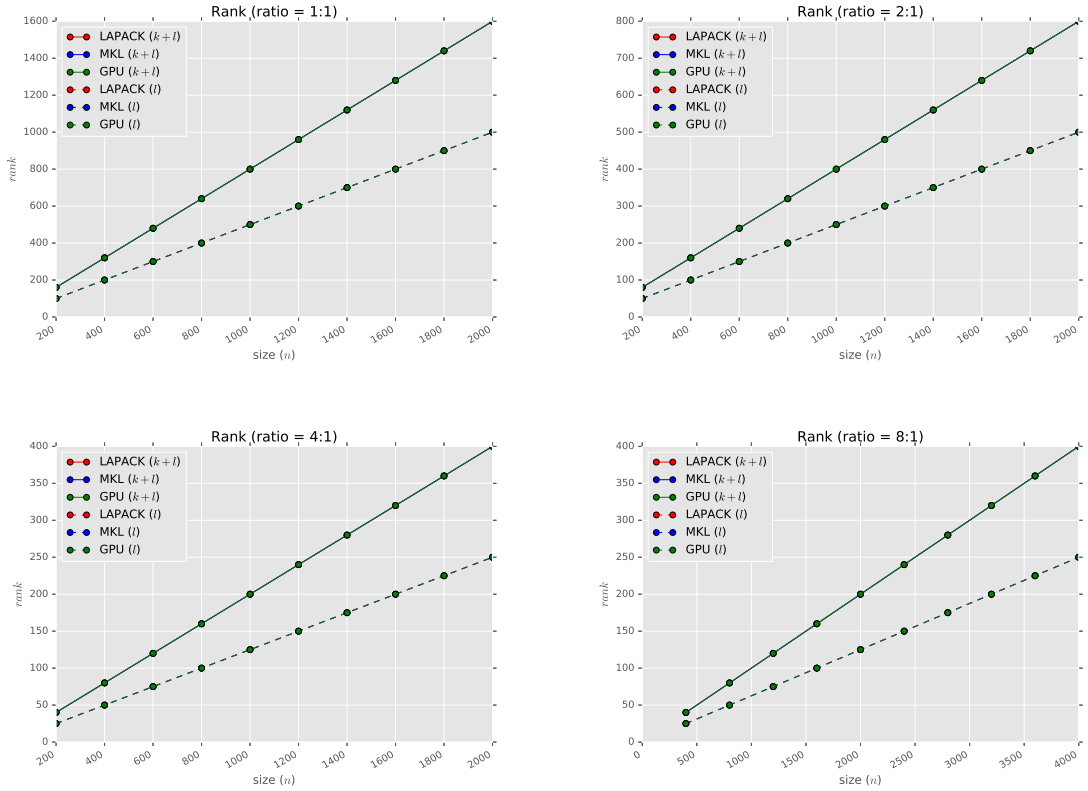

Figure 3.2: The ranks (k and $k + l$) of the matrices computed by our algorithm, LAPACK and MKL. The computed l is marked with the striped lines and $k+l$ with the full lines. Our algorithm is shown in green, MKL's version in blue and LAPACK's in red.

3.3 Rank Detection

For the GSVD algorithm for triangular matrix pair (either implicit Kogbetlianz or Hari–Zimmerman) to work it is crucial for the preprocessing step to accurately determine the rank of the matrices A and B and return a nonsingular B_{13}. For this reason the rank detection properties, i.e. computed k and l were also tested. Figure 3.2 compares the rank detection of our algorithm and LAPACK's and MKL's implementations. The x-axis again represents the matrix size and the y-axis the ranks of the matrices. The stripped line is the rank of the matrix B (parameter k) and the full line is the rank of the pair (A, B). We can see that all of the algorithms successfully compute the required ranks.

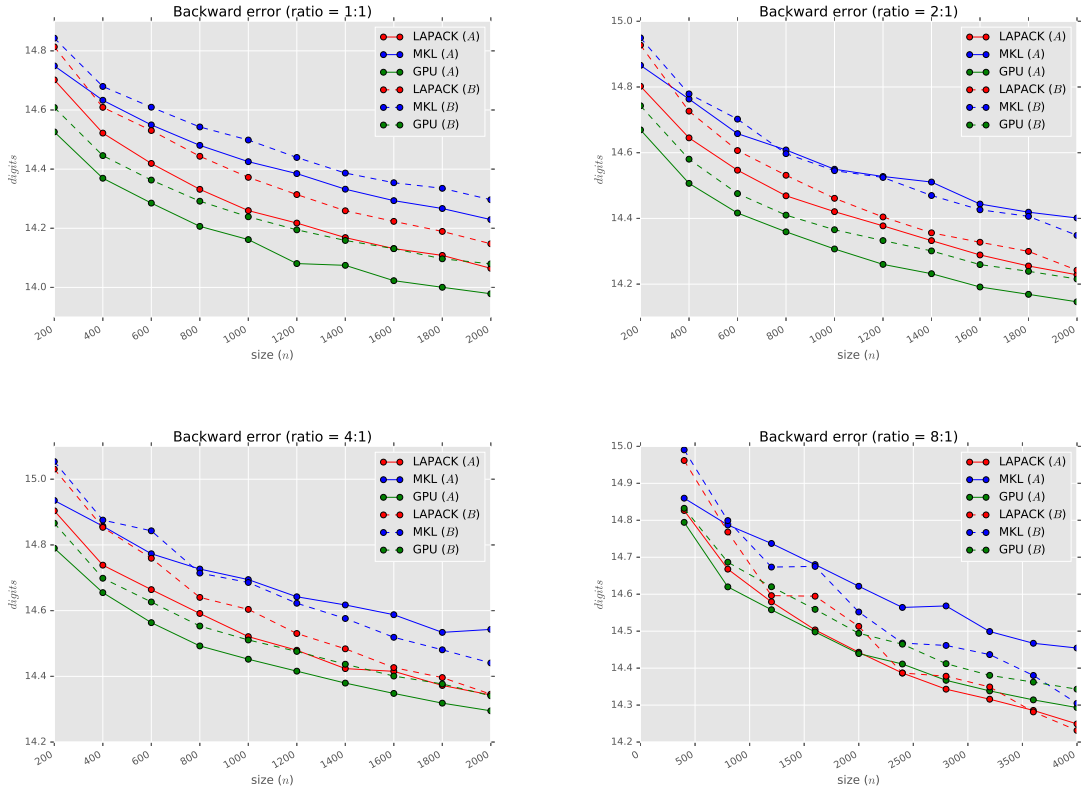

Figure 3.3: Backward errors of our algorithm (green), LAPACK (red) and MKL (blue). Full lines show the number of correct digits when "reconstructing" the matrix A, and striped lines the same for the matrix B.

3.4 Backward Error

To test the stability of our algorithm we look at the backward error. Suppose that the true preprocessing step decomposition is $A = UR_AQ^T$, $B = VR_BQ^T$, but due to round off errors we compute a slightly perturbed $\tilde{U}$, $\tilde{R}_A$, $\tilde{Q}$, $\tilde{V}$ and $\tilde{R}_B$, i.e. we get a decomposition

$$A + \delta A = \tilde{U}\tilde{R}_A\tilde{Q}^T, \quad B + \delta B = \tilde{V}\tilde{R}_B\tilde{Q}^T,$$

where δA and δB are small relative to A and B, respectively. To approximate the accuracy of our algorithm we compute the relative sizes of δA and δB:

$$\frac{||\delta A||_F}{||A||_F} = \frac{||A - \tilde{U}\tilde{R}_A\tilde{Q}^T||_F}{||A||_F}, \quad \frac{||\delta B||_F}{||B||_F} = \frac{||B - \tilde{V}\tilde{R}_B\tilde{Q}^T||_F}{||A||_F}.$$

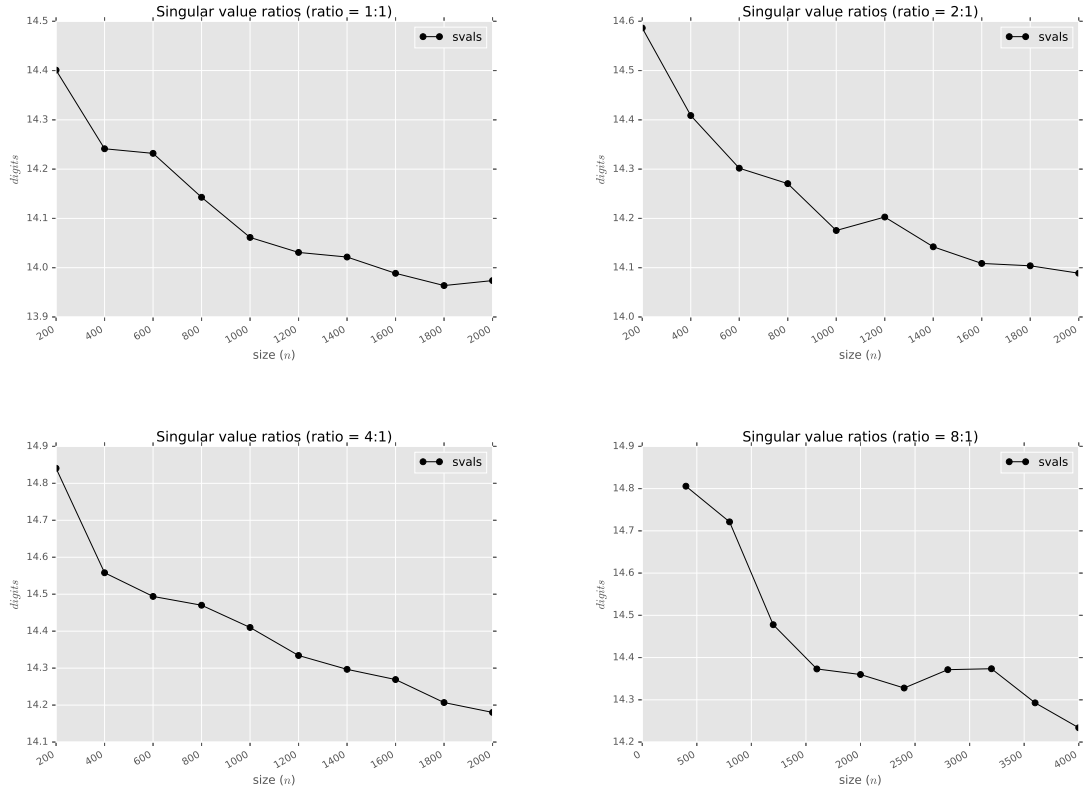

Figure 3.4: Difference of the generalized singular values computed by LAPACK and by our algorithm

Figure 3.3 gives the comparison of our algorithm, LAPACK and MKL. The x-axis is again the number of rows of the matrix and the y-axis is $-\log_{10}(\|\delta M\|_F/\|M\|_F)$,where $M = A$ (full line) or $M = B$ (striped line). Since the $\|\cdot\|_F$ norm gives the average element value in the matrix (in the context of quadratic mean), the expression above gives the number of correct digits of the average matrix element. Our algorithm is slightly worse than LAPACK and MKL, losing about 0.2 of a digit on square matrices and reaching LAPACK's precision on extremely tall and skinny matrices.

3.5 Effect on the Generalized Singular Values

The last accuracy measure studied is the effect of our algorithm on the generalized singular values, compared to the values obtained by LAPACK. The `xGGSVD` routine which computes the GSVD consists of two steps. First, the `xGGSVP` routine is called, which reduces the

matrix pair to upper triangular form, followed by a call to `xTGSJA`, which computes the GSVD of the triangular matrix pair with nonsingular B using the implicit Kogbetliantz approach. To test the effect of our algorithm we replace the `xGGSVP` call with our algorithm and call `xTGSJA` on the result. We then compare the obtained generalized singular values $\sigma_i^{(gpu)} = \alpha_i^{(gpu)}/\beta_i^{(gpu)}$ with the corresponding $\sigma_i^{(lap)} = \alpha_i^{(lap)}/\beta_i^{(lap)}$ obtained from the call to `xGGSVD` with the same input matrices. We only compare the l nontrivial (nonzero and finite) values. If the ranks of the matrix B computed by `xGGSVD` and our algorithm differ we use the smaller rank as the number of nontrivial generalized singular values. This rarely happens, as shown in Section 3.3, but if it does the largest singular values are likely close to infinity as they are obtained by dividing α_i with a very small β_i. For each of the generalized singular values we compute

$$-\log_{10}\left(\frac{1}{l}\sum_{i=1}^{l}\frac{|\sigma_i^{(gpu)}-\sigma_i^{(lap)}|}{\max\{\sigma_i^{(gpu)},\sigma_i^{(lap)}\}}\right),$$

which has roughly the same meaning as the expression from Section 3.4. On the x-axis of graphs in Figure 3.4 is again the number of rows of the matrices and on the y-axis the value of the expression written above. The difference is, as expected, of the same order of magnitude as the backward error.

Acknowledgment

I would like to thank my advisor Sanja Singer for proposing the subject of this thesis, all the invaluable comments, creation of some of the figures and guidance while writing this thesis. I would also like to thank my parents and my entire family for their financial and moral support for the entire duration of my studies. Last, but not the least, I would like to thank Jelena for her love and support, for always being there in times of need. Thank you all, without you this wouldn't be possible.

Bibliography

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, LAPACK Users' guide, vol. 9, SIAM, 1999.

[2] M. Anderson, G. Ballard, J. Demmel, K. Keutzer, Communication-avoiding QR decomposition for GPUs, in: Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International, IEEE, 2011, pp. 48–58.

[3] Z. Bai, The CSD, GSVD, their applications and computations (1992).

[4] Z. Bai, J. W. Demmel, Computing the generalized singular value decomposition, SIAM Journal on Scientific Computing 14 (6) (1993) 1464–1486.

[5] Z. Bai, H. Zha, A new preprocessing algorithm for the computation of the generalized singular value decomposition, SIAM Journal on Scientific Computing 14 (4) (1993) 1007–1012.

[6] J. W. Demmel, L. Grigori, M. Gu, H. Xiang, Communication avoiding rank revealing QR factorization with column pivoting, SIAM Journal on Matrix Analysis and Applications 36 (1) (2015) 55–89.

[7] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, A. R. LeBlanc, Design of ion-implanted MOSFET's with very small physical dimensions, IEEE Journal of Solid-State Circuits 9 (5) (1974) 256–268.

[8] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, D. Burger, Dark silicon and the end of multicore scaling, in: Computer Architecture (ISCA), 2011 38th Annual International Symposium on, IEEE, 2011, pp. 365–376.

[9] P. G. Martinsson, G. Quintana-Orti, N. Heavner, R. van de Geijn, Householder QR factorization: Adding randomization for column pivoting. FLAME working note# 78, arXiv preprint arXiv:1512.02671.

[10] J. J. Modi, M. R. B. Clarke, An alternative Givens ordering, Numerische Mathematik 43 (1) (1984) 83–90.

[11] V. Novaković, S. Singer, S. Singer, Blocking and parallelization of the Hari–Zimmermann variant of the Falk–Langemeyer algorithm for the generalized SVD, Parallel Computing 49 (2015) 136–152.

[12] C. C. Paige, Computing the generalized singular value decomposition, SIAM Journal on Scientific and Statistical Computing 7 (4) (1986) 1126–1146.

Sažetak

U ovom radu proučavamo algoritam pretprocesiranja za generalizirani SVD i njegovu paralelizaciju na grafičkoj kartici. Algoritam pretprocesiranja reducira par proizvoljnih matrica (A, B) na gornje trokutasti par $(\tilde{A}, \tilde{B})$ pri čemu je $\tilde{B}$ regularna matrica. Nakon algoritma pretprocesiranja za računanje generaliziranog SVD-a mogu se iskoristiti specijalizirani algoritmi, kao što je implicitni Kogbrliantzov ili implicitni Hari–Zimmermanov algoritam, koji zahtjevaju ovakvu strukturu para matrica.

U prvom poglavlju rada definiramo generalzirani SVD i opisujemo gornje trokutastu faktorizaciju koja je rezultat pretprocesiranja. Dodatno, opisujemo potpunu ortogonalnu faktorizaciju korištenu u postupku pretprocesiranja. U drugom poglavlju opisuju se izazovi koji se pojavljuju pri razvoju algoritama za grafičke kartice te se daje efikasna implementacija algoritma pretprocesiranja. U zadnjem poglavlju testiramo razvijeni algoritam na matricama različitih veličina i uspoređujemo brzinu, detekciju ranga, povratnu grešku i utjecaj na generalizirane singularne vrijednosti našeg algoritma te LAPACK-ove i MKL-ove verzije procedure $xGGSVP$.

Summary

In this thesis we study a preprocessing algorithm for the generalized SVD and its parallelization on the graphics processing unit. The preprocessing algorithm reduces a general matrix pair (A, B) to an upper triangular matrix pair $(\tilde{A}, \tilde{B})$ where B is nonsingular. After the preprocessing step a specialized algorithm, such as the implicit Kogbetliantz or the Hari–Zimmerman algorithm, which requires this structure of the matrix pair can be used to compute the generalized SVD.

In the first chapter of the thesis we define the generalized SVD and describe the upper triangular decomposition resulting from the preprocessing step. Additionally, we describe the complete orthogonal decomposition which is used in the preprocessing step. The second chapter describes the challenges that arise when developing algorithms for the GPU and presents an efficient implementation of the preprocessing step. In the final chapter we test the algorithm on a variety of matrices of different sizes and compare the speed, rank detection, backward error and the effect on generalized singular values of our algorithm and LAPACK's and MKL's versions of `xGGSVP`.

Životopis

Goran Flegar rođen je 4. listopada 1992. godine u Varaždinu. Živi u Sračincu gdje pohađa osnovnu školu. Nakon osnovne škole upisuje prirodoslovno-matematički smjer Prve Gimnazije Varaždin te se nakon maturiranja 2011. seli u Zagreb gdje upisuje sveučilišni preddiplomski studij matematike na Matematičkom odsjeku Prirodoslovno-matematičkog fakulteta. 2014 završava preddiplomski studij i upisuje diplomski studij računarstva i matematike na istom fakultetu.

Matematika i računarstvo ga interesiraju od osnovne škole, pa sudjeluje na državnim i nekoliko međunarodnih natjecanja iz informatike. Za vrijeme studija bio je demonstrator iz kolegija Programiranje, Uvod u paralelno računanje te Primjena paralelnih računala. Nakon preddiplomskog studija provodi mjesec dana u softverskoj tvrtki IGEA, gdje razvija mobilnu aplikaciju za održavanje Hrvatskih autocesta. Trenutno radi na projektu EkoRaMa, čiji je cilj razvoj novog kurikuluma za diplomski studij računarstva i matematike. Tjekom studiranja primio je nagrade za izniman uspjeh na preddiplomskom i diplomskom studiju.

Biography

Goran Flegar was born on 4th October 1992 in Varaždin. He lived in Sračinec where he attended primary school. He graduated from Prva Gimnazija Varaždin and in 2011 moved to Zagreb to study mathematics at the Faculty of Science, Department of Mathematics. In 2014 he finished the Undergraduate Programme of Mathematics and enrolled in the Graduate Programme of Computer Science and Mathematics at the same faculty.

He was interested in mathematics and computer science since primary school, and participated in a number of national and a few international competition in computer science. During his studies at the Department of Mathematics he was a student mentor for the courses Computer Programming, Introduction to Parallel Programming and Applications of Parallel Computers. After the undergraduate programme he spent a month in a software company IGEA as an intern, where he developing a mobile app for the maintenance of Croatian Motorways. Currently, he is working on the EkoRaMa project, whose goal is to create a new curriculum for the Graduate Programme in Computer Science and Mathematics. He received rewards for excellent achievement for both the undergraduate and the graduate study.