HALMSTAD UNIVERSITY

Computer Science Engineering 300 hp

MASTER THESIS

Acceleration of Massive MIMO algorithms for Beyond 5G Baseband processing

Ellen Nihl, Eek de Bruijckere

Halmstad University, June 19, 2023–version 1.0

ABSTRACT

As the world becomes more globalised, user equipment such as smartphones and Internet of Things devices require increasingly more data, which increases the demand for wireless data traffic. Hence, the acceleration of next-generational networks (5G and beyond) focuses mainly on increasing the bitrate and decreasing the latency. A crucial technology for 5G and beyond is the massive MIMO. In a massive MIMO system, a decoder processes the received signals from multiple antennas to decode the transmitted data and extract useful information. This has been implemented in many ways, and one of the most used algorithms is the Zero Forcing (ZF) algorithm. This thesis presents a novel parallel design to accelerate the ZF algorithm using the Cholesky decomposition. This is implemented on a GPU, written in the CUDA programming language, and compared to the existing state-of-the-art implementations regarding latency and throughput. The implementation is also validated from a MATLAB implementation. This research demonstrates promising performance using GPUs for massive MIMO detection algorithms. Our approach achieves a significant speedup factor of 350 in comparison to a serial version of the implementation. The throughput achieved is 160 times greater than a comparable GPU-based approach. Despite this, our approach reaches a 2.4 times lower throughput than a solution that employed application-specific hardware. Given the promising results, we advocate for continued research in this area to further optimise detection algorithms and enhance their performance on GPUs, to potentially achieve even higher throughput and lower latency.

ACKNOWLEDGEMENTS

We would like to sincerely thank our supervisors, Hazem Ali and Ali Nada, for their valuable guidance, support, and mentorship throughout our master's thesis. Their expertise and constructive feedback have greatly influenced the direction and quality of this research. We are grateful for their belief in our abilities and the opportunities they provided for our personal and academic growth.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

MIMO Multiple Input Multiple Output

IoT Internet of Things

BS Base Station

ZF Zero Forcing

GPU Graphical Processing Unit

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

MMSE Minimum Mean Squared Error

CSI Channel State Information

TDD Time Division Duplex

AMC Asymmetric Multi-Core processors

FPGA Field-Programmable-Gate Array

HPC High-Performance Computing Landscapes

ASIC Application Specific Integrated Circuit

ASIP Application-Specific Instruction set Processor

SIMD Single Instruction Multiple Data

EDP Explicit Data Parallelism

ILP Implicit Instruction-Level Parallelism

SM Streaming Multiprocessors

RVM Relevance Vector Machine

PUSCH Physical Uplink Shared Channel

UE User Equipment

MF Matched Filtering

VLIW Very Large Instruction Word

1 INTRODUCTION

As the world becomes progressively globalised, user equipment such as smartphones and Internet of Things (IoT) devices require increasingly more data, significantly increasing wireless data traffic. According to the Ericsson Mobility report 2022, the average monthly data consumption per smartphone will reach over 19 GB in 2023. Additionally, 5G mobile subscriptions are expected for 2028 to reach 5 billion [1].

Hence, the progress of evolving wireless network generations (*5G and beyond*) aims to increase mainly the bitrate and throughput, lower the latency and increase energy and spectral efficiency to account for the current rapid developments in mobile usage [2].

Massive Multiple Input Multiple Output (MIMO) is considered a crucial technology for the upcoming 5G wireless systems, as it enables the base station to serve multiple users simultaneously while using a large number of antennas. This enhances spectral efficiency and link reliability in comparison to small-scale MIMO systems. However, the data detection process in massive MIMO involves high baseband processing complexity, which makes optimal data detection methods impractical. Linear detection algorithms have gained attention due to their simplicity and ability to perform under certain conditions. One such algorithm is the Zero Forcing algorithm. ZF is a linear detection technique that eliminates inter-user interference by multiplying the received signal with the inverse of the channel matrix. Although it can perform well in scenarios with a large number of antennas and a small number of users, it suffers from noise amplification and error propagation in scenarios with a large number of users [3, 4].

The main goal of this thesis is to enhance the throughput for massive MIMOs. To achieve this, we investigate the use of GPUs for the ZF algorithm to exploit possible parallelism, as GPUs are optimised for high throughput and specialised for handling matrix calculations, given they are vector processors. The increase in antenna and user count for massive MIMOs leads to even larger matrices to consider, increasing the computational complexity, which further validates our choice of GPUs. The experimental results show a significant speedup factor of 350 of our GPU implementation compared to a serial version of the implementation. Furthermore, compared to [4], which also had a GPU-based approach, our implementation achieves a 160 times greater throughput. However, compared to [5]'s application-specific approach, we get a 2.4 lower throughput.

This thesis is organised as follows, Chapter 2 presents the background and contains the knowledge needed to understand the problem and the tools used to solve it. Chapter 3 is the literature review which contains a comprehensive summary and analysis of the existing research in the field of massive MIMO systems and decomposition algorithms. Chapter 4 presents the proposed ZF algorithm acceleration problem solution. Chapter 5 presents this work's results, discussion and conclusions.

2

BACKGROUND

This chapter aims to provide the necessary knowledge for understanding the problem addressed in this thesis. Matrices are represented by uppercase bold letters, while vectors are denoted by lowercase bold letters. The row of a matrix is indicated by i, and the column by j, and the element at row i and column j of matrix $\mathbf{A}$ is denoted by a_{ij}. The inverse of a matrix $\mathbf{A}$ is denoted by $\mathbf{A}^{-1}$.

This chapter starts with Section 2.1 which presents a description of massive MIMOs. Next, Section 2.2 explains the Cholesky decomposition algorithm's equations and limitations. Section 2.3 explains the architecture of a GPU and how the CUDA programming model works. Finally, Section 2.4 explains the problem that is being addressed in this thesis.

2.1 MASSIVE MIMOS

Massive MIMO is a crucial technology for next-generational wireless networks. It is an extension of conventional MIMO, a technology for data transmission using multiple antennas at the Base Station (BS) that serve multiple users, increasing the bitrate and throughput, simultaneously. Massive MIMO exploits an even larger amount of antennas at the BS to serve many users, achieving greater spectral and energy efficiency [6] and easing the baseband processing for the User Equipment (UE) [7]. As the number of antennas grows, the greater the channels' complexity becomes [5, 3]; as seen in Figure 1, the number of antennas N and amount of users K create an $N \times K$ channel matrix $\mathbf{H}$ (Equation 1).

$$\mathbf{H} = \begin{bmatrix} h_{11} & h_{12} & \cdots & h_{1j} \\ h_{21} & h_{22} & \cdots & h_{2j} \\ \vdots & \vdots & \ddots & \vdots \\ h_{i1} & h_{i2} & \cdots & h_{ij} \end{bmatrix} \tag{1}$$

The channel matrix $\mathbf{H} \in \mathbb{C}^{N \times K}$ contains the elements h_{ij} which is the channel gain or loss from every connection pair of the corresponding receive antenna i at the BS and transmit antenna j from the UE [8].

Massive MIMOs use spatial multiplexing which refers to the ability of the system to transmit multiple data streams simultaneously using the same frequency, which increases the data rate and link reliability.

The data streams are referred to as symbols. The symbols received by the antennas at the BS are represented by Equation 2, and the symbols transmitted by the UE is denoted as Equation 3. The relationship between the receive symbols and transmit symbols are seen in Equation 4.

$$\mathbf{y} = [y_1, y_2, \cdots, y_N]^T \tag{2}$$

$$\mathbf{x} = [x_1, x_2, \cdots, x_K]^T \tag{3}$$

$$\mathbf{y} = \mathbf{H}\mathbf{x} + \mathbf{n} \tag{4}$$

Where $\mathbf{H}$ is the channel matrix, $\mathbf{y}$ is the receive symbol vector, $\mathbf{x}$ is the transmit symbol vector, and $\mathbf{n}$ is the $N \times 1$ noise vector.

The complex channels affect the detection and precoding of up and down-link data processes on the BS by complex matrix manipulations, mainly matrix-matrix multiplications and matrix inversions. Current detection -and precoding methods in massive MIMOs are preferably linear because of their robustness and guaranteed performance. There are many linear detector algorithms, e.g. Zero Forcing (ZF) and Minimum Mean Squared Error (MMSE) equalisation. These algorithms require the calculation of the inverse channel matrix. Decomposing the channel matrix can speed up the processes immensely by performing decomposition algorithms such as Cholesky, QR and LDL decompositions [3, 9].

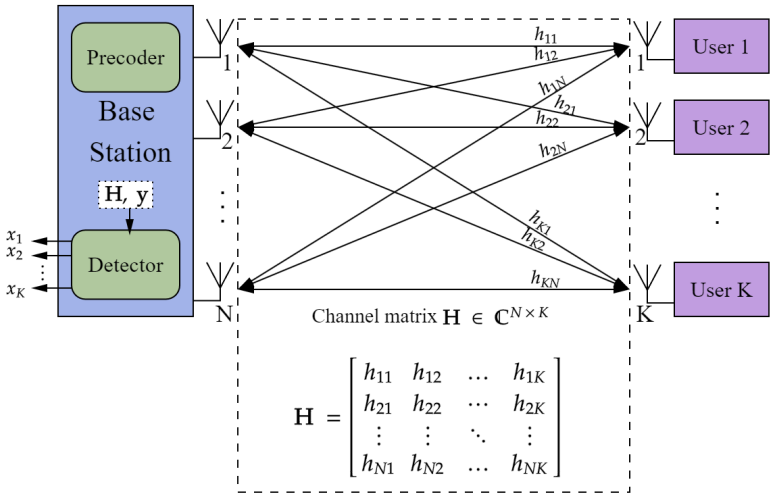

Figure 1: A simple overview of the massive MIMO architecture.

2.1.1 *Precoder/Detector*

The detector and precoder are essential components of the BS, where user data is transformed to optimise the transmission and reception of data over channels. The goal of the precoder is to enhance the transmission of the signal by utilising the Channel State Information (CSI) to shape the signal, thereby maximising the received signal strength and minimising interference between users. The goal of the detector is to optimise the received signals by using the same CSI, represented by the channel matrix. One way to obtain the CSI is by downlink piloting using Time Division Duplex (TDD) [10]. There are various methods to implement precoding and detection, this thesis focuses on linear detection. The two most popular linear detection methods are ZF, and linear MMSE equalisation [3]. ZF precoding is a linear technique with reasonable computational complexity that achieves full spatial multiplexing and multiuser diversity gains. ZF's ability to completely cancel multiuser interference makes it suitable for massive MIMO. However, its performance may only be optimal in noise-limited situations. The ZF algorithm is defined in Equation 5, where $\mathbf{x}$ and $\mathbf{y}$ are signal vectors, and $\mathbf{H}$ is the channel matrix. Here, $\mathbf{H}^H$ is the Hermitian transpose of $\mathbf{H}$.

$$\mathbf{x}_{ZF} = (\mathbf{H}^H\mathbf{H})^{-1}\mathbf{H}^H\mathbf{y} \tag{5}$$

The most computationally heavy part of the ZF algorithm is the inversion of the Gramian matrix $\mathbf{H}^H\mathbf{H}$. Decomposition of the matrix can speed up these calculations. Some commonly used decomposition algorithms are the QR decomposition, where Q is the orthogonal matrix and R is the upper triangle matrix, Cholesky decomposition, and LDL^T, where L is the lower triangular matrix and D is the diagonal matrix. These decompositions work so that when the matrices are multiplied, they result in the original matrix [11, 12, 7].

2.2 CHOLESKY DECOMPOSITION

Cholesky decomposition is a widely used matrix factorisation method that can be used to solve a variety of linear algebra problems, including those encountered when implementing linear detection algorithms such as ZF. Cholesky decomposition is a variant of LU decomposition. The LU (lower-upper) decomposition is a matrix factorisation technique that decomposes a matrix into a lower and upper triangular matrix. This decomposition is such that when the two matrices are multiplied, they result in the original matrix. But the Cholesky decomposition has the added property that the resulting lower triangular matrix is also symmetric. Given a positive definite matrix $\mathbf{A}$, Cholesky decomposition finds a lower triangular matrix $\mathbf{L}$ such that

multiplying the lower triangle matrix with its own transpose equals **A** (Equation 6). There are two expressions describing how to calculate the **L** matrix. For the diagonal elements Equation 7 is used and, for non-diagonal elements Equation 8 is used [13].

$$\mathbf{A} = \mathbf{L}\mathbf{L}^{H} \tag{6}$$

$$l_{i,i} = \sqrt{a_{i,i} - \sum_{k=1}^{j-1} l_{i,k}^2} \tag{7}$$

$$l_{i,j} = \frac{1}{l_{j,j}}(a_{i,j} - \sum_{k=1}^{j-1} l_{i,k} l_{j,k}) \tag{8}$$

The Cholesky decomposition algorithm can be parallelised, but it requires careful consideration of its dependencies. In order to calculate the diagonal elements (as shown in Equation 7), all elements in the row must have already been calculated. Similarly, to calculate the non-diagonal elements (as shown in Equation 8), the diagonal element in the same column must be known. These dependencies must be taken into account when parallelising the algorithm. One possible solution to minimise the complexity of these dependencies is to use *Block Cholesky*, which is explained in Section 2.2.1. This approach can significantly enhance the parallelisation of the algorithm while minimising the impact of its dependencies.

In the context of linear detection algorithms, the Cholesky decomposition can be used to speed up the inversion of the Gramian matrix in the ZF algorithm. Instead of explicitly computing the inverse of the Gramian matrix, the Cholesky decomposition can be used to compute the inverse of the lower triangular matrix **L**, which can then be used to solve the linear system involving the Gramian matrix more efficiently. The inversion shown in Equation 9 is used where **L** is the lower triangle matrix and can be inverted with backwards-substitution [5].

$$\mathbf{A}^{-1} = (\mathbf{L}\mathbf{L}^{H})^{-1} = (\mathbf{L}^{H})^{-1}\mathbf{L}^{-1} \tag{9}$$

2.2.1 *Block Cholesky*

Block Cholesky is a commonly used parallelisation technique for the Cholesky algorithm [7, 14, 15]. It is an important part of this thesis; this section will describe how it works.

The input is a $K \times K$ symmetric positive definite matrix **A** that is to be divided as described in Figure 2. The calculation starts with

the diagonal element d_n (blue). The square root of the element is taken, and the result is put back into the matrix. The next step is calculating the column underneath the diagonal element $\mathbf{c}_n$ (purple). This is done by dividing the column vector with the diagonal element. The third step is to calculate the rest of the non-finished matrix $\mathbf{U}_n$ (orange). This is done by subtracting this matrix with the column matrix times itself transposed (Equation 10).

$$\mathbf{U}_n = \mathbf{U}_n - \mathbf{c}_n \times \mathbf{c}_n^H \tag{10}$$

When this is done, the first column of the lower triangle matrix is completed, and next, the rest of the matrix is prepared. The algorithm moves to this new lower triangle matrix, which is now completely independent from the previous column, and starts the algorithm over until the whole matrix is calculated. The pseudo-code for this algorithm is described in algorithm 1.

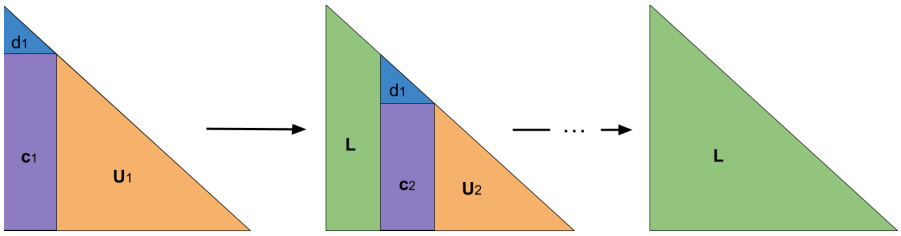

Figure 2: Explanation of block Cholesky. Blue are the diagonal elements. Purple are the column under the diagonal elements. Orange are the rest of the unfinished matrix. Green is the resulting **L** matrix

Algorithm 1 Block cholesky

Input: Matrix $\mathbf{A} = \mathbf{H}^H\mathbf{H}$
Out: Matrix **L** where $\mathbf{A} = \mathbf{L}\mathbf{L}^H$
n = Size(**A**)
1: **for** n to K **do**
2: $a_n = \sqrt{a_n}$
3: $\mathbf{c}_n = \mathbf{c}_n / a_n$
4: $\mathbf{U}_n = \mathbf{U}_n - \mathbf{c}_n \times \mathbf{c}_n^H$
5: **end for**

2.3 GPUS AND CUDA

This section goes through the general architecture of a Graphical Processing Unit (GPU) and presents the terminology used in this context. Then, some important functionalities of the Compute Unified Device Architecture (CUDA) programming model are also presented.

2.3.1 *GPU architecture*

In the post-Moore era, accelerating computer architectures have moved from solely focusing on the minimisation of transistors and increasing the clock speed of processors to integrating specialised processors into computer systems, for example, GPUs, Asymmetric Multi-Core processors (AMC) and Field-Programmable-Gate Array (FPGA). The focus has also shifted to simultaneously bettering the software performance of these systems.

To better understand the architecture of a GPU, it is necessary to compare it to a conventional Central Processing Unit (CPU). A CPU is general purpose and has optimised performance for serial execution. In contrast, GPUs are optimised for increasing the throughput of a system. GPUs started as special-purpose processors for graphical rendering for video games. Now, it has shifted to using GPUs for a more general purpose to accelerate computations in High-Performance Computing Landscapes (HPC). A CPU can exist of a smaller set of complicated processing cores, while a GPU consists of a massive amount of simpler processing cores. This enables the GPU to achieve *latency hiding*, which refers to when latency occurs from an instruction, another instruction can be processed during that same time. A CPU mainly exploits Implicit Instruction-Level Parallelism (ILP) to achieve parallelism, whereas a GPU mainly exploits Explicit Data Parallelism (EDP) [16]. Implicit ILP refers to a processor's ability to execute multiple instructions simultaneously within a thread, which happens without the programmer's help specifying how the computations are parallelised precisely, such as pipelining. Conversely, EDP refers to when the programmer explicitly divides the computations over cores, threads or GPUs to run concurrently.

Figure 3 presents a simple overview of a heterogeneous computing system consisting of a CPU and GPU, which are the structures of many NVIDIA models, such as the NVIDIA RTX A5000 [17] and NVIDIA RTX 6000 [18]. The CPU is referred to as the *host*, and the GPU is referred to as the *device*. The GPU consists of multiple Streaming Multiprocessors (SM) that are in charge of the execution of programs. Every SM contain a set of processing units called CUDA cores, and an on-chip Level 1 (L1) cache memory, which works as shared memory for the SM.

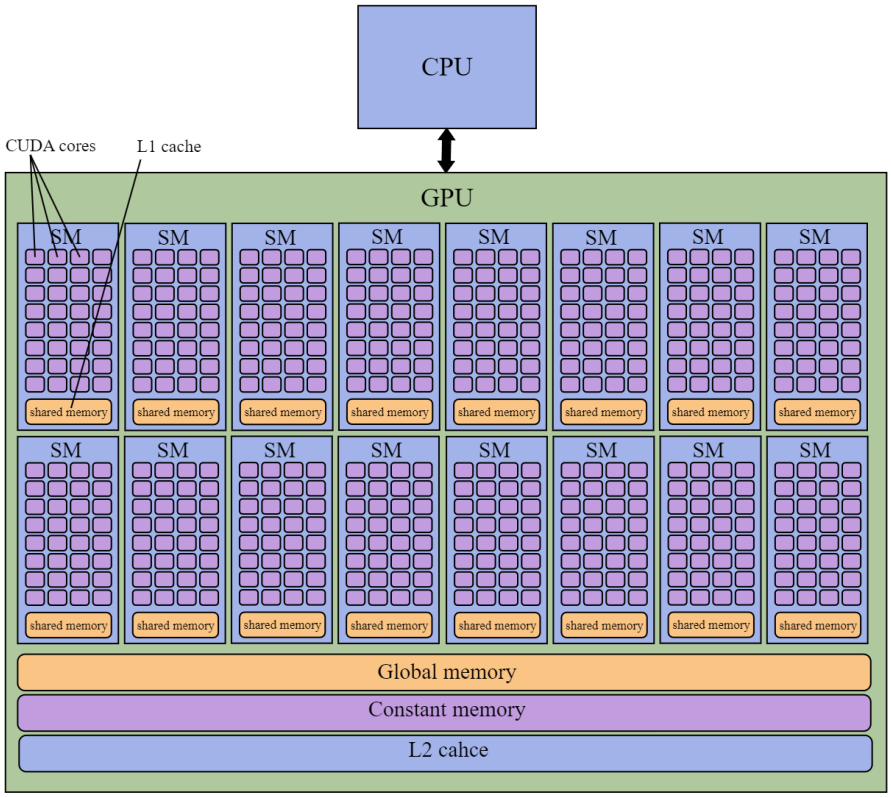

Figure 3: A simple overview of an NVIDIA heterogeneous computer architecture consisting of a CPU and GPU. SM stands for the streaming multiprocessors.

Many types of memory are used in architectures like these. Global memory can be read and written by both the host and the device and is accessible by every SM, but it has the downside of being very slow to access. So the off-chip Level 2 (L2) cache can be used to provide faster access to global memory. Constant memory can also be accessed by all SMs but is read-only [19].

2.3.2 *CUDA programming model*

This paragraph explains the basic terminology associated with the CUDA programming model. Figure 4 presents an overview of the CUDA programming model, which is used to program NVIDIA GPUs. The code running on the CPU is called host code, and it can launch multiple CUDA kernels. In the context of GPUs, a kernel is a function that can be executed in parallel by many threads. The primary purpose is to have every thread in a CUDA kernel execute the same code but with a unique set of data, which enables parallelism. A *thread* is an execution unit which can be mapped to a CUDA core [20]. A *block* is a set of threads on the same SM, meaning that the threads share their resources if they are in the same SM and can communicate through the shared memory. A *grid* consists of several blocks, and each ker-

nel is organised by one grid in the device. When a CUDA kernel is launched to a grid, it is split into coarse subproblems divided into multiple blocks, then split further into smaller pieces that the threads can process in the blocks. Threads in a block are put together in sets of 32 threads, and these sets are referred to as *warps* [21, 20, 19].

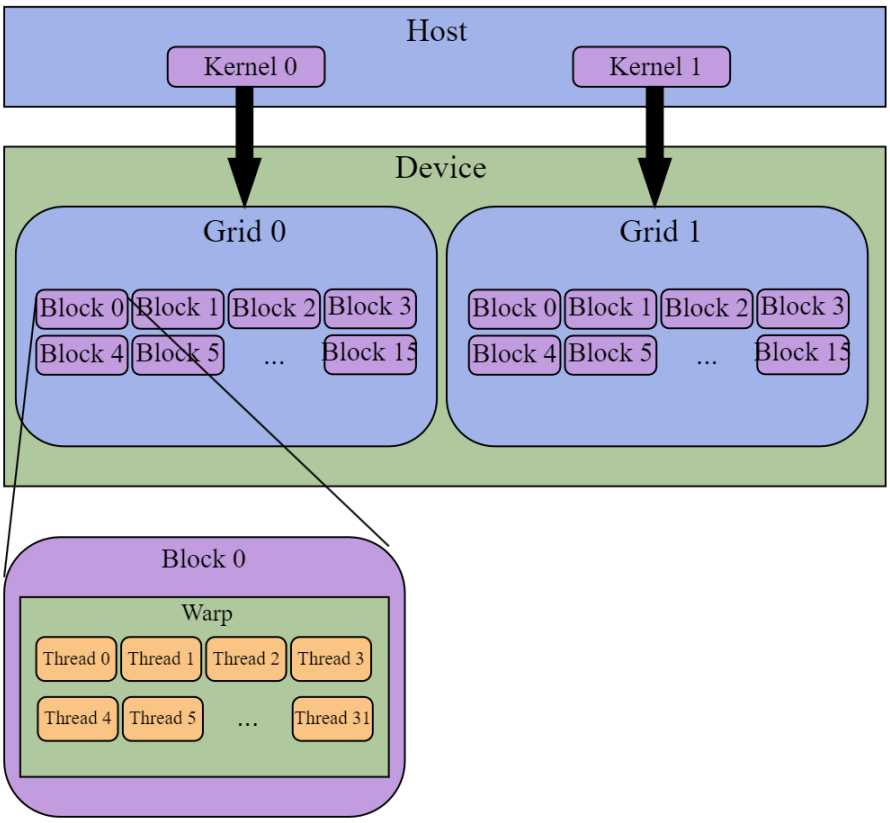

Figure 4: A simple overview of the CUDA programming model.

2.4 PROBLEM FORMULATION

Baseband processing is a main component of wireless communication systems, which involves processing signals at the base station or mobile device before transmission or reception. With the growing demand for higher data rates, increased reliability, and lower latency, there is a need to accelerate baseband processing in wireless communication systems [1]. Massive MIMO is a wireless communication technology that uses a large number of antennas at both the transmitter and receiver to increase the spectral efficiency and energy efficiency of wireless networks. The technology is particularly suited for 5G networks and beyond, as it can provide high data rates, improved coverage, and reduced interference. Massive MIMO systems use hundreds or even thousands of antennas to serve multiple users simultaneously, which increases the computational requirements for baseband processing. The essential factor contributing to this compu-

tational complexity is the user-antenna ratio, which should be within a 1:5 to 1:10 range to maintain a correct reception of the signal. Otherwise, the error increases. Because of this ratio requirement, the sizes of the matrices increase drastically, hence the number of calculations too.

This thesis investigates the acceleration of the ZF signal processing detection algorithm commonly used in massive MIMOs. The ZF algorithm separates the signals transmitted by multiple antennas to different users while minimising interference between the users. This is achieved by inverting the channel matrix, which represents the relationship between the transmitted and received signals. However, the inversion of the channel matrix can be computationally expensive, especially in large-scale communication systems with thousands of antennas and users. The inversion of the channel matrix can be done by using matrix decomposition algorithms such as QR or Cholesky decomposition algorithms. In this thesis, we selected the Cholesky algorithm to compute the channel matrix inversion by decomposing it into a lower triangular matrix and its Hermitian transpose. This decomposition can be performed more efficiently than QR decomposition for Hermitian positive-definite matrices [3], resulting in a significant reduction in computational complexity due to the fewer elements required to be computed in a triangular matrix. However, the Cholesky decomposition algorithm has inherent dependencies that may impact its acceleration according to Equation 7 and Equation 8. This thesis aims to answer the following research questions:

1. Finding a parallel design to accelerate the ZF signal processing detection algorithm used in baseband processing of wireless communication in 5G networks and beyond to satisfy the growing demand for higher data rates and lower latency, and tackling the dependencies in the Cholesky decomposition to limit its effect on the performance.

2. Demonstrating and investigating the scalability of the ZF algorithm, specifically for the order of thousands of antennas.

3

RELATED WORK

This chapter examines the related work to this thesis, which is divided into two sections. The first section, Section 3.1, explores hardware-based solutions such as custom accelerators and Application Specific Integrated Circuits (ASICs) designed to accelerate massive MIMO systems. The second section, Section 3.2, focuses on software-oriented approaches, such as parallelisation methods, to accelerate the algorithms used in massive MIMO systems. Then, Section 3.3 presents software approaches for acceleration of matrix decomposition algorithms, that are independent of massive MIMOs. Lastly, Section 3.4 presents a summary of the literature review findings.

3.1 HARDWARE ACCELERATION OF MASSIVE MIMO ALGORITHMS

A hardware-focused paper is in [5], where the authors express that massive MIMO architectures are essential for current and next-generational wireless networks. Additionally, alongside the progress of 5G and beyond comes the increased requirements for flexibility for the architectures. With this reasoning, the authors present their implementation of a baseband massive MIMO Application-Specific Instruction set Processor (ASIP) consisting of, among other things, a Single Instruction Multiple Data (SIMD) vector processor, a parallel memory subsystem, and a systolic array, along with a Very Large Instruction Word (VLIW) architecture. Their architecture achieves the wanted flexibility from the programmable processor while simultaneously having Application Specific Integrated Circuit (ASIC) performance. The ZF algorithm is used for the detection, and the QRD, extended QRD and Cholesky decomposition algorithms (accompanied by backward- and forward substitutions) are investigated and compared in terms of operational complexity. Their ASIP performance reached a throughput of 3.93 Mb/s for the Cholesky decomposition. They have a similar problem formulation to this thesis, "*In the post-Moore era the focus has shifted from component miniaturization to algorithms/software performance engineering along with specialization of computer architecture.*" (p. 3814). Their focus takes the hardware-specific route, whereas this thesis focuses solely on software performance and implementing it on a GPU.

The authors of [7] developed an adaptive uplink detection scheme for massive MIMOs. They are using a combination of Matched Filtering (MF) and ZF with Cholesky. The MF is used for non-interference-limited users, whereas the ZF is used for the rest of the users. They developed a custom accelerator for the calculation of the Cholesky

decomposition that works up to 16x16 matrices, and is optimal for 8x8 matrices. They used a block-based Cholesky decomposition that utilises parallelism. In summary, their focus is on the creation of the accelerator in regards to hardware rather than creating a scalable Cholesky decomposition algorithm that can handle larger-sized matrices.

3.2 SOFTWARE ACCELERATION OF MASSIVE MIMO ALGORITHMS

The authors of [22] present a decentralised architecture for baseband processing, specifically for data detection and beamforming algorithms (ZF and MMSE), by distributing the computations across GPU clusters. They demonstrate the scalability of their implementation for up to 32 users and up to 1024 BS antennas and found that their decentralised approach achieves a low complexity, but the throughput is not high enough for 5G wireless systems. They use the Cholesky factorisation and forward-backwards substitution using existing functions from the cuBLAS [23] library. This indicates that their approach does not delve into the optimisation of the Cholesky decomposition together with the ZF algorithm, which is central to our thesis work.

In [24], the authors propose the use of GPUs to accelerate the signal processing of MIMO radar, providing an alternative to the 3D Fast Fourier Transform (FFT). This paper does not have any connection to detector algorithms, but it does highlight the potential of GPUs to improve performance in terms of processing time and resolution for MIMOs.

In paper [25], non-linear approaches for multiuser massive MIMOs are investigated. They discuss the use of GPUs and FPGAs, and many-core architectures for this purpose, and suggest that further research needs to be done to unlock the full potential of future parallel non-linear processing for 6G systems.

The authors of [26] implemented a partially linear multiuser detection algorithm for massive MIMOs, based on the Adaptive Projected Subgradient Method (APSM). The algorithm is implemented on a GPU using CUDA to exploit the parallelism capabilities of the GPU, such as latency hiding and shared memory utilisation, to achieve real-time multiuser detection. The paper demonstrates that the system can perform multiuser detection with a detection latency of below one millisecond, meeting the requirements of 5G and beyond systems.

In [27], the authors present a parallel technique to compute the estimation vector of a linear massive MIMO detector on a GPU using the CUDA programming language. The paper focuses on the Cayley Hamilton with Maximum Eigenvalue (CHTME) linear detection algorithm, which is different from the standard ZF algorithm discussed in our thesis. They acknowledge a significant memory copy overhead in their GPU implementation. However, when ignoring this overhead,

the runtime of the estimation of the vector reaches 33.33 microseconds for 64 users. The primary focus of acceleration in this paper is the parallelisation of matrix-vector multiplication.

In [11], the implementation of the ZF algorithm with QR decomposition was presented on a multi-core microcontroller platform, the Epiphany board [28]. The results concluded that the Epiphany board is unsuitable for the calculations and proposed that the platform requires hardware updates to be viable to be used for massive MIMO systems. The Epiphany board in use consists of 16 cores, and it is mentioned in the future scope that a 64-core Epiphany board would have achieved a greater speed-up factor but still not comparable to the state-of-the-art architectures. The sizes of the matrices used for the detector algorithm are at most 16×16. The QR decomposition algorithm was used for the ZF algorithm, and the Householder Transformations and Givens Rotation algorithms were used for the matrix inversion.

A paper that chose to work with GPUs for programmability and use pipelining and parallelism techniques is [4], where the authors presented a GPU-based uplink detector for massive MIMOs. They use the MMSE detector using Cholesky decomposition for more accurate results, and use conjugate gradient for less accurate results. They are using the cuBLAS [23] library to calculate the Cholesky factorisation. Contrary to this thesis, they are not optimising the Cholesky decomposition itself but are more focused on switching between Cholesky and conjugate gradient.

In [3], the authors analyse traditionally used matrix decompositions for small-scale MIMOs. The decompositions analysed in their paper are the QR, Cholesky and LDL. They compared the three with complexity analysis and found that Cholesky and LDL are less complex than QR. They conclude that the Cholesky and the LDL decomposition are most suitable for a massive MIMO system, where the LDL does not use the square root operations at the expense of more matrix multiplications. Their analysis is done on systems up to 256 BS antennas and 32 users, which are smaller values than what is used in this thesis. Additionally, they conclude that the decomposition algorithms should be analysed using a more realistic channel model for their future work. In contrast to this thesis, this paper's focus is the comparative analysis between different decomposition methods, not the actual implementation and acceleration of a specific system. Therefore, they do not discuss parallelism methods for the algorithms.

Another paper about the acceleration of detection/precoding algorithms for massive MIMOs is [29]. They explore the optimisation of the lower physical layer for 5G Physical Uplink Shared Channel (PUSCH) reception on two many-core systems (MemPool [30] with 256 cores and TeraPool with 1024 cores) by implementing the software

design for parallelisation of three important kernels: Fast Fourier Transform, Matrix-Matrix Multiplication, and Matrix Decomposition (Cholesky decomposition). The authors compare their systems with a single RISC-V core serial execution of the kernels, finding that their implementation of the optimised parallel kernels can be a promising approach for PUSCH parallel software implementations since it reached a speedup of 871 for the TeraPool system. Contrary to this thesis's work, the main focus is solely on accelerating the decomposition algorithm rather than the whole detector algorithm. Also, their implementation is applied on many-core processors, whereas this thesis's work targets parallelisation using GPU processors. Another difference is that the sizes of the matrices to be decomposed are, at most, 32x32, whereas we aim to have a scalable system including larger-sized matrices.

3.3 SOFTWARE ACCELERATION OF MATRIX DECOMPOSITION ALGORITHMS

The authors in [12] analysed the parallelisation capabilities of the Cholesky decomposition and proposed an implementation on GPUs using the OpenCL framework [31]. They used the LDL decomposition to avoid using square roots because they claim it is not precise and slow for a GPU. In contrast, this thesis uses the LL Cholesky decomposition and works in CUDA. The takeaway of this paper is that they used a *blocked* Cholesky method to parallelise the algorithm, which is a recurring approach in the related work regarding our thesis. Another paper [32] uses a similar method, the Block Cholesky method [14], which is explained in Section 2.2.

The authors in [32] investigate the acceleration of the Relevance Vector Machine (RVM) algorithm by parallelising the Cholesky decomposition algorithm (which is a crucial part of RVM) on GPUs. They implement a recursive version of Block Cholesky, using the LDL^T decomposition. The experimental results show that the GPU implementation has a speed-up factor of four compared to an Intel CPU. It is important to note that the authors might have dated hardware as the report is from 2010.

Another paper [15], from the same authors as aforementioned, made a performance comparison of the Cholesky decomposition algorithm on GPUs and FPGAs. They found that the FPGA implementation is the most efficient in terms of clock cycles, meaning it needs the minimum amount of cycles for the calculations, while the GPU implementation had better run time execution for larger matrices. Similarly to the previously mentioned paper [32], the same parallelisation method is used, and the hardware might be dated as it is from 2010.

An alternative way of finding the inverse of a matrix using the Cholesky decomposition algorithm is presented in [13]. They com-

pare the operational complexity with different methods for the matrix inversion and found that their proposed method gives the operational complexity of $\frac{1}{2}n^3$. Their proposed method performs better than doing matrix inversion with equation solving and triangular matrix operations. The authors did not investigate possible parallelism in their approach.

3.4 SUMMARY

Regarding software acceleration, the focus has been primarily on parallelising the decomposition algorithms used using methods such as Block Cholesky. However, not the parallelisation of the whole detector algorithm. In contrast, this thesis focuses on parallelising the ZF algorithm alongside the Cholesky decomposition.

In terms of hardware selection, the related work consists of using custom accelerators, ASICs, many-core platforms and GPUs. The GPU approaches, however, have lacked the investigation of optimising specifically the ZF algorithm together with the Cholesky decomposition.

In general, regarding the scalability aspect, the massive MIMO systems in the related work assume the number of antennas to be in the order of hundreds. In contrast, this thesis assumes the number of BS antennas reaches the order of thousands instead.

4 METHOD AND IMPLEMENTATION

This chapter explains the proposed method in Section 4.1, going through the column-by-column matrix inversion in Section 4.1.1 and explains the proposed algorithm and explanation in Section 4.1.2. In Section 4.2, a complexity analysis is conducted. The platform used to apply the proposed method to is explained and argued for in Section 5.1.

4.1 PROPOSED METHOD

This thesis aims to answer the research questions presented in Section 2.4 as follows:

1. **Finding a parallel design to accelerate the ZF algorithm** - This thesis creates a novel parallel design of the ZF algorithm and addresses the dependency problem in the Cholesky decomposition by pipelining stages of the Cholesky decomposition and matrix inversion. The different stages of calculating the ZF algorithm can be pipelined as the stages finish column-by-column, which is explained in Section 4.1.1 and Section 4.1.2.
2. **Demonstrate scalability** - To demonstrate the scalability of this thesis, we perform a complexity analysis for our proposed algorithm and assume greater amounts of antennas in the BS compared to the state-of-the-art work, reaching up to the order of thousands of BS antennas. Then, we perform a throughput analysis from those results.

4.1.1 *Column-by-column Matrix Inversion*

The column-wise matrix inversion gets the input from the Cholesky decomposition one column at a time. When the column is received, it starts inverting. The $\mathbf{L}$ matrix is the received lower triangular matrix with the size K. The $\mathbf{I}$ matrix is initialized to an identity matrix of size K and will become the resulting inverted $\mathbf{L}$ matrix $\mathbf{L}^{-1}$. When writing $\mathbf{L}_{(a,b)}$, the element in row a and column b is chosen. The * symbol means that all elements are chosen. When the ith column is received, then Equation 11 starts. It takes the ith row of $\mathbf{I}$ matrix and divides it with the first element in the received column. Then the rest of the column is iterated through where j is the iterated number from $j = i + 1$ to $j = K$. Each row $I_{(j,*)}$ becomes itself subtracted from the product of row $I_{(i,*)}$ and the element $L_{j,i}$ (Equation 12). Figure 5 shows how the first two steps of the matrix inversion of a matrix are

performed. The **I** matrix finishes its calculations row-by-row. There are two equations that are needed for the inversion. Equation 11 is used for the first index in a column, and Equation 12 is for the rest of the elements in a column.

$$I_{(i,*)} = I_{(i,*)}/L_{(i,i)} \tag{11}$$

$$I_{(j,*)} = I_{(j,*)} - I_{(i,*)}L_{(j,i)} \tag{12}$$

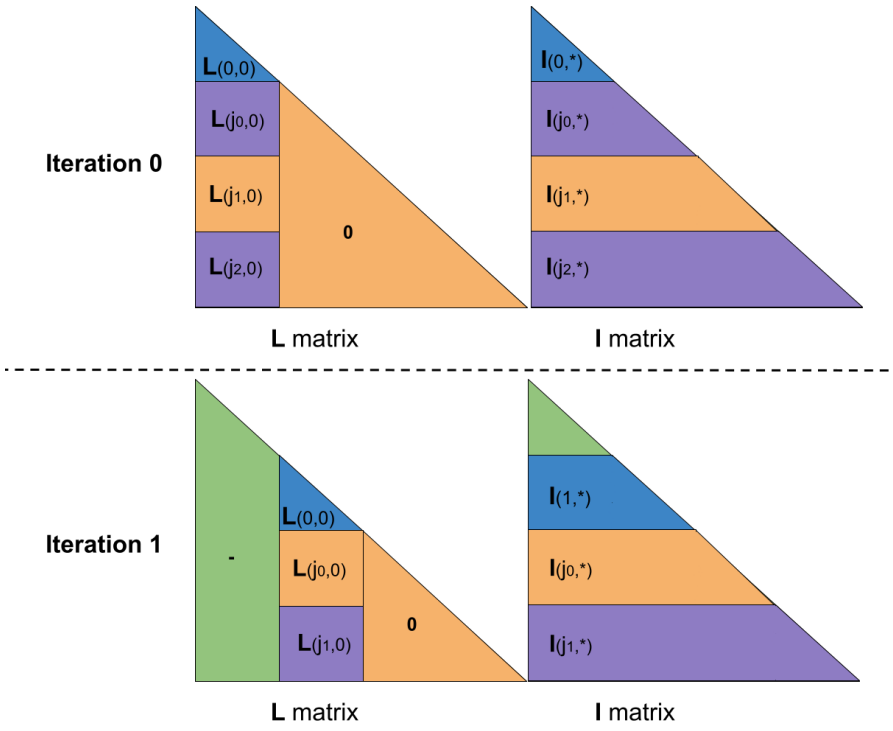

Figure 5: Column-by-column matrix inversion.

4.1.2 *Proposed Algorithm*

H is the channel matrix, $\mathbf{H} \in \mathbb{C}^{N \times K}$. The dimensions of this matrix are determined by the number of antennas on the BS, represented as N, and the amount of UE, represented as K. This matrix consists of the channel gain or loss. The received antenna vector **y** consists of the symbols (data stream) received from the users. Its dimensions are determined by the number of antennas at the BS, which is $N \times 1$. The Hermitian transpose of **H** is represented as $\mathbf{H}^H$. The Gramian matrix is denoted as **A**, while **L** is the lower triangular matrix of the Gramian matrix, according to Equation 6. $\mathbf{L}^H$ is the hermitian transpose of **L**.

The variables of the Cholesky decomposition are described in Figure 2. a_n are the diagonal elements, $\mathbf{c}_n$ refers to the column beneath

the diagonal element, and $\mathbf{U}_n$ is the remaining matrix that is yet to be calculated.

Algorithm 2 is the host code which runs on the CPU, as explained in Figure 4. Figure 6 presents an overview of the design and shows which stages run in parallel and when the different stages execute in time. The host code's primary task is to obtain the input matrix $\mathbf{H}$ and vector $\mathbf{y}$. Once these are obtained, the host launches the kernel that computes the Gramian matrix (line 1). Next, on line 2, the host synchronises the kernel, which ensures that the Gramian matrix finishes execution before continuing with the algorithm. At line 3, the Cholesky decomposition and matrix inversion pipelining begins.

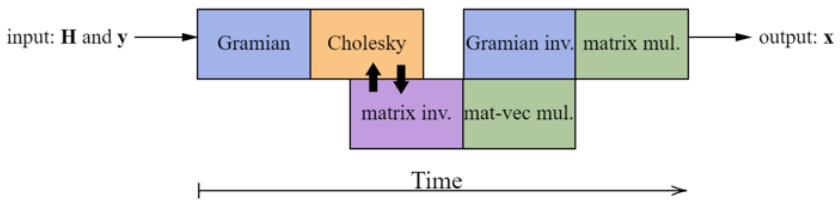

Figure 6: Overview of the algorithm's execution stages in relation to time.

Algorithm 2 Host

Input: Channel matrix $\mathbf{H}$, received symbol vector $\mathbf{y}$
Output: symbol vector $\mathbf{x}_{ZF}$

1: `launch`: calculate $\mathbf{A} = \mathbf{H}^H\mathbf{H}$
2: `synchronise` Gramian matrix
3: **for** each column K **do**
4: `launch`: Cholesky_part1
5: `synchronise` column
6: `launch`: matrix inversion_part1
7: `launch`: Cholesky_part2
8: `synchronise` matrix
9: `launch`: matrix inversion_part2
10: **end for**
11: `synchronise` $\mathbf{L}^{-1}$
12: `launch`: matrix multiplication $\mathbf{A}^{-1} = (\mathbf{L}^H)^{-1}\mathbf{L}^{-1}$
13: `launch`: calculate $\mathbf{B} = \mathbf{H}^H\mathbf{y}$
14: `synchronise` $\mathbf{A}^{-1}$
15: `launch`: matrix multiplication $\mathbf{x}_{ZF} = \mathbf{A}^{-1}\mathbf{B}$
16: `synchronise` device

The Cholesky and inversion loop iterates over each column of the Gramian matrix $\mathbf{A}$ and calculates the first part of the Cholesky decomposition at this column (Algorithm 3), modifying the same matrix $\mathbf{A}$.

Algorithm 3 computes the current column according to lines 2-4 of Algorithm 1 explained in Chapter 2.

Algorithm 3 Block Cholesky part1

Input: Matrix **A**, K size of matrix **A**, column number n
Out: Part of Matrix **L** where **A = LL**^{H}
1: **if** $a_n = diagonalelement$ **then**
2: $a_n = \sqrt{a_n}$
3: **end if**
4: `synchronise threads`
5: $\mathbf{c}_n = \mathbf{c}_n / a_n$

After this step, the threads are synchronised to enable the first part of the matrix inversion (Algorithm 4) of the corresponding row (which is the same index as the current column) to start its computations, which is performed according to Equation 11. This kernel only reads from the synchronised column and stores the result in the $\mathbf{L}^{-1}$ matrix.

Algorithm 4 Column-wise inversion part 1

Input: L received column-by-column, column number i
Out: part of inversion of L^{-1}
1: **if** diagonal element **then**
2: $L^{-1}_{(i,i)} = 1$
3: **end if**
4: $L^{-1}_{(i,*)} = L^{-1}_{(i,*)} / L_{(i,i)}$

Simultaneously, as the first part of the inversion is executing, the second part of the Cholesky decomposition (Algorithm 5) is calculated, which modifies the rest of the **A** matrix after the current column according to Equation 10.

Algorithm 5 Block Cholesky part2

Input: Matrix **A**, K size of matrix **A**, column number n
Out: prepared matrix for next column
1: $\mathbf{U}_n = \mathbf{U}_n - \mathbf{c}_n \times \mathbf{c}_n^H$

At this point, the whole matrix must be synchronised before initiating the second part of the matrix inversion (Algorithm 6), which reads from the rest of the rows of the **A** matrix and stores the result in the $\mathbf{L}^{-1}$ matrix, according to Equation 12. However, the first part of the Cholesky decomposition can already begin execution of the next column since the **A** matrix was synchronised at line 8. Then this procedure continues until the full $\mathbf{L}^{-1}$ matrix is computed.

Algorithm 6 Column-wise inversion part 2

Input: L received column-by-column, column number i
Out: part of inversion of L^{-1}

1: **for** int j = i+1, j < i, j++ **do**
2: $L^{-1}_{(j,*)} = L^{-1}_{(j,*)} - (L^{-1}_{(i,*)} \times L_{j,i})$
3: **end for**

When the Block Cholesky and the inversion are completed, the host launches two kernels that run concurrently: one for computing the matrix multiplication $(\mathbf{L}^H)^{-1}\mathbf{L}^{-1}$ to retrieve the Gramian inverse, and another for performing the multiplication of $\mathbf{H}$ and $\mathbf{y}$. Lastly, the final matrix multiplication is done at line 15, and then the host synchronises with the device and the output is obtained.

4.2 COMPUTATIONAL COMPLEXITY ANALYSIS

In this section, we perform a time complexity analysis for our proposed algorithm. We divide the algorithm into the stages that were presented in the previous section, calculate the complexity for each, and then combine them to derive the overall time complexity of the algorithm. The factors that primarily impact the complexity are the number of users (K) and the number of antennas (N).

Table 1 shows the Big O notations of the stages for both serial and parallel processing, and the required number of threads to achieve the parallel time complexity when having one thread per element in the resulting matrix.

To achieve one thread per element in these kernels, it is important to understand that it is the resulting matrix that determines the size of the threads. Consider the Gramian matrix $\mathbf{H}^H\mathbf{H}$: the matrix $\mathbf{H}^H$ is of size $K \times N$, and the matrix $\mathbf{H}$ is of size $N \times K$. As a result, the size of the output matrix is $K \times K$, hence the use of K^2 threads.

The parallel time complexity is obtained by dividing the serial time complexity by the number of threads. For example, in the calculation of $\mathbf{H}^H$, we have a matrix of size $N \times K$, and when distributed to $N \cdot K$ threads (one thread per element), the theoretical parallel time complexity becomes $\mathcal{O}((N \cdot K)/(N \cdot K)) = \mathcal{O}(1)$.

	Serial	Parallel	# Threads
$\mathbf{H}^H$	$\mathcal{O}(N \cdot K)$	$\mathcal{O}(1)$	$N \cdot K$
$\mathbf{H}^H \cdot \mathbf{H}$	$\mathcal{O}(N \cdot K^2)$	$\mathcal{O}(N)$	K^2
Chol1	$\mathcal{O}(K)$	$\mathcal{O}(1)$	K
Chol2	$\mathcal{O}(K^2)$	$\mathcal{O}(1)$	K^2
Inv1	$\mathcal{O}(K)$	$\mathcal{O}(1)$	K
Inv2	$\mathcal{O}(K^2)$	$\mathcal{O}(1)$	K^2
$\mathbf{H}^H \cdot \mathbf{y}$	$\mathcal{O}(N \cdot K)$	$\mathcal{O}(N)$	K
$\mathbf{A}^{-1}$	$\mathcal{O}(K^3)$	$\mathcal{O}(K)$	K^2
$\mathbf{A}^{-1} \cdot \mathbf{B}$	$\mathcal{O}(K^2)$	$\mathcal{O}(K)$	K

Table 1: Computational complexity of our algorithm.

To determine the overall time complexity of the algorithm, we need to include the for-loop that iterates through the number of users (K) in the host code. The calculations performed in the loop are the Cholesky decomposition and matrix inversion. Consequently, these parts are multiplied by K due to the looping process. It is important to note that the process Chol2 is executed concurrently with Inv1, and the matrix-vector calculation $\mathbf{H}^H\mathbf{y}$ occurs concurrently with the Gramian inversion $\mathbf{A}^{-1}$. This parallel execution eliminates one $\mathcal{O}(1)$ from the Chol2 and $\mathcal{O}(K)$ from the $\mathbf{A}^{-1}$ process. As a result, we find the total theoretical time complexity of the algorithm as given in Equation 13:

$$\mathcal{O}(1 + N + K(1 + 1 + 1) + N + K) = \mathcal{O}(2N + 4K + 1) \tag{13}$$

Therefore, the final time complexity of the algorithm is in the order of N, $\mathcal{O}(N)$, since the number of antennas is greater than the number of users.

4.3 PLATFORM

The platform used to execute our proposed algorithm has been chosen to be a GPU, using the CUDA programming language to implement it. A GPU provides programmability and flexibility and is optimised for throughput and handling matrix operations. This is an advantage of GPUs in this context since the aim for massive MIMOs is to increase the number of users, consequentially raising the number of antennas significantly because of the ratio requirements, which results in larger matrices to compute. Importantly, GPUs are also designed to exploit data-level parallelism. This optimises the performance of tasks with repetitive operations on different data sets. For instance, tasks such as matrix computations, where the same operation can be done on multiple data simultaneously. GPUs have also

become more commonly used in real-world embedded applications, such as scheduling and communication, machine learning, image processing and computer vision applications [33]. In addition to this, GPUs are often preferred for their high throughput capabilities. It is worth noting that the cost of off-the-shelf GPUs is significantly lower compared to hardware-specific solutions that require custom design and manufacturing. Additionally, GPUs enable flexibility, allowing for scalability with minimal effort and expense.

Given the limited timeframe of this thesis, the use of CUDA is a practical and user-friendly approach to implement the proposed method and evaluate its effectiveness.

The results of this thesis are compared to the state-of-the-art work concerned with acceleration in hardware and software. Additionally, the accuracy of our implemented algorithm is verified by comparing its output results to the output of a simulator tool built using MATLAB as a part of the ELLIIT project [34].

5 RESULTS AND DISCUSSION

This chapter presents the experimental setup in Section 5.1 and the findings from two tests. The first test presented in Section 5.2 compares the execution times of the proposed method running on a GPU versus a sequential version of the proposed method running on a CPU. The second test, presented in Section 5.3, investigates the proposed method's throughput on a GPU on 128×8 and 128×16 matrices and compares the results to the related work in Section 5.4. Lastly, Section 5.5 presents the discussion and Section 5.6 the conclusions of this thesis.

Inputs for the tests, specifically the $\mathbf{H}$ matrices and $\mathbf{y}$ vectors, are generated using MATLAB. The corresponding $\mathbf{x}_{zf}$ is provided for each matrix to validate the results. CUDA events are used to measure the time of executing the ZF algorithm, not including the memory copying overhead, as this algorithm is supposed to be a part of a larger process.

An N×K matrix refers to a matrix with N antennas and K users. The notation (X,Y) details the number of threads involved. Here, X represents the block dimensions containing an X×X matrix of threads, while Y is the grid dimensions containing a Y×Y matrix of blocks. Therefore, the total number of threads used equals $X^2 * Y^2$.

5.1 EXPERIMENTAL SETUP

This thesis uses an NVIDIA RTX A5000 and an NVIDIA GeForce RTX 2080 Ti NvLink GPU platform provided by Halmstad University. This thesis work utilises CUDA, which was created by NVIDIA specifically for their GPUs.

5.2 PERFORMANCE EVALUATION

This test aims to compare the execution times of the proposed solution compared to running a sequential version of it on a CPU, achieving the speedup factor. The sequential code, a non-multithreaded version of the proposed solution in C, is executed on an Intel Xeon Processor W-2255.

Three antenna configurations are tested, 1024, 2048 and 4096 and two sets of users, 128 and 256. For the ZF algorithm, the number of users is dictated by the number of antennas in around 5:1 to 10:1 ratios. Consequently, matrices ranging from 32:1 to 4:1 ratios were selected.

In CUDA, the threads are divided into blocks. In this case, a two-dimensional array of blocks is used. Each block is within a grid, which is also represented by a two-dimensional array. This is the (X,Y) notation explained earlier. To determine the optimal block and grid proportions, a test is performed so that it examines all sizes from (1,1) to (34,8) for every matrix size, which is presented in graph 7.

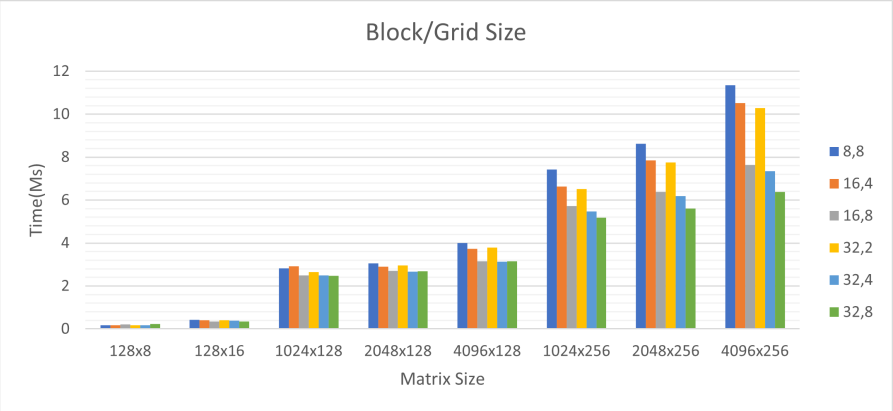

Figure 7: Comparison of what (block,grid) size are the most optimal.

From the block and grid size test having at least as many threads as the square of the number of users K gives the best performance. This is because the majority of calculations are performed on square matrices, consisting of the number of users squared ($K \times K$). Furthermore, it is notable that there is no significant downside to having unused threads to a certain limit for the calculation of one output vector. This observation can be made from all the tests with 128 users with the (32,8) thread configuration, which has twice the number of threads as the elements in the square matrix and still have a similar computation time as the (32,4) with an equal number of elements and threads. However, if the number of threads is fewer than the elements in the user matrix, there is a noticeable increase in time. As seen in graph 7, the runs that have 32 blocks give the best performance. Based on the test results, the (32,4) is optimal for the smaller-sized matrices with 128 users, and (32,8) for larger-sized matrices of 256 users.

The speedup is calculated by dividing the CPU execution time by the time taken by the GPU version. The results, as plotted in graph 8, show a significant increase in speedup with an increase in the number of antennas and users.

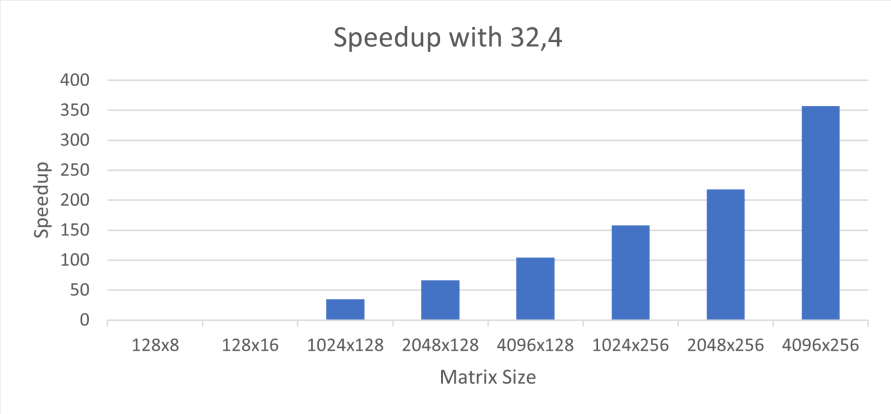

Figure 8: Speedup relative to serial code on CPU.

This is consistent with expectations, as larger matrices are capable of utilising more threads, thereby achieving more parallelism than smaller-sized matrices. Another interesting observation is that the higher speedup is associated with an increase in users. This can be observed in the comparison between the 2046×32 and the 1024×128 matrices. This is due to the fact that the ZF algorithm performs the majority of its calculations on square matrices where the dimensions are determined by the number of users ($K \times K$), as seen in algorithm 2.

5.3 ANALYSIS OF LARGER MATRICES' THROUGHPUT

In our analysis of throughput performance for larger matrices, we use the same set of larger matrices from the previous experiment. The experiments are executed for 1-80 vector calculations concurrently, with the results plotted in graph 9.

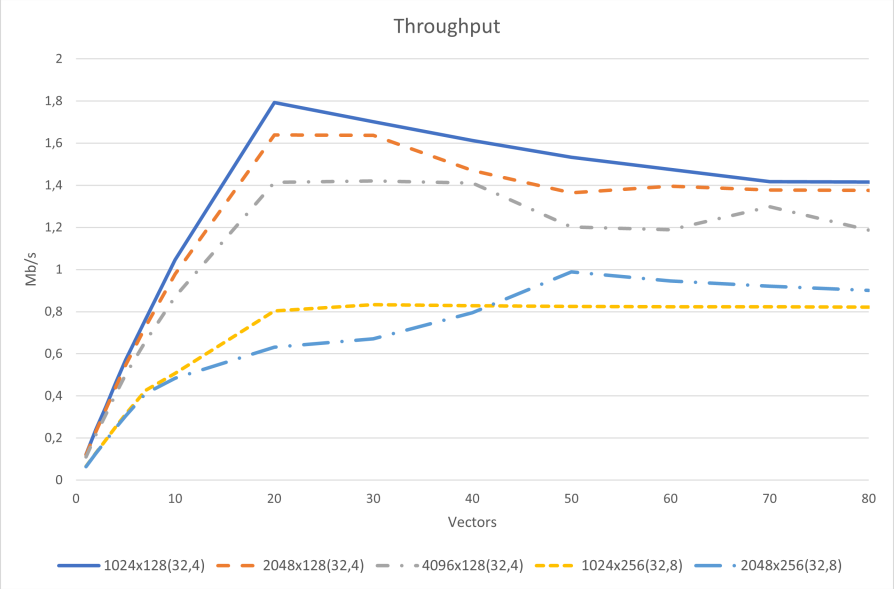

Figure 9: Throughput performance for the set of larger matrices.

The throughput of the system is influenced by the number of antennas, with improved throughput observed with a lesser amount of antennas. This outcome is expected since an increase in the number of antennas requires more computations, thus impacting the overall throughput. What is observed in the 128-user case when doubling the antennas, the throughput drops by 0.2 Mb/s but still converges to a similar throughput when the vector count is high. However, the number of users in the system has the most significant impact on performance. When the number of users is doubled from 128 to 256, the throughput is nearly halved. This suggests that the system experiences a substantial decrease in performance as the number of users increases. The experiments with the shortest vector calculation latency and the highest throughput are presented in table 2.

Matrix size	Vectors	Latency (ms/vector)	Throughput (Mb/s)
1024x128	20	0.428	1.793
2048x128	20	0.468	1.639
4096x128	30	0.541	1.421
1024x256	30	1.844	0.833
2048x256	50	1.552	0.990

Table 2: Best performing latency and throughput for the scalability test.

5.4 COMPARISON TO RELATED WORK

This section presents comparisons between the related work and our proposed method in terms of latency and throughput. The overall comparisons are presented in table 3.

5.4.1 *Comparison with work of M. Attari et al.*

This section compares the throughput performance of the proposed method to [5]. The experiments are performed on 128×8 matrices, which is the same as in the referenced paper. An optimal configuration of (8,1) for block and grid sizes was determined from earlier tests, where we assign one thread per element in the user matrix. The GPU utilised for this comparison is the NVIDIA GeForce RTX 2080 Ti NvLink.

Throughput is calculated using equation 14:

$$\text{Throughput} = \text{users} \cdot \text{modulation\_bits} \cdot \text{vectors/sec} \tag{14}$$

where the modulation bits are determined by the 64QAM modulation scheme, which is 6, and the number of users for this test is 8. The vectors per second are obtained by dividing the number of vectors by the time from the first vector's appearance to the completion of the last. The results presented in graph 10 show an initial steady increase in throughput until it reaches 10 vectors, after which it slowly increases, converging around 1.6 Mb/s. The highest throughput achieved is 1.575 Mb/s with a latency of 0.030479 ms/vector, which occurs at 90 vectors. However, a throughput of 1.512 Mb/s is already achieved at 50 vectors.

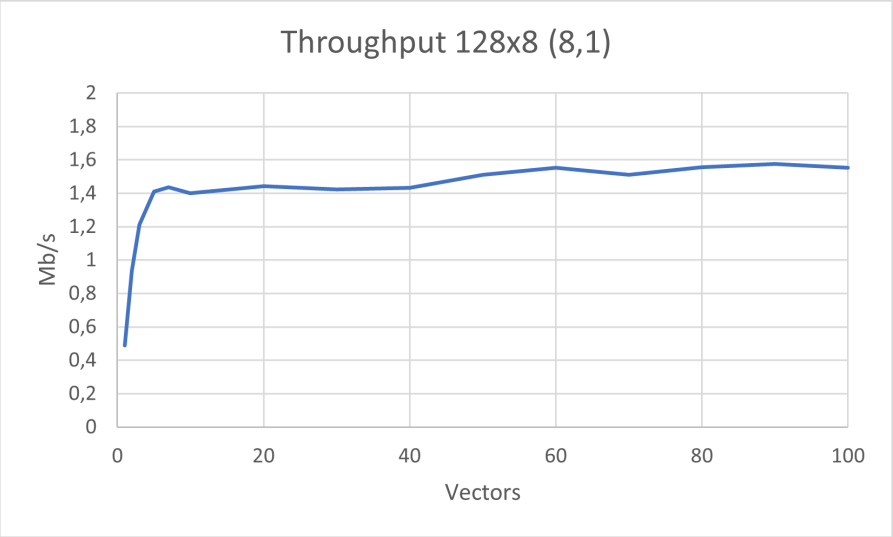

Figure 10: Throughput for 128 antennas and 8 users.

When using the data from [5] with 8 users, 64QAM (6 modulation bits), and given latency of 0.0122 ms/vector for each vector calculation, it gives a throughput of 3.934 Mb/s. Comparing our result with [5], our throughput is 2.4 times lower.

5.4.2 *Comparison with work of K. Li et al.*

This section presents a performance comparison between the proposed solution and [4]. The comparison method remains the same as in the previous section, with the difference being the matrix size of 128×16 instead. The larger matrix size made it necessary to increase the grid size to 8. Similarly to previous tests, the throughput increases dramatically up to 10 vectors, followed by a slower growth up to 40 vectors, eventually converging to 1.8 Mb/s, as seen in graph 11. The peak throughput of this test is 1.846 Mb/s with a latency of 0,0520 ms/vector at 40 vectors. Considering that [4] includes the host-device memory copy overhead, we also performed a test which includes this overhead, resulting in a latency of 0.07467 ms/vector and throughput of 1,286 Mb/s.

Using the data from [4] with 16QAM (therefore 4 modulation bits), 16 users, and a latency of 3.98 ms, their throughput is then 0.008 Mb/s. Comparing this to our code, which includes host-device memory copy overhead, we achieve a 160 times faster throughput.

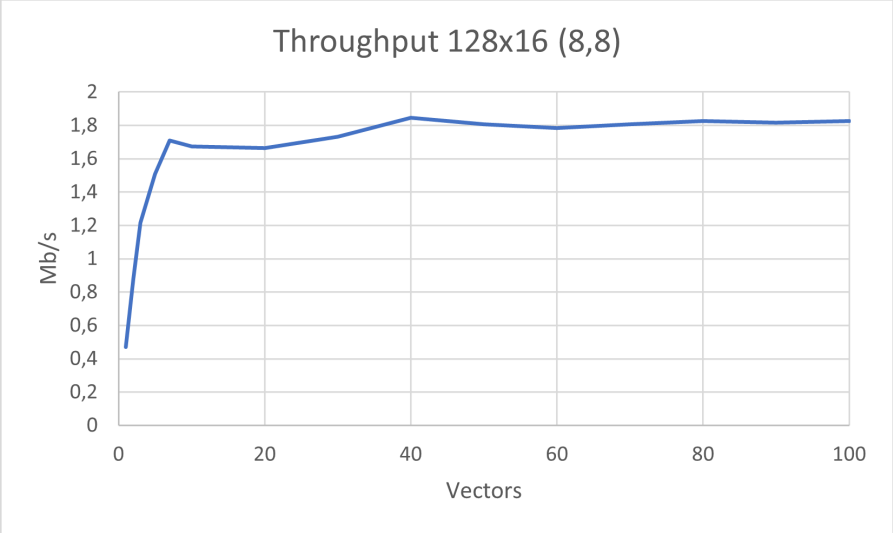

Figure 11: Throughput for 128 antennas and 16 users.

	Latency ms/vector	**Throughput** Mb/s	**Including memcpy overhead**
128x8			
Paper [5]	0.012	3.93	No
Proposed method	0.030	1.57	No
128x16			
Paper [4]	3.98	0.00804	Yes
Proposed method	0.075	1.29	Yes

Table 3: Comparisons to throughput and latency performance. Throughput = users x modulation bits x vectors/seconds.

5.5 DISCUSSION AND FUTURE WORK

5.5.1 *Discussion of Results*

Despite not matching the results of [5] in terms of latency and throughput, we are still approaching their latency. With further improvements to the code, we can get even closer to their latency. It is also important to consider the cost associated with hardware selection for massive MIMO development. While application-specific designs provide high performance, they come with a high cost. In contrast, GPUs are already available off-the-shelf, which reduces the cost financially and in terms of development time. This is one reason we advocate for further exploration of using GPUs for massive MIMO detection.

Regarding the scalability aspect of our system, we found that increasing the size of the matrices leads to a decrease in throughput, as expected due to the increase of computations needed. However, the antenna count does not have that big of an impact compared to the user count. The throughput drops significantly when doubling the number of users. This is explainable as the most computationally intensive parts of the algorithm happen on the matrices that are dependent on the number of users. Currently, there is a lack of research investigating up-link detection with bigger antenna-user configurations, as we have done. Therefore, we are unable to compare our results with these larger matrices. Instead, this thesis is a solid starting point for future work.

5.5.2 *Code Optimisation Opportunities*

There is potential for further optimisation of our code that would most likely increase the performance. The primary area for optimisation lies in throughput optimisation, specifically by enhancing the throughput code that processes multiple vectors simultaneously. It

was challenging to ensure that all vectors started computation at the same time. Currently, we use for-loops to launch the asynchronous kernel calls, where each vector begins its computation one at a time. However, modifying the code to allow all vectors to start simultaneously at time 0 could potentially result in further improvements in throughput.

Memory management presents another area where optimisation can be done. The current version of the code does not specify which part of the memory is used and therefore includes substantial access to global memory. By incorporating shared memory access for each block in the kernels, the latency would most likely decrease for each computation, thus increasing the throughput. This is due to the fact that shared memory is a significantly faster cache than global memory. Another possibility for optimising the memory is to allocate memory only for half of the matrix when using a lower/upper triangular matrix, rather than allocating for the entire matrix. This is because, in all of the computation steps in the Cholesky decomposition algorithm, half of the matrix is either the same as the other half (symmetric matrix) or consists of zeros.

5.5.3 *Further Analysis*

The results for the throughput for the 128×8 matrix and the 1024×128 are quite similar. Between these, we have the 128×16 that has the highest throughput of all. What would be of interest is examining further antenna-user configurations between the aforementioned ones to determine the peak throughput. Another test that would be interesting to perform is to have a fixed number of antennas and study the behaviour from minimum to maximum users. We have partially performed such tests, but adding more antenna-user configurations within that range could be beneficial for future work.

Another interesting find was how the choice of GPU affected the performance. The tests were performed on an NVIDIA GeForce RTX 2080 Ti NvLink, with a base clock speed of 1.35 GHz and 4352 CUDA cores. However, when tested on an NVIDIA RTX A5000, which has a base clock speed of 1.17 GHz and 8192 CUDA cores, the performance of throughput and latency improved for the larger matrices, at the expense of reduced performance for the smaller matrices. This implies that the smaller matrices benefit from the higher clock speed of the first GPU, whereas the larger matrices benefit from the increased number of cores in the second GPU. As the scope of this thesis is limited in the choice of GPUs, this could not be investigated further. However, we believe this is an interesting find that should be investigated.

It is also crucial to investigate the energy efficiency of GPU usage. This could be achieved by executing the existing design on the

GPU and recording the clock cycles and time. By then using profiling tools, energy consumption can be estimated. Through this, we can investigate the trade-off between energy efficiency and the design's performance. Unfortunately, this analysis could not be done due to time constraints and is left for future work.

5.6 CONCLUSIONS

As the number of antennas increases for massive MIMOs, the detection process becomes more computationally heavy due to the increased size of the channel matrix, resulting in more matrix calculations to consider. This thesis explores the acceleration of the ZF detector algorithm by parallelisation of the algorithm using the Cholesky decomposition, across a broader range of antenna configurations. Using the block Cholesky decomposition and column-by-column matrix inversion, we have developed a parallel design of the ZF algorithm. The design was tested on a GeForce RTX 2080 Ti NvLink GPU. For a 128×8 matrix, we achieved a latency of 0.030 ms/vector and throughput of 3.93 Mb/s. For the 128×16 matrix, the latency was 0,0520 ms/vector, and the throughput reached 1.846 Mb/s. For a 1024×128 matrix, the latency became 0.428 ms/vector and the throughput 1.793 Mb/s.

In the case of the 1024×128 matrix, the performance is found to be comparable to that of the smaller matrices. However, as the number of antennas or users further increases, the throughput begins to decrease once again.

These findings provide a significant step toward the use of GPUs for massive MIMO detection. Given that our results can be further improved to bring them closer to state-of-the-art solutions that use application-specific hardware, and considering that GPUs are general-purpose and available off-the-shelf, our approach offers an alternative with a low cost both financially and in terms of development time.

BIBLIOGRAPHY

[1] Ericsson. Ericsson mobility report. Accessed: 2023-02-22.

[2] Robin Chataut and Robert Akl. Massive mimo systems for 5g and beyond networks-overview, recent trends, challenges, and future research direction. *Sensors*, 20(10):2753, 2020.

[3] Shahriar Shahabuddin, Muhammad Hasibul Islam, Mohammad Shahanewaz Shahabuddin, Mahmoud A. Albreem, and Markku Juntti. Matrix decomposition for massive mimo detection. In *2020 IEEE Nordic Circuits and Systems Conference (NorCAS)*, pages 1–6, 2020.

[4] Kaipeng Li, Bei Yin, Michael Wu, Joseph R. Cavallaro, and Christoph Studer. Accelerating massive mimo uplink detection on gpu for sdr systems. In *2015 IEEE Dallas Circuits and Systems Conference (DCAS)*, pages 1–4, 2015.

[5] Mohammad Attari, Lucas Ferreira, Liang Liu, and Steffen Malkowsky. An application specific vector processor for efficient massive mimo processing. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 69(9):3804–3815, 2022.

[6] Liesbet Van der Perre, Liang Liu, and Erik G. Larsson. Efficient dsp and circuit architectures for massive mimo: State of the art and future directions. *IEEE Transactions on Signal Processing*, 66(18):4717–4736, 2018.

[7] Rakesh Gangarajaiah, Hemanth Prabhu, Ove Edfors, and Liang Liu. A cholesky decomposition based massive mimo uplink detector with adaptive interpolation. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, 2017.

[8] Mahmoud A Albreem, Markku Juntti, and Shahriar Shahabuddin. Massive mimo detection techniques: A survey. *IEEE Communications Surveys & Tutorials*, 21(4):3109–3132, 2019.

[9] Francesco Aquilante, Linus Boman, Jonas Boström, Henrik Koch, Roland Lindh, Alfredo Sánchez de Merás, and Thomas Bondo Pedersen. Cholesky decomposition techniques in electronic structure theory. In *Linear-Scaling Techniques in Computational Chemistry and Physics*, pages 301–343. Springer, 2011.

[10] Nusrat Fatema, Guang Hua, Yong Xiang, Dezhong Peng, and Iynkaran Natgunanathan. Massive mimo linear precoding: A survey. *IEEE Systems Journal*, 12(4):3920–3931, 2018.

[11] Yash Sawant. Beyond 5g baseband processing on epiphany architecture. Master's thesis, , School of Information Technology, 2021.

[12] Liang Wang, Zhang Yisheng, Bin Zhu, Chi Xu, Xiao Tian, Chao Wang, Jian Mo, and Jian Li. Gpu accelerated parallel cholesky factorization. *Applied Mechanics and Materials*, 148-149:1370–1373, 12 2011.

[13] Aravindh Krishnamoorthy and Deepak Menon. Matrix inversion using cholesky decomposition. In *2013 Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA)*, pages 70–72, 2013.

[14] The University of Texas at Austin Department of Computer Science. Cholesky factorization. Accessed: 2023-02-14.

[15] Depeng Yang, Junqing Sun, J Lee, Getao Liang, David D Jenkins, Gregory D Peterson, and Husheng Li. Performance comparison of cholesky decomposition on gpus and fpgas. In *Symposium on Application Accelerators in High Performance Computing*, 2010.

[16] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. Query processing on heterogeneous cpu/gpu systems. *ACM Computing Surveys (CSUR)*, 55(1):1–38, 2022.

[17] NVIDIA. Nvidia rtx a6000 datasheet. Accessed: 2023-03-07.

[18] NVIDIA. Nvidia quadro rtx 6000 datasheet. Accessed: 2023-03-07.

[19] Tianyi Wang and Qian Kemao. *GPU Acceleration for Optical Measurement*. 12 2017.

[20] Moises Hernandez Fernandez, Ginés Guerrero, José Cecilia, José García, Alberto Inuggi, Saad Jbabdi, Timothy Behrens, and Stamatios Sotiropoulos. Accelerating fibre orientation estimation from diffusion weighted magnetic resonance imaging using gpus. *PloS one*, 8:e61892, 04 2013.

[21] Ramandeep Singh Dehal, Chirag Munjal, Arquish Ali Ansari, and Anup Singh Kushwaha. Gpu computing revolution: Cuda. In *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*, pages 197–201, 2018.

[22] Kaipeng Li, Rishi R. Sharan, Yujun Chen, Tom Goldstein, Joseph R. Cavallaro, and Christoph Studer. Decentralized baseband processing for massive mu-mimo systems. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 7(4):491–507, 2017.

[23] NVIDIA. cublas library. Accessed: 2023-03-13.

[24] Eric Pitre, Vincent Roberge, Joey Bray, and Mostafa Hefnawi. Comparison of massively parallel algorithms on graphics processing unit for mimo radar. *e-Prime - Advances in Electrical Engineering, Electronics and Energy*, 2:100063, 2022.

[25] Konstantinos Nikitopoulos. Massively parallel, nonlinear processing for 6g: Potential gains and further research challenges. *IEEE Communications Magazine*, 60(1):81–87, 2022.

[26] Matthias Mehlhose, Guillermo Marcus, Daniel Schaufele, Daniyal Amir Awan, Nikolaus Binder, Martin Kasparick, Renato L. G. Cavalcate, Slawomir Stanczak, and Alexander Keller. Gpu-accelerated partially linear multiuser detection for 5g and beyond urllc systems. *IEEE Access*, 10:70937–70946, 2022.

[27] Sayyed Shafivulla, Aaqib Patel, and Mohammed Zafar Ali Khan. Parallel implementation of a massive mimo linear detector. In *2022 13th International Symposium on Communication Systems, Networks and Digital Signal Processing (CSNDSP)*, pages 744–749, 2022.

[28] Adapteva, epiphany introduction. Accessed: 2023-03-13.

[29] Marco Bertuletti, Yichao Zhang, Alessandro Vanelli-Coralli, and Luca Benini. Efficient parallelization of 5g-pusch on a scalable risc-v many-core processor. *arXiv preprint arXiv:2210.09196*, 2022.

[30] Matheus Cavalcante, Samuel Riedel, Antonio Pullini, and Luca Benini. MemPool: A shared-l1 memory many-core cluster with a low-latency interconnect. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, feb 2021.

[31] JY Xu. Opencl–the open standard for parallel programming of heterogeneous systems. 2008.

[32] Depeng Yang, Getao Liang, David D Jenkins, Gregory D Peterson, and Husheng Li. High performance relevance vector machine on gpus. In *Symposium on application accelerators in high performance computing*, 2010.

[33] Li Minn Ang and Kah Phooi Seng. Gpu-based embedded intelligence architectures and applications. *Electronics*, 10(8):952, 2021.

[34] Liang Liu, Håkan Johansson, Yousra Alkabani, Hazem Ali, Edfors, and Süleyman Savas. Baseband processing for beyond 5g wireless. Accessed: 2022-11-27.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

http://code.google.com/p/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

.

PO Box 823, SE-301 18 Halmstad
Phone: +35 46 16 71 00
E-mail: registrator@hh.se
www.hh.se