Implementing a Tiled Singular Value Decomposition: A Framework for Tiled Linear Algebra in Julia

by

Evelyne Ringoot

M.Sc. Civil Engineering, Vrije Universiteit Brussel, 2020
B.Sc. Engineering Sciences, Vrije Universiteit Brussel, 2018

Submitted to the Center for Computational Science and Engineering
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTATIONAL SCIENCE AND ENGINEERING

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2024

© 2024 Evelyne Ringoot. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Evelyne Ringoot
Center for Computational Science and Engineering
August 18, 2024

Certified by: Alan Edelman
Professor of Applied Mathematics, and
Computer Science and AI Laboratories, Thesis Supervisor

Accepted by: Nicolas Hadjiconstantinou
Professor of Mechanical Engineering
Co-Director, Center for Computational Science & Engineering

Implementing a Tiled Singular Value Decomposition:
A Framework for Tiled Linear Algebra in Julia

by

Evelyne Ringoot

Submitted to the Center for Computational Science and Engineering
on August 18, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTATIONAL SCIENCE AND ENGINEERING

ABSTRACT

High-performance computing (HPC) is essential for scientific research, enabling complex simulations and analyses across various fields. However, the specialized knowledge required to utilize HPC effectively can be a barrier for many scientists. This work introduces a hardware-agnostic, large-scale tiled linear algebra framework in Julia designed to enhance accessibility and usability without compromising performance. By providing a flexible abstraction layer, the framework simplifies the development and testing of new algorithms across diverse computing architectures. Julia language's multiple-dispatch and type inference facilitate the development of type-agnostic, hardware-agnostic, and multi-use frameworks by allowing composability. Utilizing a tiled approach, the implemented framework improves data locality, parallelism, and scalability, making it well-suited for modern heterogeneous environments. Its practical benefits are demonstrated through the implementation of tiled QR-based singular value decomposition (SVD), demonstrating how it streamlines the development process and accelerates scientific discovery. The developed framework is used to implement an in-GPU tiled SVD and an out-of-core GPU-accelerated SVD. Furthermore, its extensibility is demonstrated by implementing a tiled QR algorithm. This work aims to democratize HPC resources by bridging the gap between advanced computational capabilities and user accessibility, empowering a broader range of scientists to fully leverage modern computing technologies.

Thesis supervisor: Alan Edelman
Title: Professor of Applied Mathematics, and
Computer Science and AI Laboratories

Acknowledgments

I wish to express my gratitude to Dr. Valentin Churavy for mentoring me in this project, for teaching me the entirety of the field of High-Performance Computing at rocket speed, for his patience throughout my journey in HPC, for the humorous life and research purpose insights, and the many fun moments working together. Valentin, I am deeply grateful for your generosity in sharing your time and knowledge with me.

Next, I would like to thank Dr. Rabab Alomairy, for the motivating and fun numerical linear algebra research discussions. Rabab, you have been an inspiration and a source of motivation for me, your knowledge of and excitement for numerical linear algebra fueled my desire to continue my research journey.

I would also like to extend my gratitude to Dr. Tim Besard, for answering all my GPU-related questions, and Julian Samaroo, for helping me review the algorithm line by line to analyze performance. Without your help, I could not have achieved the results in this thesis.

I am grateful to my supervisor, Professor Alan Edelman, for his unwavering support, for the many opportunities for collaboration I receive in the Julialab, for the critical questions that pushed me to rethink my research vision, and for the research direction talks when I am lost. Your support and excitement about the singular value decomposition have been vital to this work.

Thank you, to my family and friends for their love and encouragement: I could not have achieved this work without your support.

Finally, I wish to acknowledge the Belgian American Foundation for financial support and the MIT SuperCloud and Lincoln Laboratory Supercomputing Center [1] for providing High-performance computing resources and technical support that have contributed to the research results reported within this thesis.

Contents

List of Figures 5

List of Code Fragments 6

List of Tables 6

Introduction **7**
- HPC accessibility . . . 7
- Generic frameworks . . . 8
- Research questions . . . 9
- Contributions of the work . . . 10
- Thesis organization . . . 10

1 The SVD and its computation on modern computing architecture **11**
- 1.1 The singular value decomposition and its numerical computation . . . 11
- 1.2 Tiled linear algebra algorithms for modern hardware . . . 13

2 Methods **17**
- 2.1 The out-of-core three-phase tiled SVD . . . 17
- 2.2 A general framework for tiled SVD . . . 21
- 2.3 The out-of-core tiled QR . . . 23
- 2.4 Performant generic QR kernels . . . 24
- 2.5 Bulgechasing and diagonalization . . . 25
- 2.6 Testing and benchmarking . . . 26

3 Results and discussion **27**
- 3.1 Performance benchmarking . . . 27
- 3.2 Data type-agnostic performance . . . 32
- 3.3 Optimization of kernel sizes . . . 32

4 Conclusions and perspectives **35**

A Code fragments **37**
- A.1 QR kernel code example . . . 37
- A.2 Partial bulge-chasing code . . . 39
- A.3 Partial GeneralTiledMatrix definition code . . . 40

References 42

List of Figures

1 Goal of the current work in the scope of high-performance computing. 7

1.1 Schematic representation of a singular value decomposition: the product of a unitary matrix, a diagonal matrix, and the transpose of a unitary matrix. 11
1.2 GPU architecture schematic. 14

2.1 Schematic overview of the three-step QR-based singular value decomposition algorithm. 18
2.2 Schematic overview of the bulgechasing step of the three-step SVD algorithm. . . . 19
2.3 Schematic overview of the communication pattern implemented for the out-of-core SVD. 20

3.1 Performance breakdown of the implemented SVD algorithm into the phase 1 KernelAbstractions-based Julia band-diagonalization, phase 2 bulgechasing, and phase 3 diagonalization. 28
3.2 Performance comparison of LAPACK and CUSOLVER SVD functions with the KernelAbstractions-based Julia band-diagonalization. 30
3.3 Performance comparison of KernelAbstractions-based Julia tiled band-diagonalization and tiled QR. 31
3.4 Performance comparison of KernelAbstractions-based Julia tiled band-diagonalization and CUSOLVER SVD for different data types. 33
3.5 Performance comparison of KernelAbstractions-based Julia tiled and CUSOLVER SVD and QR for different data types. 34

List of Code Fragments

2.1 SVD Algorithm . . . 21
2.2 Definition of specialized QR multiplication functions for *GeneralTiledMatrix* data types. 23
2.3 QR Algorithm . . . 24
2.4 LQ kernels expressed as transposes of QR kernels. . . . 25

A.1 QR kernel written in KernelAbstractions.jl. . . . 38
A.2 Partial bulge-chasing code. . . . 39
A.3 Definition of data types for the singular value decomposition. . . . 41

List of Tables

2.1 Hardware specifications for benchmarking . . . 26

Introduction

HPC accesibility

Over the last decade, the amount of available data has grown exponentially[2], resulting in today's applications processing terabyte-sized datasets[3–5]. As a result of the increased data availability, computer science has embedded itself in every scientific field, and simulations are becoming a requirement for almost any theoretical or experimental study. The field of High-Performance Computing (HPC) has developed in kind, but such advanced computational tools have not necessarily become easier to use. This raises the question: "Should every scientist become an HPC specialist to keep up with the demands of modern science?"

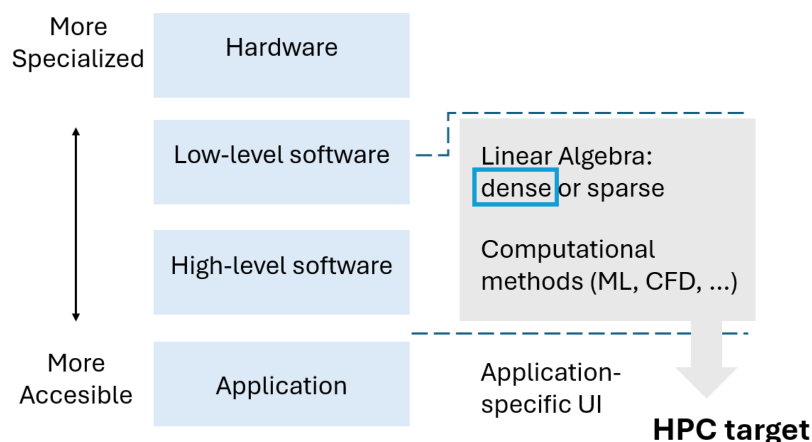


Figure 1: The goal of the current work in the scope of high-performance computing. HPC ranges from more accessible user-level applications to high-level software and low-level software, to the most specialized hardware programming. Today linear algebra underlying many more complex computational methods such as machine learning and computational fluid dynamics is typically situated between low-level and high-level software. The goal of this work is to bring linear algebra to a more accessible application level.

This work proposes an alternative and introduces a hardware-agnostic, scale-agnostic, and type-agnostic tiled linear algebra framework in Julia designed to enhance accessibility and usability without compromising performance. Currently, linear algebra and advanced computational methods find them approximately between the high-level and low-level software. The developed abstraction would bring the dense linear algebra field towards the user-application level as shown in Figure 1, increasing its accessibility to the scientific community.

Generic frameworks

As we enter the post-Moore era, and hardware approaches the bounds of clock speed, we can no longer expect automatic performance gains over time. To keep up with increasing computational demands, alternative avenues and new technologies need to be explored. However, these developments in hardware seldom provide free performance gains and require re-design at every step. [6]

The popular linear algebra libraries are a prime example of an application that has constantly evolved to run on the newest hardware architecture since the launch of the first supercomputers 60 years ago[7, 8]. In the 1960s, the emergence of vector operations resulted in the vector-optimized LINPACK library[9]. Next, the development of the cache-optimized and tiled LAPACK library followed[10]. In the nineties, the growth of multicore architectures gave rise to the ScaLAPACK library[11]. The PLASMA/MAGMA libraries were developed subsequently for GPU and hybrid architecture[12]. Finally, in recent decades SLATE has been developed as the successor to ScaLAPACK to include more memory flexibility and heterogeneous architecture support[13]. In the meantime, the original LINAPACK libraries have also been subject to revision for GPU- and multicore architecture. [14]

One of the lead authors of the linear algebra libraries libraries has recently stated that:

> "[With every new hardware] we scramble for the next three or four years to figure out how to use it effectively, [which is] an incredibly intensive process of redesigning algorithms"
>
> — Jack Dongarra, 2024[15]

The secondary goal of the tiled hardware-agnostic, scale-agnostic, and type-agnostic tiled linear algebra framework proposed in this work is genericness. Julia language's multiple-dispatch and type inference facilitate such frameworks by allowing composability: the same function can be re-used or re-defined for a different data type, making implementations future-proof as novel components can be seamlessly integrated into an existing ecosystem.

Research questions

This thesis aims to answer the following research questions:

1. ***Tiled Abstraction:*** What type of computational abstraction can be designed for tiled linear algebra that can serve multiple data sizes and hardware types?

2. ***Singular value decomposition:*** How can a GPU-accelerated out-of-core SVD algorithm be implemented in this abstraction?

3. ***Performance:*** Does such an abstraction have an impact on performance?

Contributions of the work

The contributions of this work are as follows:

1. ***Tiled Abstraction:*** We implement a hardware-agnostic, large-scale tiled linear algebra framework in Julia. This framework attempts to provide a higher-level abstraction available to non-HPC specialists who desire to develop linear algebra using HPC computing resources.

2. ***Singular value decomposition:*** We implement a tiled singular value decomposition in this framework, demonstrating how the same algorithm can be used for a GPU-only band-diagonalization of matrices, or for out-of-core GPU/CPU-combined band-diagonalization. Additionally, the extensibility of the framework for other algorithms is demonstrated by implementing a tiled QR factorization.

3. ***Performance:*** The performance of the implemented tiled singular value is demonstrated: it can calculate singular values for matrices larger than the GPU memory, at a lower matrix size-dependence than CPU-based methods.

Thesis organization

The singular value decomposition and its computation, as well as the symbiosis of tiled linear algebra fameworks with modern hardware architecture, are discussed in Chapter 1, the three-phase QR-based tiled single value decomposition algorithm and its implementation are discussed in Chapter 2, and its performance benchmarking in Chapter 3. Conclusions and avenues for further work are discussed in Chapter 4.

Chapter 1

The SVD and its computation on modern computing architecture

1.1 The singular value decomposition and its numerical computation

As the size of databases continues growing, the singular value decomposition is an algorithm of particular interest because it reveals the best low-rank approximation of datasets. It is widely used in machine learning[16], image processing,[17–19], quantum information theory[20, 21], and in any field that uses principal component analysis, matrix rank estimation, matrix inverse, or least squares calculation.[21, 22] Figure 1.1 shows a schematic representation of its format.

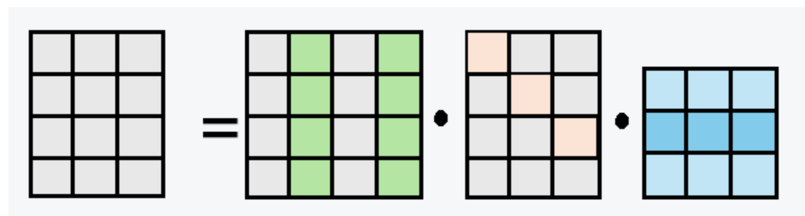


Figure 1.1: Schematic representation of a singular value decomposition: the product of a unitary matrix, a diagonal matrix, and the transpose of a unitary matrix.

The reduced singular value decomposition (SVD) $U\Sigma V^T$ of a general matrix $A \in \mathbb{C}^{m\times n}$ with $r = \text{rank}(A)$, where $U \in \mathbb{C}^{m\times r}$ and $V \in \mathbb{C}^{n\times r}$ are unitary matrices and $\Sigma \in \mathbb{C}^{r\times r}$ is a non-negative diagonal matrix, can be understood as follows: [23, 24]

- The columns of U and V are the unitary bases for the column and row space of A, respectively. We call them the left and right singular vectors.
- The column-vectors of U and V are the eigenvectors of the AA^T and A^TA matrices, respectively. Σ^2 contains their eigenvalues.
- The SVD decomposes the linear mapping $A : \mathbb{C}^m \rightarrow \mathbb{C}^n$ into a rotation, an extension and a rotation. In other words, it maps the m-dimensional unit hypersphere to an n-dimensional hyperellipse.

The full SVD $A = U'\Sigma'V'^T$, where $U' \in \mathbb{C}^{m\times m}$, $V' \in \mathbb{C}^{m\times n}$, and $\Sigma' \in \mathbb{C}^{m\times n}$ can be obtained by completing the compact SVD to a full basis and completing the Σ matrix with zeros. In the remainder of this work, we will assume full-rank input matrices and will use the term singular value decomposition to refer to the compact singular value decomposition.

For the numerical calculation of the dense singular value decomposition three groups of algorithms can be considered[25, 26]:

- ***Jacobi methods:*** The Jacobi-methods[27] consist of the application of subsequent appropriate Givens rotations to zero elements.
- ***Divide-and-conquer methods:*** Divide-and-conquer methods[28] are an extension of Jacobi-methods for multicore architecture: the original matrix is split into sub-blocks that are each diagonalized with the Jacobi-methods, and then joined back together.
- ***QR-based methods:*** These methods[29–31] consist of the application of unitary Q matrices to the whole matrix. The Q-factors are obtained by the QR-factorization of a small subpart of the matrix in order to zero elements out.

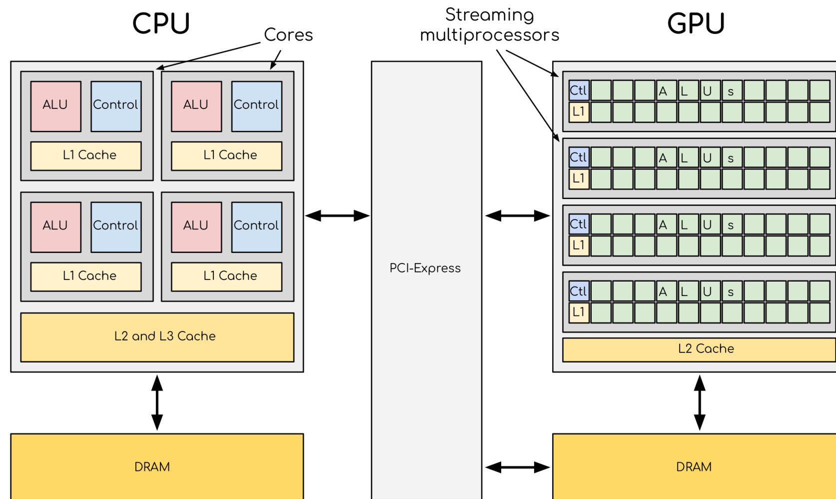
While the QR method is plagued by serial bottlenecks, it is in practice still generally considered faster than the Jacobi methods, as Jacobi-methods require iterating until convergence is reached, while the number of iterations for bidiagonalization using the QR algorithm is deterministic. All three strategies have been optimized for modern GPU and multicore architectures, but the current state-of-the-art methods in terms of performance are tiled QR-based algorithms [29–31]. Extensive work has been done into their optimization, in particular for scheduling overhead [32, 33]. Novel avenues for optimizing SVD performance include nested task parallelism[34].

A series of different approaches exist for the computation of the highest or lowest singular values; randomized methods [4, 35–39], Krylov subspace methods[40, 41], Tucker decomposition[42], power methods[30], QR-based Dynamically Weighted Halley methods,[43], and polar decomposition[44, 45]. For the estimation of the number of singular values in a specific range, specialized spectrum methods are available[46]. Additionally, specific methods exist for particular matrix types, such as the Multiple Relatively Robust Representations for a symmetric tridiagonal matrix [47], the K-means clustering for sparse data [48]. For higher-order tensors, many of the same methods have been extended. [49, 50]

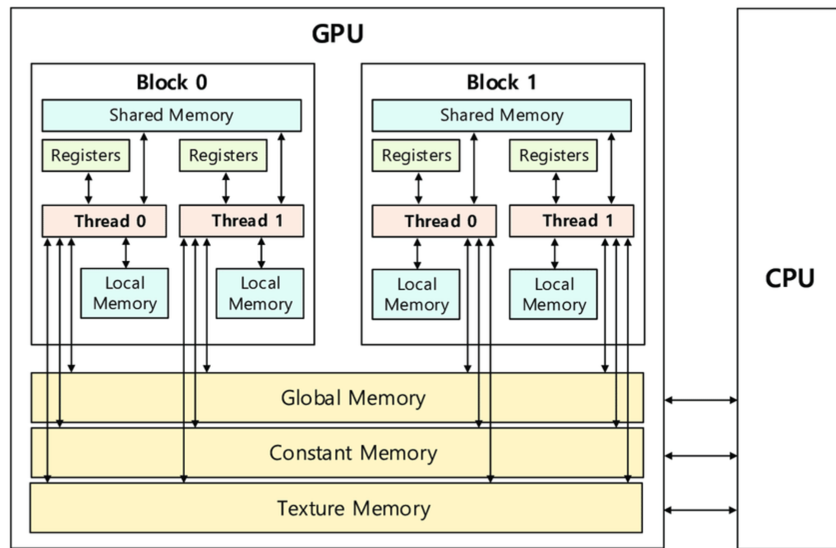
1.2 Tiled linear algebra algorithms for modern hardware

To understand the value of tiled algorithms, let us consider the design of modern architectures used to accelerate computing, GPUs, and their differences with CPUs. Where CPUs consist of highly optimized cores that can process a large amount of complicated operations, but only a limited amount simultaneously, GPUs have a large amount of small cores, called threads, which handle simple operations best, as represented in Figure 1.2(a). They are grouped in blocks, inside which memory is limited and communication between the different threads in a block goes relatively fast(approximately 30 cycles[51, 52]) through their shared memory shown in Figure 1.2(b). However, communication between the different blocks runs through the lower-bandwitch GPU global memory (access speed approximately 500 cycles per [51, 52]). In other words: there is now an

abundant number of computes that can happen, but communications come at a high cost. Thus, the development of linear algebra routines that prioritize the localization of data at the cost of additional flops is a natural result. [29, 53, 54]



(a) GPU architecture schematic from [55]



(b) GPU architecture schematic from [56]

Figure 1.2: GPU architecture schematic, demonstrating the many small versus few large computing cores on the GPU and CPU, respectively in sub-figure (a). The memory hierarchy is illustrated in sub-figure (b): several threads are organized into a thread-block with a shared high-bandwidth memory and have access to the global GPU lower-bandwidth memory. Finally, the communication between the CPU and GPU has the lowest bandwidth.

Abstracting the tiled algorithms into an accessible layer with widely-utilizable building blocks has been the topic of extensive recent research. Such frameworks has been developed for out-of-core tiled algorithms for neural network acceleration [57], and for linear algebra in C/C++ [58]. Furthermore, other libraries have come out providing other levels of abstractions, such as a generic set of BLAS kernels for the development of custom-kernel developed for user-accessibility in C[59, 60], frameworks for development of kernels for heterogenous architecture [61], and frameworks for vendor-agnostic dense linear algebra[62].

The proposed framework in this work does not have the ambition to replace the existing libraries, rather its purpose is to demonstrate the capabilities of the Julia Language in implementing such general lightweight frameworks with maximal composability and minimal development requirements.

Cache and memory access effects

Considering the lower-memory bandwidth between the different thread-blocks on a GPU , making optimal use of this memory is crucial. On the GPU there are two main memory access effects to consider: memory coalescing and bank conflicts[54, 63].

- **Memory coalescing** for accessing global memory by threads in a warp. Global memory is accessed in *data chunks* of at least 32 bytes. When multiple threads in a warp access non-consecutive data pieces smaller than 32 bytes, each data access will necessarily read 32 bytes. In contrast, when aligned data is read by threads in a warp, a single 32-byte memory chunk read can be coalesced to multiple threads in the warp. For memory coalescing to be possible, the data types need to be of size 1,2,4,8, or 16 bytes, and the first address needs to be a multiple of the type size. Misalignment typically occurs when accessing non-contiguous data and can be prevented by memory padding.

- **Bank conflicts** for accessing shared memory by threads in a warp. Shared memory is divided into 32 memory banks, each of a set size (typically 4 or 8 bytes). Data is contiguously distributed over the lanes, and each memory bank can only be accessed by a single thread at a time. For optimal performance, all memory lanes should be used and the number of bank conflicts (i.e. multiple threads accessing the same lane) minimized.

In a recent benchmark of the novel NVIDIA architecture[63], the authors describe how understanding the architecture behavior is a critical factor in optimizing performance: "Authors have also shown that knowledge of instruction encoding and microarchitectural behavior is necessary to achieve this full potential." In the framework we propose here, we attempt to make some of these parameters and characteristics that optimize performance available on a higher-level, facilitating performance fine-tuning without requiring low-level coding.

Chapter 2

Methods

2.1 The out-of-core three-phase tiled SVD

The out-of-core tiled QR-based singular value decomposition as described in [64] is implemented in Julia. This framework consists of a three-step-singular value algorithm that is visualized in Figure 2.1. The QR-based algorithm consists of applying appropriate unitary transformations so that the elements outside the bidiagonal become zero, as unitary transformations preserve the singular values. The algorithm is particularly appropriate for GPU and multicore architectures as it localizes calculations: the number of operations increases locally, but communication is limited to smaller data units.

1. **STEP 1 BAND-DIAGONALIZATION:** The matrix is divided into $n \times n$ smaller blocks. For each n, a QR sweep and an LQ sweep are applied. The QR sweep consists of the calculation of the QR-factorization of the first block of the first row to zero out the lower triangular part of the block and applying the unitary Q matrix to the remainder of the row to retain a matrix with the same singular values. Next, the QR factorization of the resulting upper triangular matrix combined with the first block of the second row is calculated. This factorization zeros out the full lower block and the resulting unitary Q matrix is applied to the remainder of the first and second row. The same process is repeated for the third row, and this concludes

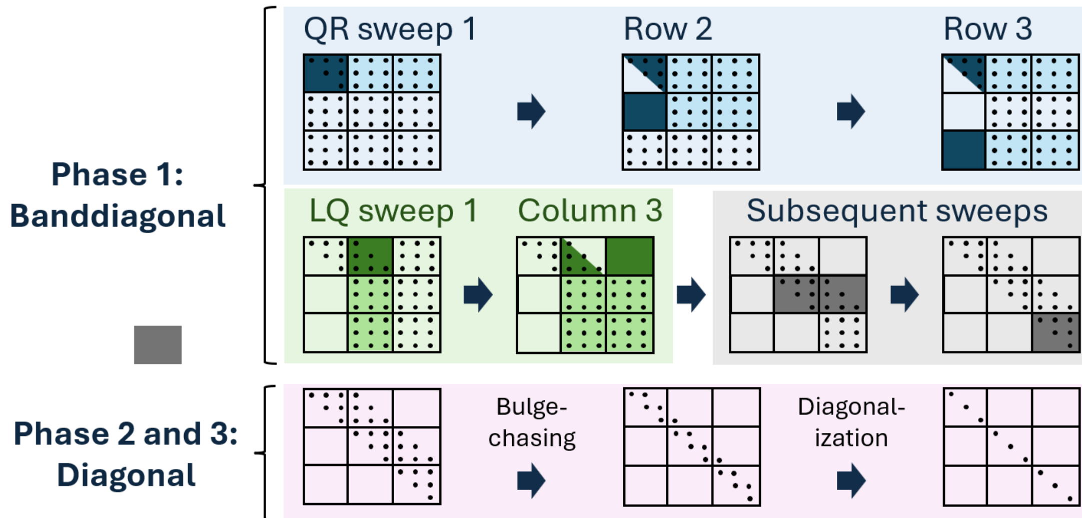


Figure 2.1: Schematic overview of the three-step QR-based singular value decomposition algorithm as described in [64]. The top two rows show step one, the band-diagonalization. In the first row, the QR sweep is shown. The left figure on the first row represents the calculation of the QR-factorization of block 1,1; and the application of the resulting Q-matrix on the blocks to the right of it. Block 1,1 becomes upper-triangular. The second figure on the first row represents the calculation of the QR-factorization of the upper triangular first block and the first block of the second row; and the application of the resulting Q-matrix on the blocks to the right of it so that the block becomes zero. The third figure on the first row represents the same procedure as for the second figure for the third row. The first and second images of the second row represent the LQ sweep: the calculation of LQ-factorizations and the application of their Q-factors to the blocks below to obtain lower triangular and zero matrices on the first row while preserving the singular values. The third and fourth images of the second row represent the QR sweep and LQ sweep for the second and third iterations. Finally, the third row shows the second and third steps, the bulgechasing and diagonalization.

the first QR sweep. The LQ sweep is the transpose of the QR sweep: transformations are calculated to obtain a lower triangular block and to zero out blocks and the transformations are applied on the columns instead of the rows. This process is repeated for every n and results in a banded diagonal matrix. This step contains a large amount of natural parallelism, in particular when applying the Q-factors to the rows and columns each block is independent, making it well-suited for GPU computations.

2. **STEP 2 BI-DIAGONALIZATION bulgechasing:** Similarly to the bidiagonalization, for every line a QR factorization is calculated to zero out elements and the resulting transformations are applied to the remainder of the matrix. This application then results in more non-zero

elements, which are then also annihilated in subsequent QR sweeps. This step is visualized in Figure 2.2. In the implemented algorithm, rather than apply a full band-reduction at every line, subsequent half-band reductions are applied, as this reduces the size of the matrices for which QR factorizations and matrix multiplies need to be calculated. Progressive half-band reductions are then executed. The smaller matrix sizes and large amount of communication needed combined with the limited amount of natural parallelism in this step, make it better suited for CPU-calculation.

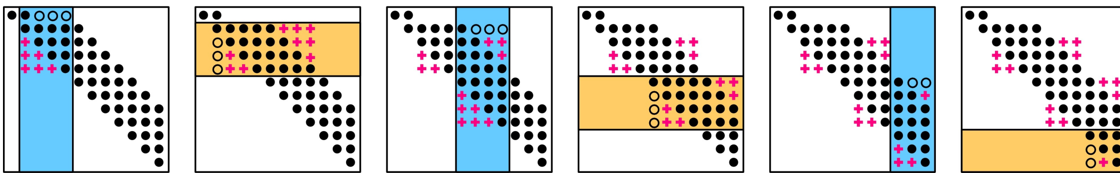


Figure 2.2: Schematic overview of the bulgechasing step of the three-step SVD algorithm, figure from [28]. The leftmost figure shows the QR-annihilation step of the lower diagonal elements, resulting in additional non-zero elements in the remainder of the row after the application of the Q-factor. The second figure then shows the annihilation of these new non-zero values, by an LQ step and the resultant additional non-zero elements at the remainder of the column. The additional elements are 'chased down' by subsequent QR and LQ sweeps until all have been zeroed out.

3. **STEP 3 DIAGONALIZATION:** An iterative procedure, for which the standard LAPACK divide-and-conquer *bdsdc* function is used. This procedure is typically faster on CPUs.

The algorithm is typically implemented as GPU-accelerated, where the first step is executed on the GPU, and steps two and three are calculated on the CPU. Executing the full first phase on the GPU without intermediate communication with the CPU is only possible when the matrix fits on GPU memory with sufficient memory for calculation left. In this case, the matrix is communicated once to the GPU, retained and calculated there, and sent back to the CPU. For matrices larger than the GPU memory, the same algorithm is used but is implemented as out-of-core, meaning that the CPU is used as 'storage space' while the GPU is used as a computing unit. The out-of-core algorithm includes more communication between the CPU and GPU, making it slower than a pure GPU algorithm, but can process larger sizes. In addition, it is typically faster than CPU-only

algorithms when those are compute-bound (versus memory-bound). As such, it combines the best of both strategies.

The optimal pattern of memory movement for out-of-core large SVD has been studied and described in [31]. We adopt a simplified version of methodology as follows: the first row and first column of QR and LQ sweeps, respectively, are held in GPU memory during a sweep, while the other rows are communicated in memory between the application of the Q-factor and communicated out after completion of this step. The work in [31] describes this framework for even larger matrices and communicates the rows and columns partially while the calculation is ongoing. We also overlap the communication and the computation step using multiple threads. This methodology is followed until the matrix formed by all the blocks right and below the diagonal sweep tile fits into GPU memory, at which point the tiled algorithm is applied as GPU-only. Figure 2.3 shows this schematically: the dark blue row is computed, while the light blue rows move in and out of the GPU memory from the CPU memory. The top row is retained in memory. Once the algorithm reaches a diagonal tile, the matrix below and right from that tile is retained in the GPU.

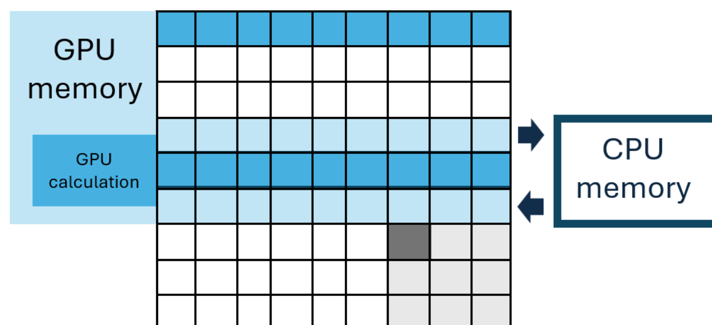


Figure 2.3: Schematic overview of the communication pattern implemented for the out-of-core SVD, based on [31]. The top row is retained in the GPU memory, while the remaining rows circulate in and out from the CPU memory. The middle row of the retained rows is used for computation, while the other two are scheduled for communication.

2.2 A general framework for tiled SVD

```
function Blockbidiag!(A::GeneralTiledMatrix; kend=A.no_tiles)
    verify_kerneldims(A)

    for k in 1:kend

        QRandmulQt1!(A,k, alg="SVD")
        for row in k+1:A.no_tiles
            QRandmulQt2!(A,row,k, alg="SVD")
        end

        (k==A.no_tiles) && break

        LQandmulQt1!(A,k, alg="SVD")
        for col in k+2:A.no_tiles
            LQandmulQt2!(A,k,col, alg="SVD")
        end
    end
    finish_algo!(A, kend, alg="SVD")
end
```

Source Code 2.1: The Singular Value Decomposition Algorithm generic code that is applicable for both GPU-only and out-of-core GPU-accelerated execution, for several data types and hardware types. Lines 6-9 execute a QR sweep, while lines 13-16 execute an LQ sweep of input matrix A. The parameter *kend* at which sweep the algorithm stops is optional. The *finish_algo!* function is included for finishing up communication in the out-of-core version.

While the out-of-core tiled singular value decomposition and the in-core singular value decomposition use the same algorithm, with as the only difference the communication step, many programming languages require reprogramming of the whole algorithm for each hardware type (CPU, GPU), each hardware vendor (Nvidia, AMD) and for each data type. Julia language's multiple-dispatch and type inference enables LLVM to generate optimized machine code for performance over various argument types.[65, 66] The multiple-dispatch allows the same function to be redefined for various input types, and enables composability of the language, which renders the implementations future-proof as new components in the systems can be seamlessly included in an existing ecosystem. Composability also enables writing generic frameworks for tiled algorithms,

mapping one-to-one with pseudo-algorithms, and functioning over several types of hardware. This is demonstrated for the SVD algorithm in Code 2.1.

The generic function *Blockbidiag!* can then be used by *GeneralTiledMatrix*, where the *QRandmult!* algorithms can be redesigned for their specific type while underlying functions could be generic again. We design to *GeneralTiledMatrix* types: a *TiledMatrix* type designed for the GPU-only tiled band-diagonalization containing a reference to the tiled views of the GPU-located matrix, and a *LargeTiledMatrix* type for the out-of-core band-diagonalization containing 4 rows used to circulate the data in and out of the GPU memory. The code for their definition is included in Appendix A.3.

```
function QRandmulQt1!(A::TiledMatrix, k::Int; alg="QR")
    QR1!(A,k)
    Qtapply1_par!(A, k)
    alg=="SVD" && triu!(A,k,k)
end

function QRandmulQt1!(A::LargeTiledMatrix, k::Int,
                        prevR::Bool,currentR::Bool, zerofactor::Bool)

    if (k>1 && currentR) #import first row back into memory
        getandset_first!(A, k-1,k, prevR, currentR )
    elseif !currentR
        getandset_first!(A, k,k, prevR, currentR )
    end

    @sync begin
        Threads.@spawn begin
            CUDA.@sync begin
                if (k+Int(!currentR)<A.no_tiles)
                    recycle_tilerow(A, k, k+1+Int(!currentR), #next row in
                                    currentR, false, false)
                end
            end
        end

        Threads.@spawn begin
            CUDA.@sync begin #calculation on current row
                QR1!(A,k)
                Qtapply1_par!(A, k)
```

```
30                  zerofactor && triu!(A,1)
31              end
32          end
33      end
34      finish_recycle!(A)
35      CUDA.synchronize()
36  end
```

Source Code 2.2: Definition of specialized QR multiplication functions for *LargeTiledMatrix* and *TiledMatrix* data types. The *QRandmult!* algorithm for the *TiledMatrix* type consists of the QR factorization of the first row (line 2) and the parallel multiplication of the tiles to the right of it by the Q-factor (line 4), followed by rendering the first block zero if the algorithm is for the SVD, rather than QR factorization (line 5). The function is identical for the *LargeTiledMatrix*, but includes the simultaneous launch of threads that execute the data communication to obtain the next row from the CPU memory (lines 17-24), as well as the importing of the first row into memory before starting the calculation (lines 10-14). The *QR1!* function of the *TiledMatrix* then operates on a tile, while the *QR1!* function of the *LargeTiledMatrix* operates on a row in the memory.

The *QRandmult!* algorithm for the *TiledMatrix* type displayed in Code 2.2 consists of the QR factorization of the first row and the parallel multiplication of the tiles to the right of it by the Q-factor. The function is identical for the *LargeTiledMatrix*, but includes the simultaneous launch of threads that execute the data communication to obtain the next row from the CPU memory. The *QR1!* function of the *TiledMatrix* then operates on a tile, while the *QR1!* function of the *LargeTiledMatrix* operates on a row in the memory.

2.3 The out-of-core tiled QR

To demonstrate the ease with which one tiled algorithm can be implemented based on a similar one, the tiled QR algorithm is implemented in Code 2.3. It consists of the consecutive application of QR sweeps in the same manner as in the SVD algorithm, without the LQ sweeps. In this thesis, the viability of a general tiled framework in Julia is demonstrated by applying it to the singular value decomposition algorithm and the QR algorithm. The same framework could in the future be readily extended to other tiled algorithms, in particular the tiled LU decomposition[67].

```julia
function tiled_QR!(A::GeneralTiledMatrix; kend=A.no_tiles)
    verify_kerneldims(A)
    for k in 1:kend
        QRandmulQt1!(A,k)
        for row in k+1:A.no_tiles
            QRandmulQt2!(A,row,k)
        end
    end
    finish_algo!(A, kend)
end
```

Source Code 2.3: The QR Decomposition Algorithm generic code that is applicable for both GPU-only and out-of-core GPU-accelerated execution, for several data types and hardware types. Lines 6-9 execute a QR sweep. The parameter *kend* at which sweep the algorithm stops is optional. The *finish_algo!* function is included for finishing up communication in the out-of-core version.

2.4 Performant generic QR kernels

To make the algorithm hardware-agnostic across GPU vendors, custom QR kernels for the GPU are written using the KernelAbstractions.jl package [68]. Recent advances in vendor-agnostic frameworks make such a cross-platform implementation possible[69–72]. The kernels are written as two-dimensional kernels that operate on data of size (32,32) with a block of threads (32, x) where x is a parameter that can be defined for the application of the Q-factor and is set to 32. One such kernel is included in Appendix A.1 as an example. While extensive work has been done into the optimization of specific kernels [73], the goal of this work is to implement rudimentary kernels based on simple principles and demonstrate this results in adequate performance such that the end-user-based implementations are not required to be hyper-optimized for adequate performance and accessibility is maintained.

Julia Language's functionality of transposing and obtaining references to parts of matrices, called views, allow the LQ kernels to be expressed as transposes of the QR kernels instead of fully rewritten, as demonstrated in Code 2.4.

```
applyLQ1!(A,T;ndrange)=applyQR1!(A',T,ndrange=ndrange)
applyLQ2!(A,B, T;ndrange)=applyQR2!(A',B', T,ndrange=ndrange)
applyQt1L!(A,B,T;ndrange)=applyQt1!(A',B',T,ndrange=ndrange)
applyQt2L!(A,B,C,T;ndrange)=applyQt2!(A',B',C', T,ndrange=ndrange)
```

Source Code 2.4: LQ kernels expressed as transposes of QR kernels. Kernel *applyQR1!* takes as input a matrix for which to calculate a QR factorization, a memory space T for the scaling factors for the householder reflectors, and the desired total number of threads across blocks *ndrange*. Kernel *applyQR2!* takes as input an upper triangular matrix, and a matrix, for whose combination to calculate a QR factorization, a memory space T for the scaling factors for the householder reflectors, and the desired total number of threads across blocks *ndrange*. Kernel *applyQt1!* takes as input a matrix that contains QR factorization, a memory space T with its scaling factors for the householder reflectors, the matrix on which to apply the Q-factor, and the desired total number of threads across blocks *ndrange*. Kernel *applyQt2!* takes as input the lower matrix resulting from the application of *applyQR2!*, a memory space T with its scaling factors for the householder reflectors, upper and lower matrix on which to apply the Q-factor and the desired total number of threads across blocks *ndrange*.

2.5 Bulgechasing and diagonalization

The bulgechasing is implemented by making use of the LAPACK-based function *geqrt!* implemented in Julia, without any parallelism across the different factorizations, but with parallelism in the QR function itself. The *views* functionality in Julia allows pointer-type access to any part of a matrix based on its indexes, facilitating the development of the bulgechasing algorithm. Part of the bulgechasing code demonstrating this functionality is included in Appendix A.2. The diagonalization uses the LAPACK divide-and-conquer *bdsdc* function.

2.6 Testing and benchmarking

The correctness of the kernels and code is tested and verified using unit tests and benchmarked against QR-based SVD algorithms CUSOLVER (GPU-only) and *gesvd!* in *gesvd* in LAPACK (CPU-only). Two hardware types are tested: a consumer laptop and a supercomputing cluster. Their technical specifications are shown in Table 2.1.

Type	**Supercomputer**	**Consumer laptop**
Vendor	**Intel**	**Intel**
Family	Xeon Gold	AMD Ryzen 7 5800H
Model	6248	80
Cores per socket	16	8
RAM/core	9GB	8GB
Clock Speed	2.5GHz	3.20 GHz
L3 Cache size	55MiB	16MiB
GPU type	NVIDIA v100 32GB	NVIDIA GeForce RTX 3050

Table 2.1: Hardware specifications for the computing resources used to obtain benchmarking. All benchmarks were run with a limit of 4 threads on both hardware types.

While the kernels are fully vendor-agnostic, the overlapping of the communication and computation is Nvidia-only for now. However, extension to other hardware is intuitive and is subject to further work.

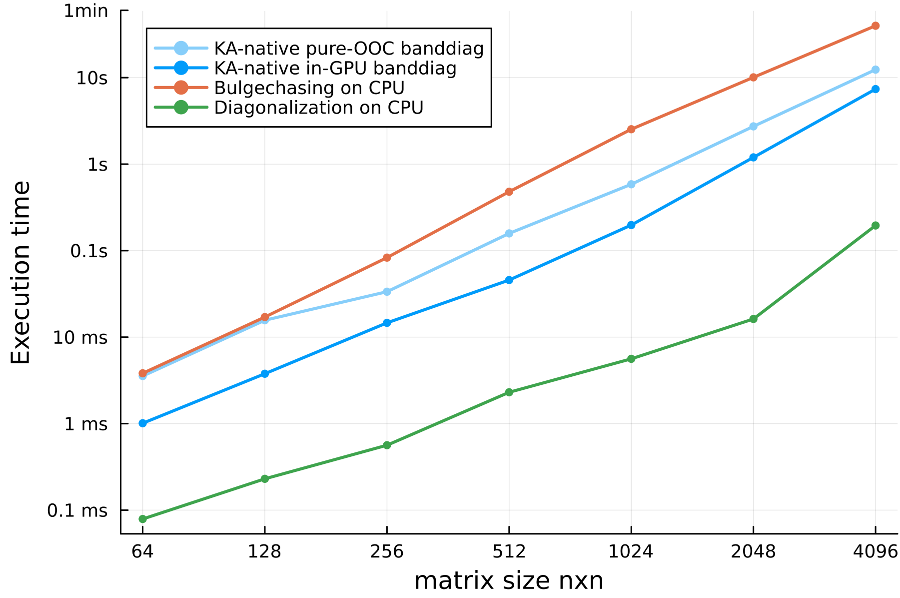
Chapter 3

Results and discussion

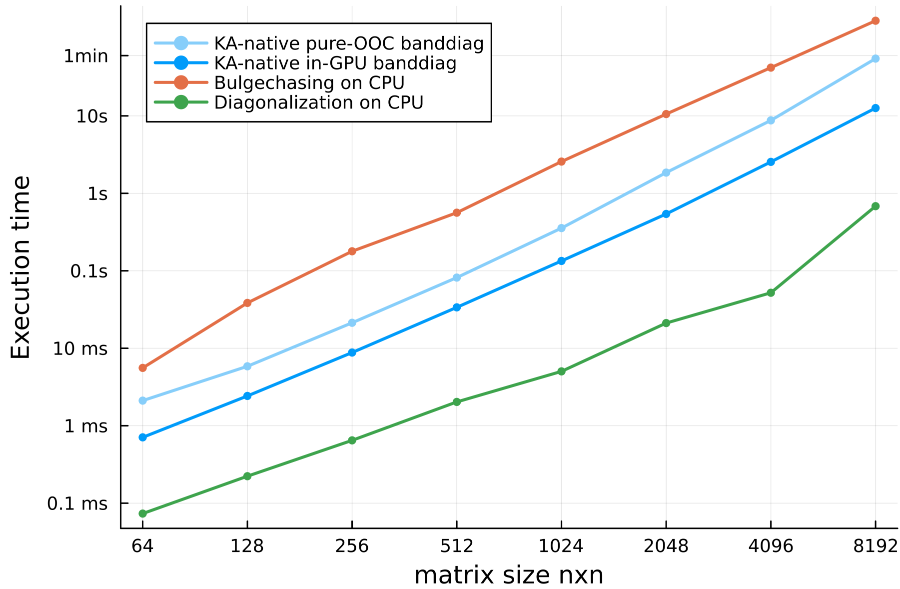
3.1 Performance benchmarking

To examine the performance of the implemented singular value decomposition algorithm, the computation time for each of the three phases, GPU-only KernelAbstractions-native band-diagonalization, bulgechasing, and diagonalization is benchmarked in Figure 3.1. Additionally, the performance of out-of-core only KernelAbstractions-native band-diagonalization is included, this refers to the tiled singular value decomposition where rows and columns are migrated to and from the CPU up and until the last QR sweep. Based on the literature[74], the first stage is the most time-consuming due to the amount of communication required.

The graph in Figure 3.1 shows the execution time in function of the matrix size. We notice that indeed the diagonalization step computation time is negligible compared to the other two stages. However, the bulgechasing step is the most time-consuming. As the band-diagonalization step has optimized specific kernels, implements multi-threading to overlap communication and computation, and is GPU-computed, it is expected to be perform better than the non-optimized rudimentary bulgechasing algorithm. Future work should include automatic parallel scheduling for the bulgechasing step to make this step competitive in the algorithm, as suggested by [74]. For the remainder of this chapter, we will examine only the band-diagonalization algorithm to obtain sensitive benchmark-



(a) Execution Time on Intel AMD Ryzen consumer laptop with RTX 3050 GPU(see Table 2.1).



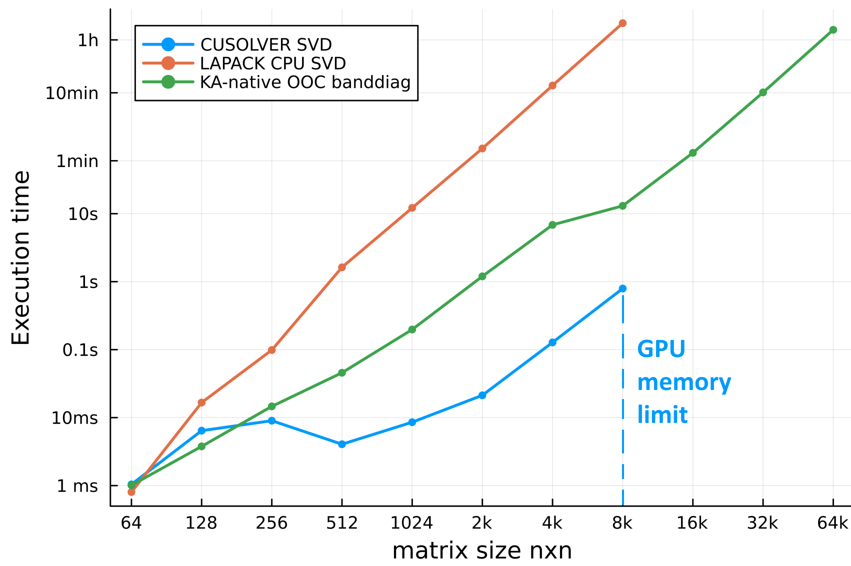
(b) Execution Time on Intel Xeon Gold supercomputer with V100 GPU(see Table 2.1).

Figure 3.1: Performance breakdown of execution time into band-diagonalization using multi-threaded Julia KernelAbstractions-based SVD on the GPU (blue), bulgechasing phase on the CPU based on LAPACK QR functions (orange), and LAPACK diagonalization (green) in function of the size n for input matrices of size $n \times n$. The total SVD time is the sum of all three methods. The comparatively largest computation time can be attributed to the bulgechasing. The KA-native band-diagonalization is shown as GPU-only without any CPU-GPU communication ("in-GPU") and as an out-of-core only version where the communication phase extends until the last tile, without holding the last part of the matrix in the GPU when possible("pure-OOC"). The difference between both is a non-matrix size dependent factor.

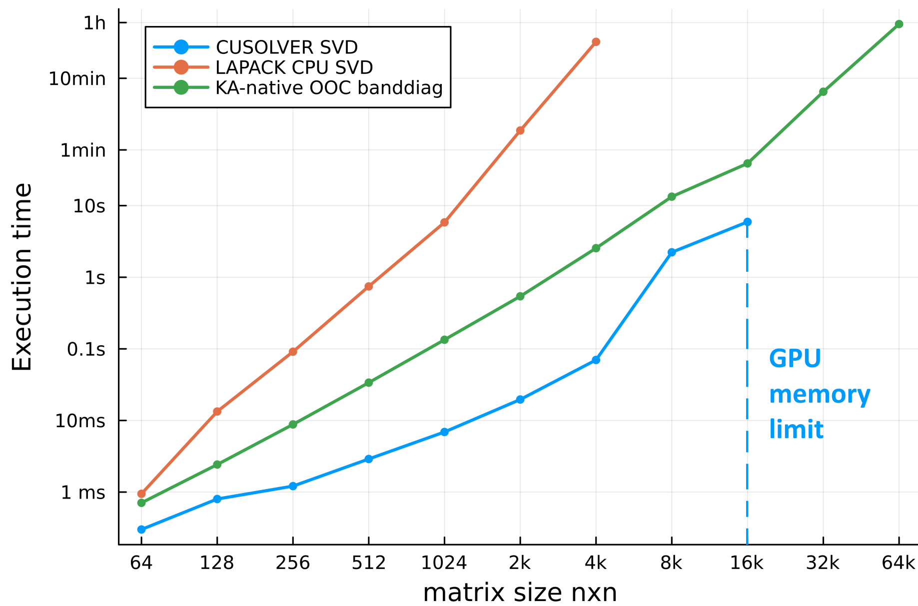
ing results focusing on this step. Comparing the in-core KA-based band-diagonalization and the out-of-core exclusive version, we find a non-matrix size dependent constant factor difference, indicating that as the matrix sizes go further up, the cost of communication relative to computation is negligible. This suggests that in optimizing for large matrix sizes in out-of-core tiled algorithms, communication cost does not necessarily need to be the focus in every case.

Figure 3.2 shows the execution time in function of the matrix sizes of the out-of-core GPU-accelerated band-diagonalization algorithm, the GPU CUSOLVER SVD, and the LAPACK CPU SVD. The figure shows a significant size-dependent speed-up of the out-of-core algorithm versus the CPU-only algorithm. For matrix sizes that are smaller than the GPU memory, CUSOLVER still provides a faster algorithm. This is hardly surprising, as NVIDIA optimizes some of their most common operations on a hardware level[63], a speed that cannot simply be matched by compiled kernels. However, the implemented band-diagonalization algorithm shows a size-dependence that is different from the CPU-based calculation, indicating that as the matrix size increases, the advantage of using GPU-accelerating increases, and can process matrices larger than the GPU memory, combining the best of both strategies. While performance is not the end goal of the tiled framework, the current benchmarking indicates such abstract frameworks do not stand in the way of performance, thanks to Julia Language's LLM-based compiler[65].

To demonstrate the portability and genericness of the framework, a tiled QR factorization was implemented based on the tiled bi-diagonalization: it contains only the QR sweeps of the algorithm. Figure 3.3 confirms that indeed the tiled QR executes approximately half of the work of the tiled band-diagonalization, demonstrating how portable the generic framework is.

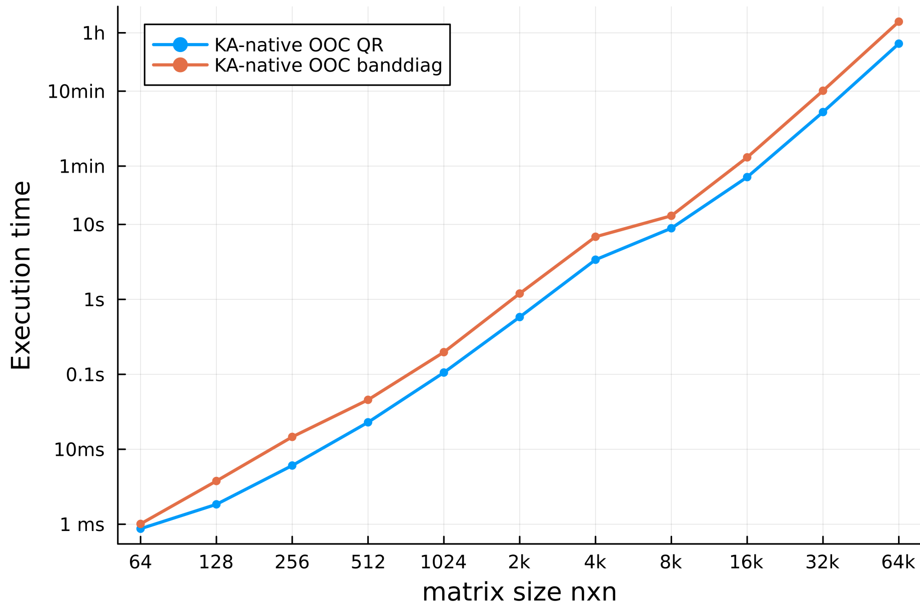


(a) Execution Time on Intel AMD Ryzen consumer laptop with RTX 3050 GPU(see Table 2.1). The GPU reaches its memory limit beyond matrix size 8k×8k.

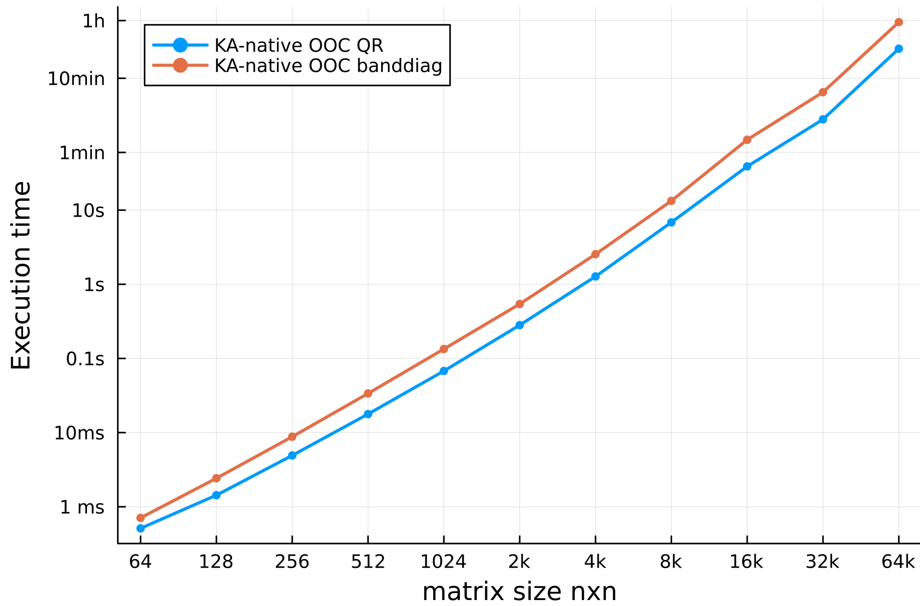


(b) Execution Time on Intel Xeon Gold supercomputer with V100 GPU(see Table 2.1). The GPU reaches its memory limit beyond matrix size 16k×16k.

Figure 3.2: Performance comparison of the execution time of LAPACK (orange) and CUSOLVER (blue) SVD functions with the multi-threaded Julia KernelAbstractions-based (KA-based) band-diagonalization with thread-sizes (32,8) in function of the size n for input matrices of size $n \times n$. The CUSOLVER-based SVD includes the time it takes to copy the matrix into GPU memory and back. It is limited to matrices that fit in the GPU memory. The KernelAbstractions-based out-of-core SVD can calculate the SVD of matrices larger than the GPU can hold, at a lower size-dependence than the LAPACK function.



(a) Execution Time on Intel AMD Ryzen consumer laptop with RTX 3050 GPU(see Table 2.1).



(b) Execution Time on Intel Xeon Gold supercomputer with V100 GPU(see Table 2.1).

Figure 3.3: Performance comparison of the execution time of KernelAbstractions-based Julia tiled band-diagonalization (orange) and tiled QR (blue) in function of the size n for input matrices of size $n \times n$. The tiled algorithms show the same size dependence, with only a non-size dependent factor difference.

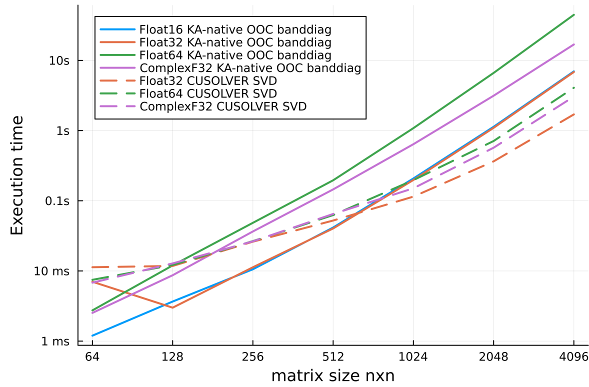
3.2 Data type-agnostic performance

Figure 3.4 demonstrates the correctness and performance of the algorithm for several data types through a single implementation, enabled by Julia language's multiple-dispatch facilitated composability. The figure displays a non-matrix size dependent difference in performance between data types, demonstrating how the single-API implementation still provides high performance.

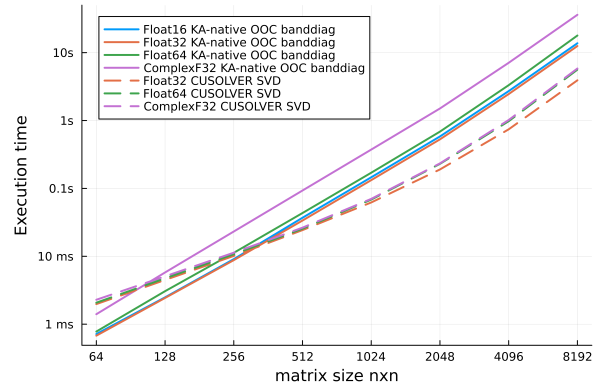
Figure 3.4 does not contain a plot for the `CUSOLVER` performance of Float16 data types, as it has not yet been included in Julia-wrappers for CUDA, highlighting the challenges that come with type-specific implementations and the advantages of composability and generality of frameworks: new data types can seamlessly integrate into an existing environment, without requiring additional adaptations in many cases.

3.3 Optimization of kernel sizes

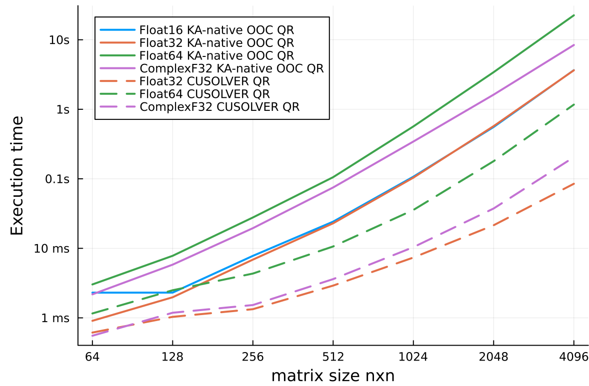
Figure 3.5 displays the performance of the KernelAbstractions-based in-GPU band-diagonalization for different thread block sizes for which the algorithm is executed. This comparison was used to optimize the individual QR kernels. The QR-kernels themselves are set as either symmetric 2D thread-block-sizes (32,32) or (32,1) while the application of the Q-factors on other tiles is the function where the thread-block-size can be varied to optimize performance. Kernels with a smaller second dimension have each thread operate on multiple elements, reducing some of the kernel launch overhead. In the figures, we notice the optimization of thread-block-sizes makes no significant difference as long as the kernel-block-size first dimension is larger. Subtle differences can be observed for the larger thread-block-sizes (32,32) and (32,16) that perform worse than the optimal thread-block-size of (32,4).



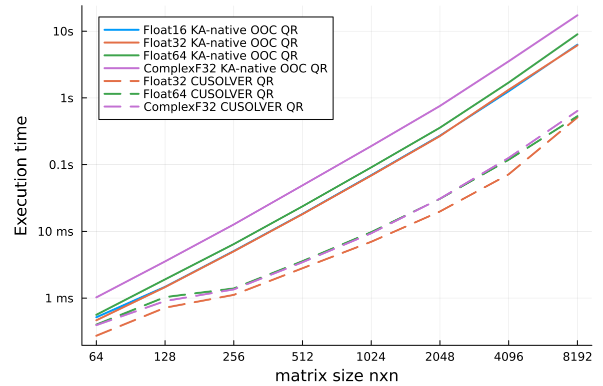
(a) Execution Time of tiled band=-diagonalization/SVD algorithm on Intel AMD Ryzen consumer laptop with RTX 3050 GPU(see Table 2.1).



(b) Execution Time of tiled band-diagonalization/SVD algorithm on Intel Xeon Gold supercomputer with V100 GPU(see Table 2.1).

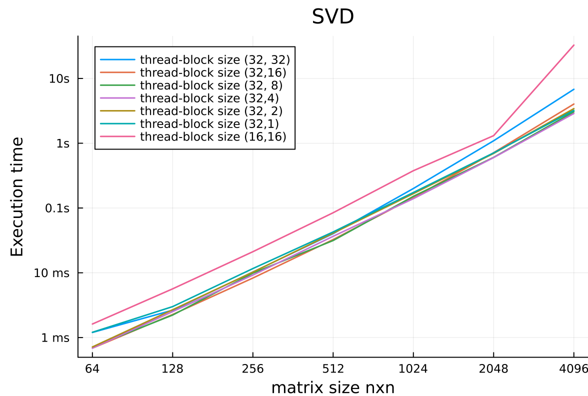


(c) Execution Time of tiled QR on Intel AMD Ryzen consumer laptop with RTX 3050 GPU(see Table 2.1).

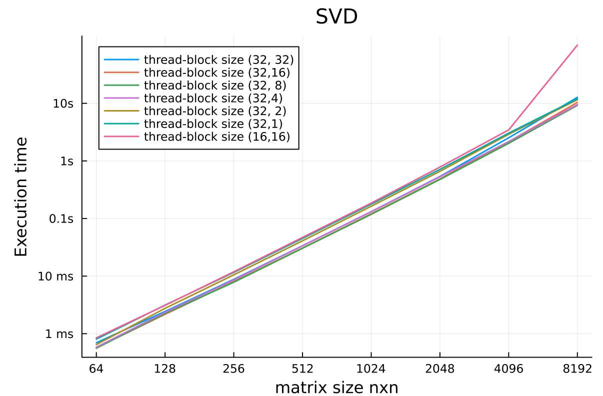


(d) Execution Time of tiled QR on Intel Xeon Gold supercomputer with V100 GPU(see Table 2.1).

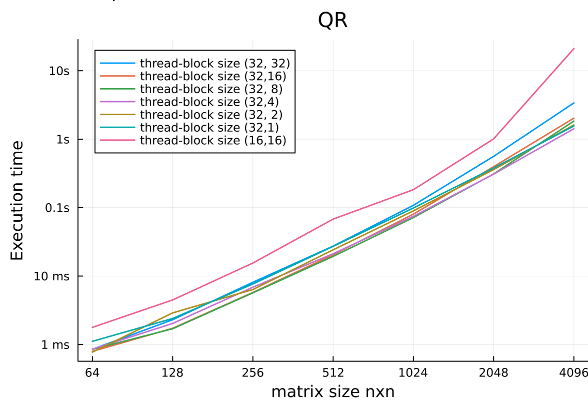
Figure 3.4: Performance comparison of the execution time of KernelAbstractions-based Julia tiled band-diagonalization and CUSOLVER SVD for Float16, Float32, Float64, and Complex32 data types in function of the size n for input matrices of size $n \times n$. No Float16 performance is included for the CUSOLVER SVD since it has not yet been included in the julia-wrapper for CUSOLVER. [75] The KA-based algorithm shows no significant matrix size-dependent performance decline between data types.



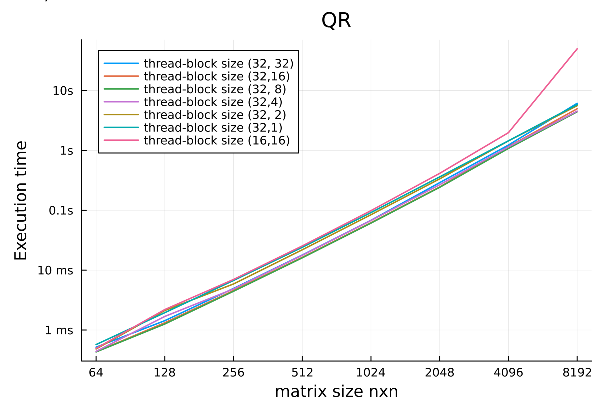
(a) Execution Time of tiled band-diagonalization/SVD algorithm on Intel AMD Ryzen consumer laptop with RTX 3050 GPU(see Table 2.1).



(b) Execution Time of tiled band-diagonalization/SVD algorithm on Intel Xeon Gold supercomputer with V100 GPU(see Table 2.1).



(c) Execution Time of tiled QR on Intel AMD Ryzen consumer laptop with RTX 3050 GPU(see Table 2.1).



(d) Execution Time of tiled QR on Intel Xeon Gold supercomputer with V100 GPU(see Table 2.1).

Figure 3.5: Performance comparison of the execution time of KernelAbstractions-based Julia tiled band-diagonalization and `CUSOLVER` SVD (first row), and for KernelAbstractions-based Julia tiled QR and `CUSOLVER` QR (second row) for several different thread block sizes in function of the size n for input matrices of size $n \times n$. Across all architectures and input data sizes, the difference in performance between the different kernel block configurations of size (32, x) is minimal. Smaller first dimensions of the thread sizes, blocks of (16,16), do cause a decline in performance.

Chapter 4

Conclusions and perspectives

Tiled abstraction

A hardware-agnostic, type-agnostic, and size-agnostic tiled linear algebra framework in Julia was developed to increase the accessibility of HPC resources. The flexible abstraction layer simplifies the development and testing of new algorithms across diverse computing architectures and demonstrates Julia language's potential for becoming the cradle of similar abstractions that democratize HPC resources in the future. The framework also provides an alternative to redeveloping software for every architecture, by integrating into a composable ecosystem.Further work could continue expanding upon the existing framework by including batched QR or SVD algorithms as in [76], by including more general-purpose kernels for other dense linear algebra algorithms, or by including a new algorithm such as the LU decomposition.

Singular value decomposition

This work demonstrated the viability of the tiled dense linear algebra framework by implementing a tiled singular value decomposition that served for both GPU-only band-diagonalization of matrices and for out-of-core GPU/CPU-combined band-diagonalization. This is based on custom-implemented QR kernels that are platform-agnostic, and a LAPACK-based bulgechasing and diagonalization phase. Additionally, the extensibility of the framework was demonstrated by implementing

a tiled QR factorization. The out-of-core band-diagonalization now provides a Julia-based implementation that can process matrices larger than the GPU memory faster than the CPU. Further work could expand upon this and render the computation/communication-overlap vendor-agnostic and demonstrate cross-platform performance for the tiled algorithms (versus for the kernels only in this work), include automatic scheduling for optimizing the bulgechasing phase, and implement a tiled memory layout to optimize the data access speed.

Performance

Through the performance benchmarking of the tiled band-diagonalization, it was demonstrated that general abstraction frameworks do not hinder performance. However, the resulting flop count of the singular value decomposition algorithm grows unavoidably with a factor n^3, excluding the communication cost between partitions of large matrices. Currently, algorithms are being devised aiming to reduce memory movement and increase data locality, but no measure is available on how much impact both factors respectively have on the final performance. While it is generally believed communication between CPU and GPU is a bottleneck, during our benchmarking we found the decrease in performance due to CPU/GPU communication a factor that is independent of the matrix size; in other words, at large matrix sizes computation might still be the bottleneck. Comprehensive theories of performance that include not only flop count but also data movement could provide a more quantitative perspective on how to best optimize for performance at different data sizes.

Appendix A

Code fragments

A.1 QR kernel code example

```
@kernel function QR_unsafe_kernel2_2d!(input, input2, tau)
    i, j = @index(Local, NTuple)
    N = @uniform @groupsize()[1]

    # +1 to avoid bank conflicts on shared memory
    tile = @localmem eltype(input) (2N + 1, N)
    cache = @localmem eltype(input) (2N + 1)
    tau_iter = @localmem eltype(input) (1)
    corrvalue = @localmem eltype(input) (1)

    @inbounds tile[N+i, j] = input2[i, j]
    @inbounds tile[i, j] = input[i, j]

    @synchronize
    for iter in 1:N
        if (j==iter)
            cache[i] = tile[i+N, iter]^2
        end
        @synchronize
        if (i == 1) && (j==1)
            tmp_sum = zero(eltype(input))
            for l in 1:N
                tmp_sum += cache[l]
            end
            tmp_sum2 = sqrt(tmp_sum + tile[iter, iter]^2)
            newvalue = tile[iter, iter] + sign(tile[iter,iter]) *tmp_sum2
            tmp_sum2 = sqrt(tmp_sum + newvalue^2)
            tau_iter[1] = 2 * (newvalue / tmp_sum2)^2
```

```
29              corrvalue[1] = newvalue
30              tau[iter] = tau_iter[1]
31          end
32          tileiNiter= tile[i+N, iter]
33          tileiterj=tile[iter, j]
34          if (j>=iter)
35              tmp_sum = zero(eltype(input))
36              for l = N+1:2N
37                  tmp_sum += tile[l, iter] * tile[l, j]
38              end
39          end
40          @synchronize
41          taucorr=tau_iter[1] / corrvalue[1]
42          corrvalue1 = corrvalue[1]
43          if (j >= iter)
44              tmp_sum += corrvalue1 * tileiterj
45              if (i==iter)
46                  tile[i, j] = tile[i,j] - tmp_sum * taucorr
47              end
48              if (j>iter)
49                  tile[i+N, j] = tile[i+N, j] - tileiNiter *
50                                  tmp_sum *taucorr / corrvalue1
51              end
52          end
53          if (j==1)
54              tile[i+N, iter] = tileiNiter / corrvalue1
55          end
56          @synchronize
57      end
58
59      @inbounds input2[i, j] = tile[N+i, j]
60      @inbounds input[i, j] = tile[i, j]
61      @synchronize
62
63  end
```

Source Code A.1: A custom QR kernel written using KernelAbstractions.jl.

A.2 Partial bulge-chasing code

```
function block_bidiagonalize!(A, n, bandwidth, target_bandwidth)
    for j=1:target_bandwidth:n-target_bandwidth

        QR_row!(A,j, min(j+bandwidth+target_bandwidth-1, n) ,
                    target_bandwidth)

        for i=j:bandwidth:n
            #index of end of block and its neighbor
            lastindex=min(i+bandwidth+target_bandwidth-1, n)
            s_capped= min(bandwidth+target_bandwidth-1,
                        max(n-i-bandwidth-target_bandwidth+1,0))
            QR_col!(A,i+target_bandwidth,lastindex,s_capped)

            i+target_bandwidth>(n-bandwidth) && break
            QR_row!(A,i+target_bandwidth,lastindex+s_capped,bandwidth)
        end
    end
    return A
end

function QR_col!(A, startindex, lastindex, indexgap)
    qr!(view(A,startindex:lastindex, startindex:indexgap+lastindex))
    view(A,startindex:lastindex, startindex:indexgap+lastindex).=
        triu(view(A,startindex:lastindex, startindex:indexgap+lastindex))
    return;
end
```

Source Code A.2: Partial code for the bulge-chasing portion of the algorithm, making use of the standard lapack-refering *qr!* kernels in Julia and using views to avoid allocations. The *block_bidiagonalize!* function executes one sweep that reduces the bandwith to a target-bandwith, both are required to be powers of two.

A.3 Partial GeneralTiledMatrix definition code

```
struct TiledMatrix
    TileViews::Array{SubArray} #pointers to the tiles
    Tau::Array{CuArray}
end

struct LargeTiledMatrix
    A::AbstractArray #CPU-located source matrix
    Rows::Array{TileRow}
end

struct TileRow
    RowTile::AbstractArray #GPU-located row matrices
    Tau::CuArray
end

GeneralTiledMatrix=Union{TiledMatrix, LargeTiledMatrix}

function TiledMatrix(A::AbstractArray{T, 2}, blocksize::Int) where T

    no_tiles=Int(size(A,1)/blocksize)
    TileViews=Array{SubArray}(undef, no_tiles,no_tiles)

    for i in 1:no_tiles
        for j in 1:no_tiles
            TileViews[i,j]=view(A, (i-1)*blocksize.+(1:blocksize),
                                   (j-1)*blocksize.+(1:blocksize))
        end
    end

    backend=KernelAbstractions.get_backend(A)
    Tau=Array{CuArray}(undef, no_tiles,no_tiles)
    for i in 1:no_tiles
        for j in 1:no_tiles
            Tau[i,j]=KernelAbstractions.zeros(backend, T, blocksize)
        end
    end

    return TiledMatrix(TileViews, Tau)

end

```

```
42 function LargeTiledMatrix(A::Matrix{T}, backend, blocksize::Int) where T
43 
44     matrixsize=size(A,1)
45     no_tiles=Int(matrixsize/blocksize)
46 
47     rows=[]
48     Tau= KernelAbstractions.zeros(backend, T, blocksize)
49     Row=KernelAbstractions.zeros(backend, T, blocksize, matrixsize)
50     copyto!(Row,copy(view(A,1:blocksize,1:matrixsize)))
51     push!(rows, TileRow(true, 1, 1, Row, Tau))
52 
53     for i in 1:3
54         Tau= KernelAbstractions.zeros(backend, T, blocksize)
55         Row=KernelAbstractions.zeros(backend, T,
56                                         blocksize, matrixsize)
57         push!(rows, TileRow(true, 1, 1, Row, Tau))
58     end
59 
60     return LargeTiledMatrix(A,  rows)
61 
62 end
```

Source Code A.3: Definition of *GeneralTiledMatrix*, *LargeTiledMatrix* and *TiledMatrix* data types for the singular value decomposition, shortened from the original to display only the important parameters. The *TiledMatrix type* is designed for the GPU-only tiled band-diagonalization containing a reference to the tiled views of the GPU-located matrix, and the *LargeTiledMatrix* type is desgined for the out-of-core band-diagonalization and contains 4 GPU matrix rows used to circulate the data in and out of the GPU memory.

References

[1] A. Reuther et al. “Interactive supercomputing on 40,000 cores for machine learning and data analysis”. In: *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE. 2018, pp. 1–6.

[2] Statista. *Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2020, with forecasts from 2021 to 2025*. Statista chart. Sept. 2022. URL: https://www.statista.com/statistics/871513/worldwide-data-created/ (visited on 10/19/2022).

[3] Google. *The Size and Quality of a Data Set*. Google Developers Foundational courses in Machine Learning. July 2022. URL: https://developers.google.com/machine-learning/data-prep/construct/collect/data-size-quality (visited on 10/30/2022).

[4] N. Halko, P. G. Martinsson, and J. A. Tropp. “Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions”. In: *SIAM Review* 53.2 (2011), pp. 217–288. DOI: 10.1137/090771806.

[5] C. Musco and C. Musco. “Randomized Block Krylov Methods for Stronger and Faster Approximate Singular Value Decomposition”. In: *Advances in Neural Information Processing Systems*. Ed. by C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett. Vol. 28. Curran Associates, Inc., 2015.

[6] J. Shalf. “The Future of Computing beyond Moore’s Law”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 378.2166 (2020), p. 20190061. DOI: 10.1098/rsta.2019.0061. eprint: https://royalsocietypublishing.org/doi/pdf/10.1098/rsta.2019.0061. URL: https://royalsocietypublishing.org/doi/abs/10.1098/rsta.2019.0061.

[7] D. Reed, D. Gannon, and J. Dongarra. *Reinventing High Performance Computing: Challenges and Opportunities*. 2022. arXiv: 2203.02544 [cs.DC]. URL: https://arxiv.org/abs/2203.02544.

[8] S. H. Fuller and L. I. Millett. “Computing Performance: Game over or next Level?” In: *Computer* 44.1 (2011), pp. 31–38. DOI: 10.1109/MC.2011.15.

[9] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart. *LINPACK users’ guide*. SIAM, 1979.

[10] J. Demmel. “LAPACK: A Portable Linear Algebra Library for High-Performance Computers”. In: *Concurrency: Practice and Experience* 3.6 (1991), pp. 655–666. DOI: 10.1002/cpe.4330030610. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4330030610. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4330030610.

[11] J. Choi, J. Dongarra, R. Pozo, and D. Walker. "ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers". In: *The Fourth Symposium on the Frontiers of Massively Parallel Computation*. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 1992, pp. 120, 121, 122, 123, 124, 125, 126, 127. DOI: 10.1109/FMPC.1992.234898. URL: https://doi.ieeecomputersociety.org/10.1109/FMPC.1992.234898.

[12] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. "Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA Projects". In: *Journal of Physics: Conference Series* 180.1 (July 2009), p. 012037. DOI: 10.1088/1742-6596/180/1/012037. URL: https://dx.doi.org/10.1088/1742-6596/180/1/012037.

[13] M. Gates, J. Kurzak, A. Charara, A. YarKhan, and J. Dongarra. "SLATE: Design of a Modern Distributed and Accelerated Linear Algebra Library". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Sc '19. New York, NY, USA: Association for Computing Machinery, 2019. ISBN: 978-1-4503-6229-0. DOI: 10.1145/3295500.3356223. URL: https://doi.org/10.1145/3295500.3356223.

[14] F. Wang, C.-Q. Yang, Y.-F. Du, J. Chen, H.-Z. Yi, and W.-X. Xu. "Optimizing Linpack Benchmark on GPU-Accelerated Petascale Supercomputer". In: *Journal of Computer Science and Technology* 26.5 (Sept. 1, 2011), pp. 854–865. ISSN: 1860-4749. DOI: 10.1007/s11390-011-0184-1. URL: https://doi.org/10.1007/s11390-011-0184-1.

[15] N. Ashton. *EP8 - Prof Jack Dongarra - High Performance Computing (HPC) Pioneer*. Youtube, 2024. URL: https://www.youtube.com/watch?v=DgQt6rktdzw.

[16] M. W. Mahoney. "Randomized Algorithms for Matrices and Data". In: *Foundations and Trends in Machine Learning* 3.2 (2011), pp. 123–224. ISSN: 1935-8237. DOI: 10.1561/2200000035.

[17] Z. Chen. "Singular Value Decomposition and Its Applications in Image Processing". In: *Proceedings of the 2018 1st International Conference on Mathematics and Statistics*. ICoMS '18. Porto, Portugal: Association for Computing Machinery, 2018, pp. 16–22. ISBN: 9781450365383. DOI: 10.1145/3274250.3274261.

[18] L. Y. Ji H. *GPU acceralated randomized singular value decomposition and its application in image compression*. 2014.

[19] R. A. Sadek. "SVD Based Image Processing Applications: State of The Art, Contributions and Research Challenges". In: *International Journal of Advanced Computer Science and Applications* 3.7 (2012).

[20] J. A. MISZCZAK. "SINGULAR VALUE DECOMPOSITION AND MATRIX REORDERINGS IN QUANTUM INFORMATION THEORY". In: *International Journal of Modern Physics C* 22.09 (2011), pp. 897–918. DOI: 10.1142/S0129183111016683.

[21] C. D. Martin and M. A. Porter. "The Extraordinary SVD". In: *The American Mathematical Monthly* 119.10 (2012), pp. 838–851. DOI: 10.4169/amer.math.monthly.119.10.838.

[22] D. Kalman. "A Singularly Valuable Decomposition: The SVD of a Matrix". In: *The College Mathematics Journal* 27.1 (1996), pp. 2–23. DOI: 10.1080/07468342.1996.11973744.

[23] G. Strang. *Introduction to linear algebra*. SIAM, 2022.

[24] S. Axler. *Linear algebra done right*. Springer, 2015.

[25] M. W. Berry, D. Mezher, B. Philippe, and A. Sameh. “Parallel algorithms for the singular value decomposition”. In: *Handbook of parallel computing and statistics*. Chapman and Hall/CRC, 2005, pp. 133–180.

[26] M. Faverge, J. Langou, Y. Robert, and J. Dongarra. “Bidiagonalization and R-Bidiagonalization: Parallel Tiled Algorithms, Critical Paths and Distributed-Memory Implementation”. In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2017, pp. 668–677. DOI: 10.1109/IPDPS.2017.46.

[27] J. Xiao, Q. Xue, H. Ma, X. Zhang, and G. Tan. “A W-cycle Algorithm for Efficient Batched SVD on GPUs”. In: *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 465–466. ISBN: 978-1-4503-9204-4. DOI: 10.1145/3503221.3508443. URL: https://doi.org/10.1145/3503221.3508443.

[28] M. Gates, S. Tomov, and J. Dongarra. “Accelerating the SVD two stage bidiagonal reduction and divide and conquer using GPUs”. In: *Parallel Computing* 74 (2018). Parallel Matrix Algorithms and Applications (PMAA’16), pp. 3–18. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2017.10.004.

[29] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. “A class of parallel tiled linear algebra algorithms for multicore architectures”. In: *Parallel Computing* 35.1 (2009), pp. 38–53. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2008.10.002. URL: https://www.sciencedirect.com/science/article/pii/S0167819108001117.

[30] I. Boureima, M. Bhattarai, M. E. Eren, N. Solovyev, H. Djidjev, and B. S. Alexandrov. “Distributed Out-of-Memory SVD on CPU/GPU Architectures”. In: *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. 2022, pp. 1–8. DOI: 10.1109/HPEC55821.2022.9926288.

[31] K. Kabir, A. Haidar, S. Tomov, A. Bouteiller, and J. Dongarra. “A Framework for Out of Memory SVD Algorithms”. In: *High Performance Computing*. Ed. by J. M. Kunkel, R. Yokota, P. Balaji, and D. Keyes. Cham: Springer International Publishing, 2017, pp. 158–178. ISBN: 978-3-319-58667-0.

[32] A. Haidar, H. Ltaief, P. Luszczek, and J. Dongarra. “A Comprehensive Study of Task Coalescing for Selecting Parallelism Granularity in a Two-Stage Bidiagonal Reduction”. In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. 2012, pp. 25–35. DOI: 10.1109/IPDPS.2012.13.

[33] A. YarKhan, J. Kurzak, and J. Dongarra. *QUARK Users’ Guide: QUeueing and Runtime for Kernels*. ICL-UT-11-02. –2011.

[34] S. Catalán, J. R. Herrero, F. D. Igual, E. S. Quintana-Ortí, and R. Rodríguez-Sánchez. “Fine-Grain Task-Parallel Algorithms for Matrix Factorizations and Inversion on Many-Threaded CPUs”. In: *Concurrency and Computation: Practice and Experience* 35.27 (2023), e6999. DOI: 10.1002/cpe.6999. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.6999. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6999.

[35] R. Murray et al. *Randomized Numerical Linear Algebra : A Perspective on the Field With an Eye to Software*. 2023. arXiv: 2302.11474 [math.NA].

[36] P.-G. Martinsson and J. A. Tropp. “Randomized numerical linear algebra: Foundations and algorithms”. In: *Acta Numerica* 29 (2020), pp. 403–572. DOI: 10.1017/S0962492920000021.

[37] N. K. Kumar and J. Schneider. “Literature survey on low rank approximation of matrices”. In: *Linear and Multilinear Algebra* 65.11 (2017), pp. 2212–2244. DOI: 10.1080/03081087.2016.1267104.

[38] Ł. Struski, P. Morkisz, P. Spurek, S. R. Bernabeu, and T. Trzciński. *Efficient GPU implementation of randomized SVD and its applications*. 2021. DOI: 10.48550/ARXIV.2110.03423.

[39] C. Iyer, A. Gittens, C. Carothers, and P. Drineas. “Iterative Randomized Algorithms for Low Rank Approximation of Tera-Scale Matrices with Small Spectral Gaps”. In: *2018 IEEE/ACM 9th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (scalA)*. 2018, pp. 33–40. DOI: 10.1109/ScalA.2018.00008.

[40] C. Musco and C. Musco. “Randomized Block Krylov Methods for Stronger and Faster Approximate Singular Value Decomposition”. In: *Advances in Neural Information Processing Systems*. Ed. by C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett. Vol. 28. Curran Associates, Inc., 2015. URL: https://proceedings.neurips.cc/paper_files/paper/2015/file/1efa39bcaec6f3900149160693694536-Paper.pdf.

[41] A. Bentbib, A. Kanber, and K. Lachhab. “Subspace Iteration Method for Generalized Singular Values”. In: (June 2019).

[42] Z. Li, Q. Fang, and G. Ballard. “Parallel Tucker Decomposition with Numerically Accurate SVD”. In: *Proceedings of the 50th International Conference on Parallel Processing*. Icpp ’21. New York, NY, USA: Association for Computing Machinery, 2021. ISBN: 978-1-4503-9068-2. DOI: 10.1145/3472456.3472472. URL: https://doi.org/10.1145/3472456.3472472.

[43] D. Keyes, H. Ltaief, Y. Nakatsukasa, and D. Sukkari. “High-Performance SVD Partial Spectrum Computation”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Sc ’23. New York, NY, USA: Association for Computing Machinery, 2023. ISBN: 9798400701092. DOI: 10.1145/3581784.3607109. URL: https://doi.org/10.1145/3581784.3607109.

[44] S. Li, J. Liu, and Y. Du. “A High Performance Implementation of Zolo-SVD Algorithm on Distributed Memory Systems”. In: *Parallel Computing* 86 (2019), pp. 57–65. ISSN: 0167-8191. DOI: 10.1016/j.parco.2019.04.004. URL: https://www.sciencedirect.com/science/article/pii/S0167819118301807.

[45] M. Onuki and Y. Tanaka. “SVD for Very Large Matrices: An Approach with Polar Decomposition and Polynomial Approximation”. In: *2018 IEEE International Conference on Data Mining Workshops (ICDMW)*. 2018, pp. 954–963. DOI: 10.1109/ICDMW.2018.00138.

[46] C. Musco, P. Netrapalli, A. Sidford, S. Ubaru, and D. P. Woodruff. “Spectrum Approximation Beyond Fast Matrix Multiplication: Algorithms and Hardness”. In: *CoRR* abs/1704.04163 (2017). arXiv: 1704.04163. URL: http://arxiv.org/abs/1704.04163.

[47] I. S. Dhillon and B. N. Parlett. “Multiple Representations to Compute Orthogonal Eigenvectors of Symmetric Tridiagonal Matrices”. In: *Linear Algebra and its Applications* 387 (2004), pp. 1–28. ISSN: 0024-3795. DOI: 10.1016/j.laa.2003.12.028. URL: https://www.sciencedirect.com/science/article/pii/S002437950300908X.

[48] M. Aharon, M. Elad, and A. Bruckstein. "K-SVD: An Algorithm for Designing Overcomplete Dictionaries for Sparse Representation". In: *IEEE Transactions on Signal Processing* 54.11 (2006), pp. 4311–4322. DOI: 10.1109/TSP.2006.881199.

[49] Z. Fang, F. Qi, Y. Dong, Y. Zhang, and S. Feng. "Parallel Tensor Decomposition with Distributed Memory Based on Hierarchical Singular Value Decomposition". In: *Concurrency and Computation: Practice and Experience* 35.17 (2023), e6656. DOI: 10.1002/cpe.6656. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.6656. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6656.

[50] C. Deng, M. Yin, X.-Y. Liu, X. Wang, and B. Yuan. "High-Performance Hardware Architecture for Tensor Singular Value Decomposition: Invited Paper". In: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2019, pp. 1–6. DOI: 10.1109/ICCAD45719.2019.8942082.

[51] W. Luo, R. Fan, Z. Li, D. Du, Q. Wang, and X. Chu. *Benchmarking and Dissecting the Nvidia Hopper GPU Architecture*. 2024. arXiv: 2402.13499 [cs.AR]. URL: https://arxiv.org/abs/2402.13499.

[52] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. "Demystifying GPU microarchitecture through microbenchmarking". In: *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*. 2010, pp. 235–246. DOI: 10.1109/ISPASS.2010.5452013.

[53] E. D'Azevedo and J. Dongarra. "The Design and Implementation of the Parallel Out-of-Core ScaLAPACK LU, QR, and Cholesky Factorization Routines". In: *Concurrency: Practice and Experience* 12.15 (Dec. 25, 2000), pp. 1481–1493. ISSN: 1040-3108. DOI: 10.1002/1096-9128(20001225)12:15<1481::AID-CPE540>3.0.CO;2-V. URL: https://doi.org/10.1002/1096-9128(20001225)12:15%3C1481::AID-CPE540%3E3.0.CO;2-V (visited on 06/15/2024).

[54] NVIDIA. *https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf*. Version 12.6. NVIDIA Corporation. United States, 2024. URL: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[55] *The GPU hardware and software ecosystem*. EuroCC National Competence Center Sweden, 2024. URL: https://enccs.github.io/gpu-programming/2-gpu-ecosystem/.

[56] S. An and S. Seo. "Efficient Parallel Implementations of LWE-Based Post-Quantum Cryptosystems on Graphics Processing Units". In: *Mathematics* 8 (Oct. 2020), p. 1781. DOI: 10.3390/math8101781.

[57] X. Wang, R. Pinkham, M. A. Zidan, F.-H. Meng, M. P. Flynn, Z. Zhang, and W. D. Lu. "TAICHI: A Tiled Architecture for in-Memory Computing and Heterogeneous Integration". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 69.2 (2022), pp. 559–563. DOI: 10.1109/TCSII.2021.3097035.

[58] N. R. Miniskar, M. A. H. Monil, P. Valero-Lara, F. Liu, and J. S. Vetter. "Tiling Framework for Heterogeneous Computing of Matrix Based Tiled Algorithms". In: *Proceedings of the 2nd International Workshop on Extreme Heterogeneity Solutions*. ExHET 23: New York, NY, USA: Association for Computing Machinery, 2023. ISBN: 9798400701429. DOI: 10.1145/3587278.3595642. URL: https://doi.org/10.1145/3587278.3595642.

[59] F. D. Igual, E. Chan, E. S. Quintana-Ortí, G. Quintana-Ortí, R. A. van de Geijn, and F. G. Van Zee. “The FLAME Approach: From Dense Linear Algebra Algorithms to High-Performance Multi-Accelerator Implementations”. In: *Journal of Parallel and Distributed Computing* 72.9 (2012), pp. 1134–1143. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2011.10.014. URL: https://www.sciencedirect.com/science/article/pii/S0743731511002139.

[60] F. G. Van Zee and R. A. van de Geijn. “BLIS: A Framework for Rapidly Instantiating BLAS Functionality”. In: *ACM Trans. Math. Softw.* 41.3 (June 2015). ISSN: 0098-3500. DOI: 10.1145/2764454. URL: https://doi.org/10.1145/2764454.

[61] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. “StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures”. In: *Euro-Par 2009 Parallel Processing*. Ed. by H. Sips, D. Epema, and H.-X. Lin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 863–874. ISBN: 978-3-642-03869-3.

[62] G. Bosilca et al. “Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA”. In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. 2011, pp. 1432–1441. DOI: 10.1109/IPDPS.2011.299.

[63] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza. “Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking”. In: *CoRR* abs/1804.06826 (2018). arXiv: 1804.06826. URL: http://arxiv.org/abs/1804.06826.

[64] J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and I. Yamazaki. “The Singular Value Decomposition: Anatomy of Optimizing an Algorithm for Extreme Scale”. In: *SIAM Review* 60.4 (2018), pp. 808–865. DOI: 10.1137/17M1117732. eprint: https://doi.org/10.1137/17M1117732. URL: https://doi.org/10.1137/17M1117732.

[65] J. Bezanson, J. Chen, B. Chung, S. Karpinski, and V. B. e. a. Shah. “Julia: Dynamism and Performance Reconciled by Design”. In: *Proc. ACM Program. Lang.* (Oct. 2018). DOI: 10.1145/3276490.

[66] M. Giordano, M. Klöwer, and V. Churavy. “Productivity Meets Performance: Julia on A64FX”. In: *2022 IEEE International Conference on Cluster Computing*. IEEE. 2022, pp. 549–555.

[67] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek. “Achieving numerical accuracy and high performance using recursive tile LU factorization with partial pivoting”. In: *Concurrency and Computation: Practice and Experience* 26.7 (2014), pp. 1408–1431. DOI: https://doi.org/10.1002/cpe.3110.

[68] U. Utkarsh et al. “Automated translation and accelerated solving of differential equations on multiple GPU platforms”. In: *Computer Methods in Applied Mechanics and Engineering* 419 (2024), p. 116591. ISSN: 0045-7825. DOI: https://doi.org/10.1016/j.cma.2023.116591.

[69] V. Churavy. *KernelAbstractions.jl*. https://github.com/JuliaGPU/KernelAbstractions.jl. 2023.

[70] T. B. Simeon Danisch. *GPUArrays.jl*. https://github.com/JuliaGPU/GPUArrays.jl. 2023.

[71] V. Churavy et al. *Bridging HPC Communities through the Julia Programming Language*. 2022. arXiv: 2211.02740 [cs.DC].

[72] W. F. Godoy, P. Valero-Lara, T. E. Dettling, C. Trefftz, I. Jorquera, T. Sheehy, R. G. Miller, M. Gonzalez-Tallada, J. S. Vetter, and V. Churavy. “Evaluating performance and portability of high-level programming models: Julia, Python/Numba, and Kokkos on exascale nodes”. In: *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, May 2023. DOI: 10.1109/ipdpsw59300.2023.00068. URL: http://dx.doi.org/10.1109/IPDPSW59300.2023.00068.

[73] T. Faingnaert, T. Besard, and B. De Sutter. “Flexible Performant GEMM Kernels on Gpus”. In: *IEEE Transactions on Parallel and Distributed Systems* 33.9 (2022), pp. 2230–2248. DOI: 10.1109/TPDS.2021.3136457.

[74] J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and I. Yamazaki. “The Singular Value Decomposition: Anatomy of Optimizing an Algorithm for Extreme Scale”. In: *SIAM Review* 60.4 (2018), pp. 808–865. DOI: 10.1137/17M1117732.

[75] N.-M. Ho and W.-F. Wong. “Exploiting half precision arithmetic in Nvidia GPUs”. In: *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. 2017, pp. 1–7. DOI: 10.1109/HPEC.2017.8091072.

[76] W. H. Boukaram, G. Turkiyyah, H. Ltaief, and D. E. Keyes. “Batched QR and SVD Algorithms on GPUs with Applications in Hierarchical Matrix Compression”. In: *Parallel Computing* 74 (2018), pp. 19–33. ISSN: 0167-8191. DOI: 10.1016/j.parco.2017.09.001. URL: https://www.sciencedirect.com/science/article/pii/S0167819117301461.