Master thesis

Embedded and Intelligent Systems 120 credits

A Conjugate Residual Solver with Kernel Fusion for massive MIMO Detection

Embedded and Intelligent Systems 120 credits

Halmstad 17 April 2023
Ioannis Broumas

HALMSTAD
UNIVERSITY

A Conjugate Residual Solver with Kernel Fusion for massive MIMO Detection

Master Thesis

2023

Author: Ioannis Broumas

Supervisor: Tiago Fernandes Cortinhal

Examiner: Slawomir Nowaczyk

School of Information Science, Computer and Electrical Engineering
Halmstad University
PO Box 823, SE-301 18 HALMSTAD
Sweden

A Conjugate Residual Solver with Kernel Fusion
for massive MIMO Detection
Ioannis Broumas

Master thesis report IDE 12XX
School of Information Science, Computer and Electrical Engineering
Halmstad University

Abstract

Wireless communication technology has radically altered how we communicate. Mobile devices now feature applications for web browsing, multimedia services, video conferencing, etc. Massive Multiple-Input Multiple-Output (m-MIMO) is an emerging technology promising to meet these market demands. However, the performance of a massive MIMO system dependents heavily on the signal detection technology. A mass amount of data received at the base station must be processed in parallel with ultra-low latency. Moreover, massive MIMO is under constant development thus, the supporting hardware platform needs to be flexible while the detection algorithms need to be efficient, of low complexity, and highly parallel.

Graphics Processing Units (GPU) are hardware accelerators with hundreds of parallel floating-point units, offering access to high-performance computing. Their popularity boomed when a friendly programming environment and easy-to-use libraries were made available. Nowadays GPUs have become a tool widely used by engineers, scientists, and businessmen to speed up heavy computational tasks in deep learning, scientific computation, or even create farms for cryptocurrency mining.

However, these readily available libraries often comprise of a highly optimized yet limited set of linear algebra operations. Unnecessary data communications often dominate the execution time of applications that are built with these libraries thus the applications fail to exploit the accelerators full potential. Attention then turns to kernel fusion, an optimization technique where the main idea is to merge two or more operations into one large but equivalent operation to potentially improve the overall performance.

In this thesis report a GPU implementation of an uplink detector for massive MIMO Software-Defined Radio (SDR) systems is presented. Coded in CUDA C, the linear detector employs the Conjugate Residual method to iteratively approximate the inverse matrix required in the equalization process. The algorithm's inherent parallelism is explored under two approaches that allow to complete unroll and gradually fuse all the separate kernels of the iterative solver until reaching a top-down hardcoded implementation of a single kernel. Two ways of taking advantage of the fast on chip memories for further optimization are tested. Results show the significant performance gains of kernel fusion for iterative solvers in the case of massive MIMO where many small matrices must be processed in parallel.

Contents

List of Figures

List of Tables

Acronyms and Abbreviations

3GPP	third Generation Partnership Project
5G	fifth Generation
ALU	Arithmetic Logic Unit
ASIC	Application-Specific Integrated Circuit
AWGN	Additive White Gaussian Noise
BER	Bit Error Rate
BS	Base Station
CG	Conjugate Gradient method
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
CR	Conjugate Residual method
CSI	Channel State Information
CUDA	Compute Unified Device Architecture
DSP	Digital Signal Processor
e.g.	for example (from the latin exempli gratia)
FLOP	Floating Point Operations
FPGA	Field Programmable Gate Array
GMRES	Generalized Minimal RESidual method
GPP	General Purpose Processor
GSM	Global System for Mobile communication
GPU	Graphics Processing Unit
GP-GPU	General-Purpose Graphics Processing Unit
i.i.d.	Independent and Identically Distributed
IC	Integrated Circuit
ICI	Inter-Carrier Interference
ICT	Information and Communication Technology
IEEE	Institute of Electrical and Electronics Engineers
ILP	Instruction Level Parallelism
IO	Input Output
IoT	Internet of Things
ISI	Inter-Symbol Interference
LOS	Line-of-Sight
MAC	Multiply-Accumulate
MF	Matched Filter
MGS	Modified Gram-Schmidt
MIMD	Multiple-Instruction Multiple-Data
MIMO	Multiple-Input Multiple-Output
MINRES	MINimal RESidual method
MISD	Multiple-Instruction Single-Data
MMSE	Minimum Mean Square Error
MPI	Message Passing Interface

MRT	Maximum-Ratio Transmission
MSE	Mean Square Error
MMSE	Minimum Mean Square Error
MU	Multi User
MU-MIMO	Multi User MIMO
m-MIMO	Massive MIMO
NS	Neumann Series
OFDM	Orthogonal Frequency-Division Multiplexing
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
PER	Packet Error Rate
QAM	Quadrature Amplitude Modulation
QPSK	Quadrature Phase Shift Keying
QRD	QR Decomposition
RF	Radio Frequency
RISC	Reduced Instruction Set Computer
RZF	Regularized Zero Forcing
SDR	Software Defined Radio
SER	Symbol Error Rate
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Threads
SINR	Signal-to-Interference plus Noise Ratio
SISD	Single Instruction Single Data
SM	Stream Multiprocessor
SNR	Signal to Noise Ratio
VLSI	Very Large-Scale Integration
ZF	Zero Forcing

Chapter 1

1 Introduction

Wireless communication technology has radically altered how we communicate [1]. Mobile devices now feature applications for web browsing, multimedia services, video conferencing, etc. As technology progresses, user demands are increasing and so is the bandwidth required to support them [2]. To cope with the exploding traffic growth rate and provide extensive coverage, researchers from both academia and industry must push to the limits to design novel wireless network technologies [1].
Massive Multiple-Input Multiple-Output (m-MIMO) is an emerging technology promising to overcome these challenges. Large antenna arrays at the base station (BS) beamform signals directly to the target user equipment (UE) resulting in near zero interference [3]. The configuration can be employed to either improve resilience to fading, increase data-rates, or support multiple users over the same time and frequency slot [4].
Graphics Processing Units (GPUs) are hardware accelerators with hundreds of computing cores available for parallel execution. Hardware and software advancements in the GPUs architecture and programming environment gave the opportunity to scientists and engineers to exploit the GPUs computing power for high performance computing making them a tool widely used today in deep learning, scientific computation and many more compute intense applications.
As many scientific and engineering applications do, signal processing in massive MIMO requires a sequence of linear algebra operations. The easiest way to compute these operations is to assign all matrix and vector computations to the GPU using library functions. The performance bottleneck of such generic implementations is the cost of memory access inside linear algebra kernels. Using library kernels leaves out a key optimization strategy for memory-bound computations which is to make efficient reuse of the processed data. Custom-built kernels present opportunities for memory optimization by combining multiple operations and keeping data in fast memory for efficient reuse when possible.

1.1 Motivation

The performance of a massive MIMO system is greatly dependent on the signal detection technology. The large number of antennas deployed at the base station in a massive MIMO system, generates mass amount of data, thus higher demands on radiofrequency (RF) and baseband processing algorithms [5]. The processing complexity scales with the number of base station (BS) antennas, the number of user equipment (UE), or both [3]. The detection algorithms need to be efficient, of low complexity, and highly parallel [5].

The execution time of applications with heavy or complex computations is drastically reduced using parallel architectures. However, utilizing parallel architectures is not trivial and it takes strong effort to parallelize an application [6]. Moreover, massive MIMO is a technology under development; researchers are facing numerous challenges delivering it to the market. Detection algorithms are evolving, the communication standards and protocols are updated regularly. Consequently, the hardware platform needs to be flexible enough to absorb such changes [5].
Combinations of algorithms and architectures has been proposed and implemented in the literature. In [3] the authors present a programmable 16-lane SIMD ASIP for Massive MIMO where three different algorithms are mapped onto the ASIP. In [7] a low complexity optimized Coordinate Descent has been proposed with a corresponding high-throughput FPGA design for large-MIMO systems. While such hardware implementations achieve high throughput, the flexibility offered is limited with respect to different system configurations.
GPUs are a tool widely used by engineers and scientists to speed up heavy computational tasks in deep learning, scientific computation etc. Previous work in [8] [9] [10] showed that GPUs could successfully be used for massive MIMO baseband processing. The authors in [8] [10] use an iterative solver, the Conjugate Gradient which also belongs to the Krylov subspace methods. In their implementations they include both built-in libraries and custom kernels with fusion optimization techniques while they provide some details of the implementation. However, they avoid fusing down to a single kernel, and they do not discuss the difference in performance their fused kernels have compared to the generic library kernels.
Residual based algorithms for massive MIMO detection have also been proposed in [11]. The algorithms tested are the MINimal RESidual (MINRES), Generalized Minimal RESidual (GMRES), and Conjugate Residual (CR) algorithm. The authors show that CR is an algorithm feasible for massive MIMO detection with a low computational complexity. They propose hardware implementations on FPGAs and do not discuss how the algorithm could be implemented in a more flexible hardware platform like a GPU.
In [12] [13] [14] the authors elaborate on the performance gains of kernel fusion for iterative solvers and basic linear algebra routines. They mainly focus on scientific computing where parallel computing is used to process input data of large sized matrices-vectors but that does not imply that kernel fusion would yield similar results in the case of massive MIMO where the input data comprise of many small matrices-vectors that must be processed in parallel.

1.2 Contribution

This thesis presents a comparison of a GPU implementation of the Conjugate Residual method as a sequence of generic library kernels against implementations of the method with custom kernels to expose the performance gains of a key optimization strategy, kernel fusion, for memory-bound operations which is to make efficient reuse of the processed data.

For massive MIMO the iterative solver is to be employed at the linear detection stage to overcome the computational bottleneck of the matrix inversion required in the equalization process, which is $\mathcal{O}(n^3)$ for direct solvers. A detailed analysis of how one more of the Krylov subspace methods that is feasible for massive MIMO can be implemented on a GPU as a unified kernel is given.
Further, to show that kernel fusion can improve the execution performance not only when the input data is large matrices-vectors as in scientific computing but also in the case of massive MIMO and possibly similar cases where the input data is a large number of small matrices-vectors that must be processed in parallel.
In more details, focusing on the small number of iterations required for the solver to achieve a close enough approximation of the exact solution in the case of massive MIMO, and the case where the number of users matches the size of a warp, two different approaches that allow to fully unroll the algorithm and gradually fuse all the separate kernels into a single, until reaching a top-down hardcoded implementation are proposed and tested.
Targeting to overcome the algorithms computational burden which is the matrix-vector product, further optimization techniques such as two ways to utilize the fast on-chip memories, preloading the matrix in shared memory and preloading the vector in shared memory, are tested and proposed to achieve high efficiency and high parallelism.

1.3 Thesis Outline

Chapter 2 The necessary background and relevant work is discussed, including Software Defined Radio (SDR), massive MIMO detection algorithms, GPU hardware architecture and programming model.
Chapter 3 The Conjugate Residual algorithm for massive MIMO detection is presented. A systematic analysis of the algorithm, kernel fusion and optimization strategies for GPU implementation are discussed.
Chapter 4 Calculations and performance results of different implementations.
Chapter 5 Results analysis, conclusion, and future work proposals.

Chapter 2

2 Background and Related Work

2.1 Software Defined Radio

Realizing universal communication requires smoothly integrating and utilizing numerous current and upcoming wireless communication protocols. Many of the available communication devices contain several nonprogrammable processors, each dedicated to the physical layer of a protocol. This solution is not scalable and eventually not feasible [2].
Software-defined radio (SDR) is a highly flexible, low-cost solution. Wireless protocols are implemented in software and executed on the same hardware platform, thus enabling rapid prototyping, easy troubleshooting, and multimode operation. New protocols and functions can be incorporated through simple updates, avoiding expensive and troublesome hardware modifications. SDR aims to substitute the application-specific integrated circuit (ASIC) processors used in the baseband processing with a fully programmable hardware platform [2].
GPUs' high parallel computational power has been proposed and successfully employed for SDR. GPUs offer higher flexibility than ASICs and digital signal processors (DSP), while they are much easier to program when compared to field programmable gate arrays (FPGA) [5]. Moreover, software-controlled scratchpad memories, like the ones available on GPUs, perform better than cache structures in the case of protocols that use a stream computation system with low data temporal locality [2]. For MIMO communication, GPU solutions have been presented for scheduling [15], Low-Density Parity Check (LDPC) decoding [9], digital predistortion (DPD) [16] [17] and in [18] for mobile GPUs, and more.

2.2 Massive MIMO

Modern communication systems use multiple antennas at the transceiver to enhance link performance, a technique known as multiple-input multiple-output (MIMO). MIMO can be expanded further to multi-user MIMO (MU-MIMO), where users are separated by their location in space, enabling denser networks and increased capacity. The latest concept of these large antenna array techniques is known as massive MIMO (m-MIMO).
The configuration of a massive MIMO system is shown in figure 1. A mass number of antennas is deployed at the base station to serve an analogously small number of user equipment. An essential requirement for this advanced spatial multiplexing to be successful is the channel estimate, upon which downlink (DL) precoders and uplink (UL) detectors can be implemented.

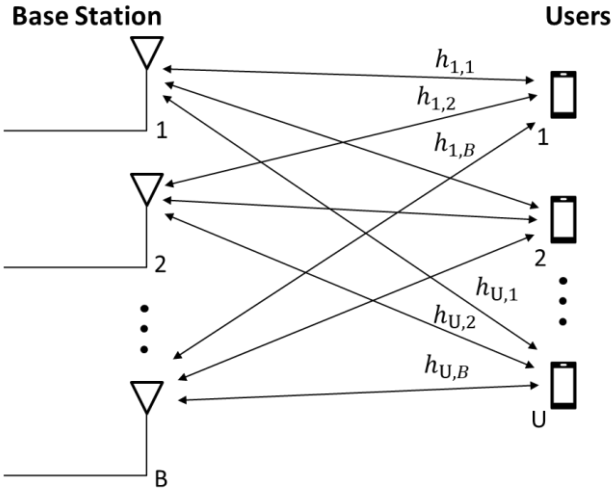

Figure 1. Simplified massive MIMO system model.

The transmitted signal vector and received vector are denoted by $\mathbf{s} = [s_1, s_2, .., s_U]^T$ and $\mathbf{y} = [y_1, y_2, .., y_B]^T$, respectively, where $\mathbf{s} \in \mathbb{C}^U$, $\mathbf{y} \in \mathbb{C}^B$. Then the system model is described as

$$\mathbf{y} = \mathbf{Hs} + \mathbf{n} \tag{2.1}$$

where $\mathbf{H} \in \mathbb{C}^{BxU}$ is an B × U uplink channel matrix, $\mathbf{n} \in \mathbb{C}^B$ is the vector representing Additive White Gaussian Noise (AWGN) with zero-mean and variance σ^2.

Massive MIMO technology achieves higher data rates than other small-scale systems by simultaneously broadcasting several data streams over the same frequency band. Downlink beamforming (precoding) yields improved spectral efficiency and reduced interference, while the technology has also the potential to decrease the costs at the BS [10]. Algorithms for traditional wireless communications are also found in m-MIMO technology, with the exception that here a greater number of data must be processed in parallel [19].

2.2.1 Detection

As the signal propagates through the medium, it is subject to distortions, attenuation, and various frequencies are delayed. Distortions caused to a symbol by neighbouring symbols is called inter-symbol interference (ISI), while distortions on a carrier due to neighbouring carriers is called inter-carrier interference (ICI). These distortions result in data errors at the decoded data at the receiver. To counterbalance these distortions channel equalization is performed on the received data. Theoretically, channel equalizers have the exact inverse frequency response of the channel. However, a perfect equalizer cannot be designed; thus, ISI will be present. Moreover, noise in the data is also amplified during the equalization process. Hence, an equalizer must balance ISI, noise, and implementation complexity [20].

Signal detection algorithms for massive MIMO can be categorized into linear and nonlinear. Nonlinear algorithms yield high accuracy in recovering the transmitted

signal but at the cost of higher implementation complexity. On the opposite side, linear algorithms are of lower complexity; their accuracy however is lower [5].
During the linear equalization process, the received signal is passed through a linear filter [20]. Zero-Forcing (ZF) and Minimum Mean Square Error (MMSE) are two popular detection algorithms. These algorithms determine the linear filter coefficients by deforming the channel matrix $\mathbf{H}$ and solving the linear matrix equation in (Eq. 2.1) [5].
Orthogonal Frequency-Division Multiplexing (OFDM) is a data transmission method also used in massive MIMO. In OFDM, the overall bandwidth is divided into several subcarriers that carry different signals at the same data rate and are transmitted over their corresponding narrowband frequencies.
In an uplink OFDM massive MIMO system, user data bits are encoded and mapped onto constellation points in a finite alphabet Ω which are then broadcasted over the wireless channel. With $y_{b,k}$ as the signal received at the b_{th} antenna for the k_{th} subcarrier at the base station, and $s_{u,k}$ as the broadcasted signal from the u_{th} user and k_{th} subcarrier, then:

$$\mathbf{y_k} = \mathbf{H_k s_k} + \mathbf{n_k} \tag{2.2}$$

where $\mathbf{y_k} \in \mathbb{C}^B$ is a vector constructed as $[y_{1,k}, y_{2,k}, .., y_{B,k}]^T$, $\mathbf{s_k} \in \Omega^U$ is a vector constructed as $[s_{1,k}, s_{2,k}, .., s_{U,k}]^T$, $\mathbf{H_k} \in \mathbb{C}^{BxU}$ is the B × U complex channel matrix and $\mathbf{n_k} \in \mathbb{C}^B$ is the noise vector $[n_{1,k}, n_{2,k}, .., n_{B,k}]^T$ with each entry $n_{b,k}$ assumed to be an i.i.d zero-mean complex Gaussian random variable with variance N_0.

Zero Forcing

In zero-forcing equalization both sides of Eq. (1.1) are left multiplied by the conjugate transpose $\mathbf{H^H}$ of the channel matrix, ignoring additive noise $\mathbf{n}$ [5].

$$\mathbf{H^H y} = \mathbf{H^H H s} \tag{2.3}$$

With matched-filter vector

$$\mathbf{y_{MF}} = \mathbf{H^H y} \tag{2.4}$$

The Gram matrix

$$\mathbf{G} = \mathbf{H^H H} \tag{2.5}$$

The detection for the transmitted signal $\mathbf{s}$ is:

$$\mathbf{s} = \left(\mathbf{H^H H}\right)^{-1}\mathbf{H^H y} = \left(\mathbf{H^H H}\right)^{-1}\mathbf{y_{MF}} = (\mathbf{G})^{-1}\mathbf{y_{MF}} \tag{2.6}$$

In Eq. (1.5) the noise is ignored. Therefore, if $\mathbf{W_{ZF}}$ is an equalization matrix with

$$\mathbf{W_{ZF}} = \left(\mathbf{H^H H}\right)^{-1}\mathbf{H^H} = (\mathbf{G})^{-1}\mathbf{H^H} \tag{2.7}$$

And

$$\mathbf{W_{ZF} H} = \mathbf{I} \tag{2.8}$$

where $\mathbf{I}$ is the identity matrix.
The transmitted signal $\mathbf{s}$ can be estimated as:

$$\hat{\mathbf{s}} = \mathbf{W_{ZF}}(\mathbf{Hs} + \mathbf{n}) = \mathbf{s} + \mathbf{W_{ZF} n} \tag{2.9}$$

In ZF equalization the estimated signal $\hat{\mathbf{s}}$ is equal to the transmitted signal $\mathbf{s}$ when the additive noise $\mathbf{n}$ is zero. ZF equalization can eliminate ISI and yield good results under high signal to noise ratio (SNR) [5]. However, noise ends up being amplified [20].

Minimum Mean Square Equalizer

The MMSE detection algorithm tries to minimize the difference between the estimated signal $\hat{\mathbf{s}} = \mathbf{Wy}$ and the transmitted signal $\mathbf{s}$. The objective function is:

$$\hat{\mathbf{s}} = \mathbf{W}_{\mathrm{MMSE}} = \underset{\mathbf{W}}{\operatorname{argmin}}\, \mathrm{E}\|\mathbf{s} - \mathbf{Wy}\|^2 \tag{2.10}$$

Solving Eq. (1.9) leads to

$$\mathbf{W}_{\mathrm{MMSE}} = \left(\mathbf{H^H H} + \frac{\mathrm{N}_0}{\mathrm{E}_s}\mathbf{I}_{\mathbf{N_t}}\right)^{-\mathbf{1}} \mathbf{H^H} \tag{2.11}$$

where $\mathbf{N_0}$ is the spectral density of noise, $\mathbf{E_s}$ is the spectral density of the signal, and $\mathbf{I_{N_t}}$ is the identity matrix [5]. In comparison, the MMSE performs better than the ZF equalizer, especially at low SNR [20].

In both ZF and MMSE detection algorithms, the calculation of a matrix inverse is required. With the channel matrix $\mathbf{H}$ being large in massive MIMO systems, this matrix inverse operation is difficult to realize in hardware in the signal detection circuit. Many algorithms have been proposed to bypass the intricacies of the matrix inversion operation. Some of the most popular being the Neumann Series Approximation (NSA) algorithm, the Chebyshev iteration algorithm, the Jacobi iteration algorithm, and the Conjugate Gradient algorithm [5].

2.3 GPU Parallel Computing

A Graphics Processing Unit (GPU) is a commercially available off-the-shelf solution with hundreds of parallel floating-point units that offers access to high-performance computing. GPUs became very popular among programmers of scientific applications when their computing power was combined with programming languages that made GPU programming easier [21].

The GPU architecture is centred on a scalable array of Streaming Multiprocessors (SMs), with each SM allowing hundreds of threads to execute concurrently. Practically all types of parallelism are represented: multithreading, Multiple Instruction Multiple Data (MIMD), Single Instruction Multiple Data (SIMD), and instruction-level parallelism [22]. In this thesis the GPU used was the GeForce 920MX by NVIDIA. In table 1 the results are shown of a device configuration query of the graphics card that was used.

GeForce 920MX	
GPU Architecture	Maxwell
CUDA Capability Major/Minor version number	5.0
Streaming Multiprocessors	2
CUDA Cores	256
GPU Clock rate	993 MHz (0.99 GHz)
Memory Speed	1800 MHz
Memory Clock rate	900 MHz
Memory Interface Width	64-bit
Memory Bandwidth (GB/sec)	14.40
L2 Cache Size	1048576 bytes
Total amount of shared memory per block	65536 bytes
Maximum number of threads per multiprocessor	2048
Maximum number of threads per block	1024
No of kernels that can execute concurrently	1

Table 1. Device properties of the GPU used.

2.3.1 The CUDA Programming Model

A GPU serves as a co-processor to a CPU. As shown in figure 2, GPUs operate in combination with a CPU-based host connected through a PCI-Express bus. In GPU computing terms, the CPU is referenced as the host while the GPU as the device [22].

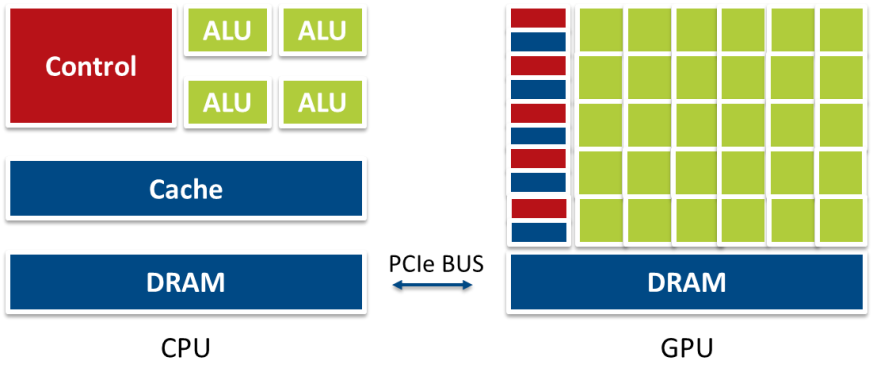

Figure 2. CPU-GPU architecture and interconnection as a heterogenous compute node.

A heterogeneous application comprises two parts: the host code and the device code. Host code is executed on the CPU while device code is executed on the GPU. The application is initialized by the CPU. The environment, code, and data are managed by the CPU code before transferring the heavy computation tasks on the device [22]. Compute Unified Device Architecture (CUDA) is a general-purpose parallel computing platform developed by NVIDIA Corporation that enables parallel computing on NVIDIA GPUs [23]. Developers can access the platform through extensions to industry-standard programming languages, like C/C++, Fortran, and Python [22]. This thesis is focused on CUDA C programming.

Host code and device code are separated during compile by NVIDIA's CUDA nvcc compiler. The host code, written in standard C, is compiled with C compilers. The device code is in CUDA C, an extension of ANSI C with keywords for marking data-parallel functions named kernels, is compiled by nvcc [22].
Accelerated libraries are also available for CUDA. For example: cuBLAS Basic Linear Algebra Subprograms (BLAS) library [24], cuFFT Fast Fourier Transforms [25], cuSOLVER Direct Linear Solvers [26]. While in [27] the authors present their implementation of a library specifically built for MIMO communication systems.

2.3.2 The CUDA Execution Model

In CUDA threads are packed into blocks and a collection of blocks form a grid. Launching a kernel grid, allocates the blocks of the grid onto available SMs for execution. Multiple thread blocks may be resident on one SM and remain there until their execution completes. Threads of a thread block can execute concurrently only on the SM that they are allocated to [22].
CUDA coined the term Single Instruction Multiple Thread (SIMT) architecture to organize and run threads in groups of 32 called warps. Thread blocks allocated to a SM are partitioned into warps. Threads in a warp execute the same instruction simultaneously. Threads have distinct instruction address counters, register states, and execute the instruction on thread dedicated data [22].
Once each SM has partitioned the thread blocks assigned to it into warps, they are scheduled for execution by the warp scheduler. Warps can be scheduled in any order, but the number of active warps is limited by SM resources [22]. Threads of a warp execute in a lock-step mode and warps are minimum scheduling units in SMs [28]. The warp scheduler of an SM can switch between eligible warps with no added overhead. If a warp for some reason is idle (for example stalled in a synchronization barrier), then the warp scheduler can select another warp that is available to execute thus effectively hiding instruction latencies [22].

2.3.3 CUDA Memory Model

Memory in modern computer architectures is hierarchically organized, thus the name ''Hierarchical Memory''. From the engineering side, it is not yet possible to create a high-capacity memory with high access speed [29]. A typical memory hierarchy is illustrated in figure 3.

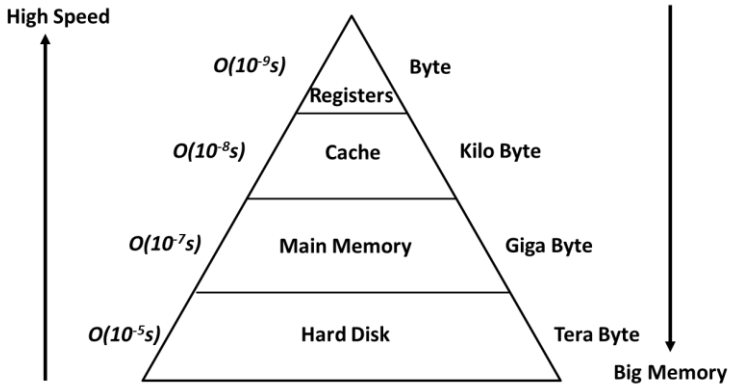

Figure 3. Hierarchical memory in modern computer architectures.

The CUDA memory model offers programmers explicit control over many types of memory, the characteristics and behaviours of these memories vary as illustrated in table 2.

Memory	On/Off chip	Cached	Access	Scope	Lifetime
Register	Off	n/a	R/W	Thread	Thread
Local	Off	*	R/W	Thread	Thread
Shared	On	n/a	R/W	Block	Block
Global	Off	*	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

* Cached only on devices with compute capability 2.x

Table 2. CUDA memory model

Shared memory, also referred to as software cache or scratchpad memory, is the most important in a GPU. The programmer has full control of the data elements to be cached thus reaching data caching efficiencies up to 100% [30].

Chapter 3

3 Implementation

The equalization process requires the computation of a pseudo-inverse of the channel matrix **H**. It consists of two matrix multiplications, one matrix inversion, and one matrix-vector multiplication. The most compute intense of these operations is the matrix inverse [19].
Solving equations of the form $\mathbf{Ax} = \mathbf{b}$ is an elementary assignment in linear algebra and scientific computing [31]. For m-MIMO the equivalent of matrix **A** would be the Gram matrix $\mathbf{A} = \mathbf{G}$ for ZF equalization, while for MMSE equalization the equivalent of matrix **A** would be the regularized Gram matrix $\mathbf{A} = \mathbf{H^H H} + (\mathbf{N_0}/\mathbf{E_s})\mathbf{I_{N_t}}$. The straightforward solution to calculating $\mathbf{A}^{-1}$ is to decompose the matrix. Hardware implementations of the matrix inversion have higher complexity and cost. The three main methods to compute the matrix inverse are explicit inversion (direct methods), implicit inversion (indirect methods), and polynomial expansion [19].
In the cases where **A** is symmetric and positive definite the Cholesky decomposition is the most efficient among direct methods like LU or QR [31]. In [3] the authors map three different direct methods onto the ASIP: basic QRD, extended QRD, and Cholesky decomposition. Direct methods can produce an exact solution, however, because of their high computing time they are not ideal for large systems.
Iterative numerical methods are techniques of lower complexity and memory footprint than direct methods [8]. Starting from an initial guess, iterative methods seek for an approximate solution of the linear system [32]. In [33] the authors present a matrix inversion based on Chebyshev and Newton iterations, in [34] a detection algorithm is proposed based on the Jacobi (JA) and Gauss–Seidel (GS) methods, in [35] a Neumann series based low-computational complexity method is presented.

3.1 Pre-processing

At the pre-processing stage the matrix is prepared to be inverted. This involves the calculation of the Gram matrix (Hermitian symmetric matrix) **G**

$$\mathbf{G} = \mathbf{H^H H}$$

the computation of the matched filtering vector $\mathbf{y_{MF}}$

$$\mathbf{y_{MF}} = \mathbf{H^H y}$$

and in the case of MMSE detection one more step that is the calculation of the regularized Gram matrix

$$\mathbf{H^H H} + \frac{\mathrm{N_0}}{\mathrm{E_s}}\mathbf{I_{N_t}} = \mathbf{G} + \frac{\mathrm{N_0}}{\mathrm{E_s}}\mathbf{I_{N_t}}$$

3.2 Conjugate Residual Method

Krylov subspace methods are regarded to be the most important amongst iterative methods [36]. In a system of n linear equations $\mathbf{Ax} = \mathbf{b}$, if x_o is an approximate starting value of the solution of the equation (here the matched filter vector is used), these techniques based on projection processes, focus on minimizing the residual norm $\mathbf{r}_o = \mathbf{b} - \mathbf{A}x_o$ by generating a series of approximate solutions.

The Conjugate Gradient (CG) and the Generalized Minimal RESidual (GMRES) methods are extensively used Krylov subspace methods for solving symmetric/non-symmetric positive definite (SPD) linear systems iteratively. Previous work in [10] [8] shows that CG based precoder/detector for m-MIMO can reduce the computational complexity while achieving good bit-error-rate (BER) performance.

The Conjugate Residual (CR) algorithm, shown in figure 4, has a similar structure as CG and can be applied in the case where **A** is Hermitian. Shown in [8] [10] [11] three iterations of the solver are sufficient to reach a close enough approximation of the exact solution for m-MIMO, while according to [11] matrix **A** can be Hermitian and CR is feasible for massive MIMO detection.

ALGORITHM: Conjugate Residual solver for m-MIMO linear detection

Input: **A** and y_{MF}

1. *Compute* $x_0 := 0, r_0 := y_{MF},\ p_o := r_0$
   $m_o := Ar_o, e_o := Ap_o$
2. *For j = 0, 1, …, until k do:*
3. $\quad a_j := (r_j, m_j)/\|e_j\|^2$
4. $\quad x_{j+1} := x_j + a_j p_j$
5. $\quad r_{j+1} := r_j - a_j e_j$
6. $\quad m_{j+1} := Ar_{j+1}$
7. $\quad \beta_j := (r_{j+1}, m_{j+1})/(r_j, m_j)$
8. $\quad p_{j+1} := r_{j+1} + \beta_j p_j$
9. $\quad e_{j+1} = m_{j+1} + \beta_j e_j$
10. *EndDo*

Output: $\hat{x} = x_j$

Figure 4. The Conjugate Residual algorithm for m-MIMO linear detection.

Compared to the Conjugated Gradient algorithm, the Conjugate Residual has one less matrix-vector product, but one more vector update [36].

Like the other Krylov subspace methods, the CR method deals with arithmetic operations on matrices/vectors. Typically, the parallel implementation of these operations is easy and more effective when dealing with large vectors. The algorithm requires to store matrix **A**, and five vectors: **x, p, Ap, r, Ar**. The computational requirements are the matrix-vector multiplication $\boldsymbol{Ar_j}$, the inner products $(\boldsymbol{r_j}, \boldsymbol{Ar_j})$, $(\boldsymbol{Ap_j}, \boldsymbol{Ap_j})$ and $(\boldsymbol{r_{j+1}}, \boldsymbol{Ar_{j+1}})$, and four axpys operations (y ← ax + y where x and y

are vectors and a is a scalar) to obtain $\boldsymbol{x_{j+1}}, \boldsymbol{r_{j+1}}, \boldsymbol{p_{j+1}}, \boldsymbol{Ap_{j+1}}$. The computation of $\boldsymbol{Ar}$ is dominant thus, the key in implementing the algorithm efficiently.
These matrix/vector operations can be implemented using the cuBLAS [24] library; however, to reduce the overhead of sharing intermediate results between kernels these operations are implemented and fused into kernels.

3.3 cuBLAS

An iterative solver, like the Conjugate Residual method, can be implemented on a GPU by expressing the main iteration as a sequence of elemental kernels executed in the order dictated by the data dependencies. The cuBLAS library provides support for dense linear algebraic computations on the GPU. Developed and maintained by lead experts; cuBLAS routines are highly optimized and serve as building blocks for linear algebra algorithms. Shown in table 3, the operation at each step of the algorithm in figure 4 and the equivalent cuBLAS function.

Step		Operation	cuBLAS Function
1.	$\boldsymbol{e_o = Ap_o}$, $\boldsymbol{m_o = Ar_o}$	Matvec	cublasCgemv
3.	$\boldsymbol{a_{j\_divident} = r_j^H Ar_j}$	Dot	cublasCdotu
3.	$\boldsymbol{a_{j\_divisor} = \|e_j\|_2}$	Norm	cublasScnrm2
3.	$\boldsymbol{a_j}$	scalar operation	-
4.	$\boldsymbol{x_{j+1} = x_j + a_j p_j}$	axpy	cublasCaxpy
5.	$\boldsymbol{r_{j+1} = r_j - a_j Ap_j}$	axpy	cublasCaxpy
6.	$\boldsymbol{m_j = Ar_j}$	Matvec	cublasCgemv
7.	$\boldsymbol{\beta_{j\_divident} = r_j^H m_j}$	Dot	cublasCdotu
7.	$\boldsymbol{\beta_{j\_divisor} = r_{j-1}^H m_{j-1}}$	Dot	cublasCdotu
7.	$\boldsymbol{\beta_j}$	scalar operation	-
8.	$\boldsymbol{p_{j+1} := r_{j+1} + \beta_j p_j}$	axpy	cublasCaxpy
9.	$\boldsymbol{e_j = m_j + \beta_j e_{j-1}}$	axpy	cublasCaxpy

Table 3. Operation at each step of the algorithm and the equivalent cuBLAS function.

Taking advantage of these highly optimized and readily available cuBLAS routines allows for rapid prototyping and deployment at a low programming effort. However, launching a sequence of ordered kernels does not yield optimal performance. For every kernel launch the contents of the registers, caches, and shared memory created by the previous kernel are flushed and data that could be reused are not persistent throughout the program execution. Unnecessary data round trips dominate the execution time of the application thus the design fails to exploit the accelerators full potential. Moreover, cuBLAS is not an open-source library; end users do not have the ability to tweak and adjust these routines.

3.4 Kernel Fusion

Host code and device code are separated at compile time. Host code, written in standard C, is compiled with C compilers, and executed on the CPU. Device code, written in CUDA C, comprises of data-parallel functions named kernels. Kernels are standalone computational routines that are dispatched to NVIDIA GPUs for parallel execution.

Kernel fusion is an optimization technique that targets to exploit the spatial and temporal locality of the data, hide memory access latencies, and minimize the overhead added from separate kernel calls. The major potential performance gain comes from eliminating global memory accesses for sharing intermediate results.

The main idea of kernel fusion is to merge two or more kernels into one large but equivalent kernel to potentially improve the overall performance. For example, suppose there are two kernels that operate on a vector. The AddKernel, which adds a constant to each element of the vector and the MulKernel, which comes after and multiplies the vector elements by a value. The sequence of operations would then be:

- AddKernel
    1. Read an element of the vector from memory.
    2. Add a constant.
    3. Write result back to memory.
- MulKernel
    1. Read an element of the vector from memory.
    2. Multiply by some value.
    3. Write result back to memory.

If these two independent kernels would be fused, then the FusedKernel would combine the source code of the AddKernel and the MulKernel in order. The sequence of operations would then be:

- FusedKernel
    1. Read an element of the vector from memory.
    2. Add a constant.
    3. Multiply by a value.
    4. Write result back to memory.

Now the instructions from the MulKernel have direct access to the output of the AddKernel without the cost of memory read instructions. That is step 1 of the MulKernel is eliminated. Further, since the output of the AddKernel is only used by the MulKernel, the associated cost of the memory write instruction is avoided. That is step 3 of the AddKernel is eliminated.

The FusedKernel produces the exact same result as the AddKernel followed by the MulKernel does, but instead with the overhead of only one kernel launch instead of two, the cost of one read instruction instead of two, and the cost of one write instruction instead of two. This overall cost reduction translates to improved performance, faster execution time. Reducing the number of read-write to memory instructions is very important specifically for memory-bound operations, with a performance gain that is usually proportional to the number of reductions.

Different fusion methods are the inner thread, inner block, inter thread block and are explained in [37]. Also, in [38] the authors present a horizontal fusion method.
Two or more kernels can be fused/merged if their properties match. They should have the same dimensionality, that is the same number of threads per block and the same number of blocks. If there are no data dependencies between the kernels then they can always be fused, whereas in the case there is a read-after-write data dependency between them then the kernels need to perform a mapped access on their data.

3.5 Unrolling

A thread strategy which fulfils the requirements of kernel fusion and would allow all the kernels to be fused is required. One thread strategy that allows all kernels to be fused would be to assign one thread per matrix. Since a single thread is processing the data, this strategy ensures that program execution follows data dependencies. The second, to map one thread to one row or column of matrix **A** and one element of each vector. Since the size of the matrix to be inverted is 32x32, it follows that one warp is assigned per matrix.
According to [8] [10] three iterations are sufficient to reach a good approximation of the exact solution in the case of m-MIMO. Unrolling the three iterations for loop is an affordable programming effort when the goal is to fully merge the algorithm into a single kernel. A pre-processing analysis of the data dependencies and order of execution paves the way to systematic fusion and to fully unroll the algorithm into a single kernel, shown in figure 5.

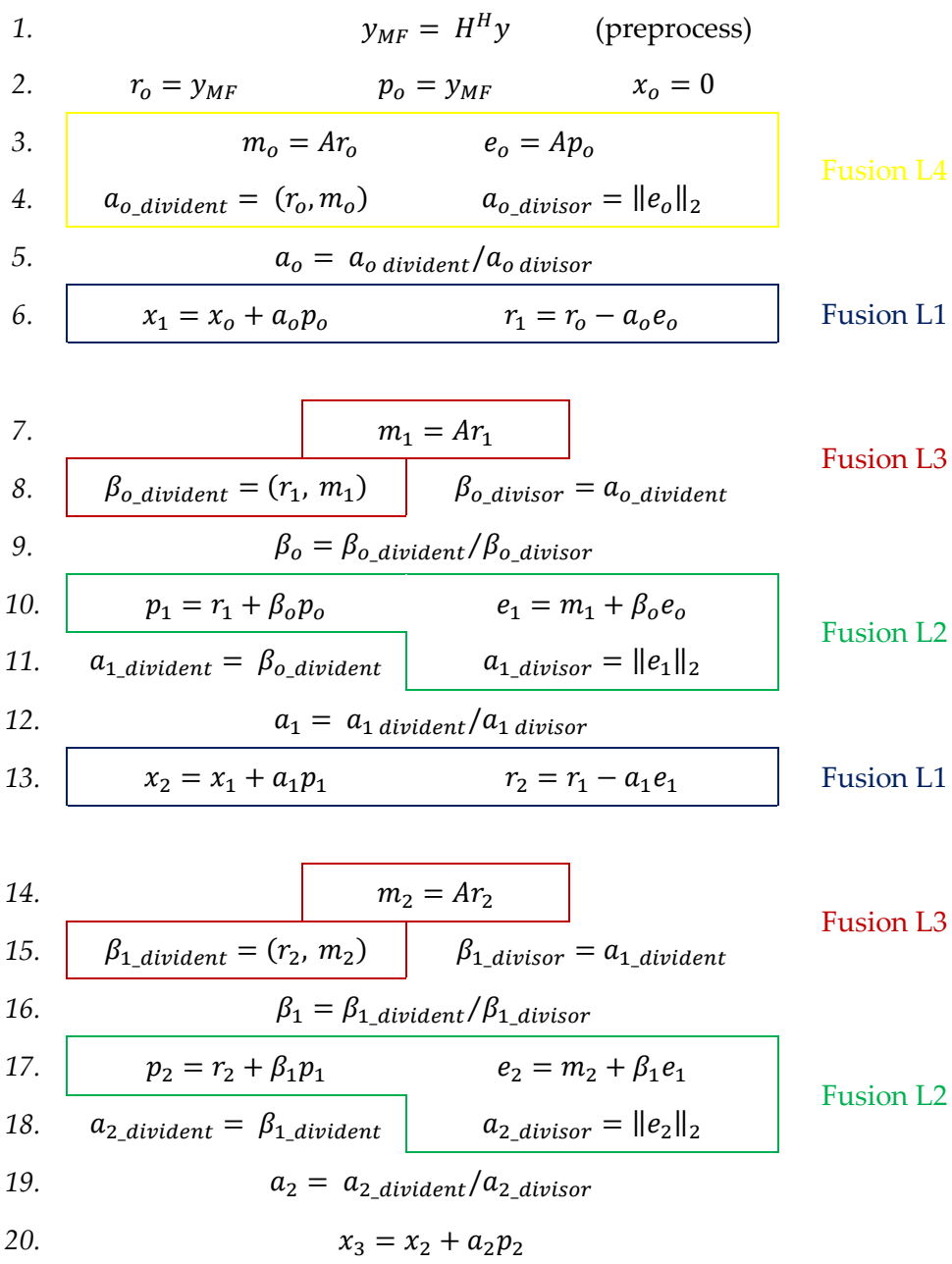

Figure 5. Analysis of the data dependencies and flow of the fully unrolled CR algorithm.

For the first step the axpy operations would be fused. There are four axpy operations in the CR algorithm in figure 4, steps 4, 5, 8, 9. Since there are no data dependencies between step 4 and step 5, they can be merged into one single routine. Similar for step 8 and step 9. Highlighted in a blue box in figure 5, instead of having a separate kernel to update each vector while switching control between CPU/GPU with the exit of each kernel, consequently adding unnecessary overhead of kernel launches, the kernels are merged into a single kernel which updates both vectors. Below is shown how the kernel could be implemented.

```
__global__ void FusedCaxpy(cuComplex alpha, cuComplex *x, cuComplex *p,
                           cuComplex *r, cuComplex *e)
{
    unsigned int idx = threadIdx.x + blockIdx.x * blockDim.x;

    if (idx < SZ)
    {
        x[idx] = cuCaddf(x[idx], cuCmulf(alpha, p[idx]));
        r[idx] = cuCsubf(r[idx], cuCmulf(alpha, e[idx]));
    }
}
```

The second step would be to fuse the axpy operations with the norm of the vector, lines 10, 11 in figure 5 marked in a green box. After updating an element of the vector **e**, the result is written to memory and then retrieved again from memory with the launch of the kernel that computes the norm of vector **e**. By fusing the kernels, the updated element is used directly to calculate a partial result of the vector norm **e,** without having to load it again from memory. Fusing these kernels results in increasing the number of arithmetic operations carried out by each thread and furthermore since the axpy operation and the norm are computed for the same vector, **e**, the number of loads is reduced by n (n the length of the vector). Below is shown how the kernel could be implemented.

```
__global__ void AxpyNorm(float *alpha, cuComplex beta, cuComplex *p,
                         cuComplex *r, cuComplex *e, cuComplex *m)
{
    unsigned int idx = threadIdx.x + blockIdx.x * blockDim.x;

    if (idx < SZ)
    {
        p[idx] = cuCaddf(r[idx], cuCmulf(beta, p[idx])); // AXPY

        cuComplex ee = e[idx]; // Load to a register
        ee = cuCaddf(m[idx], cuCmulf(beta, ee)); // AXPY
        e[idx] = ee; // Store to memory

        float a = (ee.x * ee.x) + (ee.y * ee.y);
        // reduce & broadcast
        a += __shfl_xor_sync(0xffffffff, a, 16, 32);
        a += __shfl_xor_sync(0xffffffff, a, 8, 32);
        a += __shfl_xor_sync(0xffffffff, a, 4, 32);
        a += __shfl_xor_sync(0xffffffff, a, 2, 32);
        a += __shfl_xor_sync(0xffffffff, a, 1, 32);
        // Thread zero of the warp stores to memory
        if (threadIdx.x & 31)
```

```
            alpha[threadIdx.x>>5] = sqrtf(a);
    }
}
```

The third step would be to merge the matrix-vector product operations with the dot product, lines 7,8 in figure 5 marked in red box. After calculating an element of the vector **m,** the result is written in memory and then retrieved again from memory with the launch of the kernel that computes the dot product **(r, m)**. By fusing the kernels, the calculated element of vector **m** and the element of vector **r** are used directly, without loading them again from memory, to calculate a partial result of the inner product **(r, m).** Below is shown how the kernel could be implemented.

```
__global__ void MatVecDot (cuComplex *A, cuComplex *r, cuComplex *m,
                                          cuComplex *beta_divident)
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    cuComplex mm = {0.0f, 0.0f}; // Register

    if (idx < SZ)
    {
        // Matrix-vector product
        for (int i = 0; i < N; i++) {
            mm = cuCaddf( mm, cuCmulf( r[i] , A[idx*N + i]) );
        }

        m[idx] = mm; // Store to memory

        __syncthreads();

        // Dot
        cuComplex beta = cuCmulf( cuConjf(r[idx]), mm );
        // reduce & broadcast
        beta.x += __shfl_xor_sync(0xffffffff, beta.x, 16, 32);
        beta.x += __shfl_xor_sync(0xffffffff, beta.x, 8, 32);
        beta.x += __shfl_xor_sync(0xffffffff, beta.x, 4, 32);
        beta.x += __shfl_xor_sync(0xffffffff, beta.x, 2, 32);
        beta.x += __shfl_xor_sync(0xffffffff, beta.x, 1, 32);

        beta.y += __shfl_xor_sync(0xffffffff, beta.y, 16, 32);
        beta.y += __shfl_xor_sync(0xffffffff, beta.y, 8, 32);
        beta.y += __shfl_xor_sync(0xffffffff, beta.y, 4, 32);
        beta.y += __shfl_xor_sync(0xffffffff, beta.y, 2, 32);
        beta.y += __shfl_xor_sync(0xffffffff, beta.y, 1, 32);
        // Thread zero of the warp stores to memory
        if (threadIdx.x & 31)
            beta_divident[threadIdx.x >> 5] = beta;
    }
}
```

The fourth step would be to merge the matrix-vector operation $\mathbf{Ar_o}$ with the dot product and the vector norm $\|\boldsymbol{e_o}\|^2$ at the beginning of the first iteration, lines 1, 2 in figure 5 marked in yellow box. Since $\boldsymbol{r_o} = \boldsymbol{p_o}$ and $\boldsymbol{e_o} = \boldsymbol{Ap_o} = \boldsymbol{m_o} = \boldsymbol{Ar_o}$, after calculating an element of the vectors $\boldsymbol{e_o} = \boldsymbol{m_o}$, the result is not written in memory and then loaded again for the computation of the vector norm **e** and again for the dot product **(r, m)**, instead the result is used directly, avoiding unnecessary load operations, to compute the partial results of the inner product and the norm. Below is shown how the kernel could be implemented.

```
__global__ void MatVecDotNrm (cuComplex *A, cuComplex *r, cuComplex *m,
      cuComplex *e, cuComplex *alpha_divident, float *alpha_divisor)
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;

    cuComplex mm = {0.0f, 0.0f}; // Register
    if (idx < SZ)
    {
        // Matrix-vector product
        for (int i = 0; i < N; i++) {
            mm = cuCaddf( mm, cuCmulf( r[i] , A[idx*N + i]) );
        }

        m[idx] = mm; // Store to memory
        e[idx] = mm; // Store to memory

        __syncthreads();

        // Dot
        cuComplex beta = cuCmulf( cuConjf(r[idx]), mm );
        // Reduce & broadcast
        beta.x += __shfl_xor_sync(0xffffffff, beta.x, 16, 32);
        beta.x += __shfl_xor_sync(0xffffffff, beta.x, 8, 32);
        beta.x += __shfl_xor_sync(0xffffffff, beta.x, 4, 32);
        beta.x += __shfl_xor_sync(0xffffffff, beta.x, 2, 32);
        beta.x += __shfl_xor_sync(0xffffffff, beta.x, 1, 32);

        beta.y += __shfl_xor_sync(0xffffffff, beta.y, 16, 32);
        beta.y += __shfl_xor_sync(0xffffffff, beta.y, 8, 32);
        beta.y += __shfl_xor_sync(0xffffffff, beta.y, 4, 32);
        beta.y += __shfl_xor_sync(0xffffffff, beta.y, 2, 32);
        beta.y += __shfl_xor_sync(0xffffffff, beta.y, 1, 32);

        // Norm
        float a = (mm.x * mm.x) + (mm.y * mm.y);

        // Reduce & broadcast
        a += __shfl_xor_sync(0xffffffff, a, 16, 32);
```

```
        a += __shfl_xor_sync(0xffffffff, a, 8, 32);
        a += __shfl_xor_sync(0xffffffff, a, 4, 32);
        a += __shfl_xor_sync(0xffffffff, a, 2, 32);
        a += __shfl_xor_sync(0xffffffff, a, 1, 32);
        // Thread zero of the warp stores to memory
        if (threadIdx.x & 31){
            alpha_divident[threadIdx.x >> 5] = beta;
            alpha_divisor[threadIdx.x >> 5] = a;
        }
    }
}
```

Finally, by completely unrolling the loop iteration and hardcoding the algorithm allows for an early exit, that means steps 5-9 in figure 4 do not need to be calculated for the last iteration.

Matrix-Vector multiplication

The matrix-vector **Ar** product is the main computational burden of the CR algorithm [36]. It is a memory-bound operation with low arithmetic intensity. The authors in [8] use the `cublasCgemmBatched` function from the cuBLAS library for the matrix-matrix and matrix-vector multiplications which computes a batch of small complex matrix products.
To reduce the complexity the symmetric property of the Hermitian matrix **A** could be exploited. However, as the authors in [39] explain that the obstacle in designing an efficient SYmmetric Matrix Vector (SYMV) multiplication kernel is the data storage format. The symmetric matrix would be stored as the upper or lower triangular part for which it is difficult to achieve coalesced load and store memory access.
At every iteration of the algorithm matrix **A** and vector **r** must be loaded. To use memory bandwidth efficiently matrix **A** or the vector **r** can be cached in shared memory from there they can be reached at a much higher speed than the global memory. During the multiplication process each row of matrix **A** is used only once while vector **r** is loaded for each row of the matrix **A**, this observation is not enough to draw safe conclusions on which approach would yield best results in the case of massive MIMO therefore both must be tested. A limiting factor of the kernel size would then be the number of matrices or vectors that can fit in the available shared memory.

Reductions as part of scalar products

The CR algorithms requires the computation of scalar products, which include global reductions operations (dot product, vector norm). Reductions pose a bottleneck when executing on parallel platforms, because of the synchronization barriers and communication between the processors.

CUDA's built-in function that allows threads in a block to coordinate and synchronize with each other is `__syncthreads`. It acts as a barrier at which a thread must stall execution and is allowed to proceed only when all threads in the block have reached the barrier. Synchronization barriers reduce the capability of hiding instruction latencies.

Typical implementations of reductions on GPUs make use of the shared memory and/or the `atomicadd` function to accomplish such computations, like the authors did in [8]. However, the small cache size in GPUs poses the limit for the simultaneous reduction of multiple vectors and furthermore atomics does not run at full memory bandwidth [40]. A more efficient way is to use warp shuffle commands. Threads active in a warp can exchange values between registers directly without the use of shared memory. For the special case where the number of users is 32, then the number of elements in each of the vectors is 32 and perfectly matches the size of a warp. That is reduction on a vector of size of 32 elements can be done with only a warp shuffle command. Shared memory usage is then eliminated, only a single instruction is required, and the level of explicit synchronization is reduced. In figure 5 a representation of the warp shuffle operation.

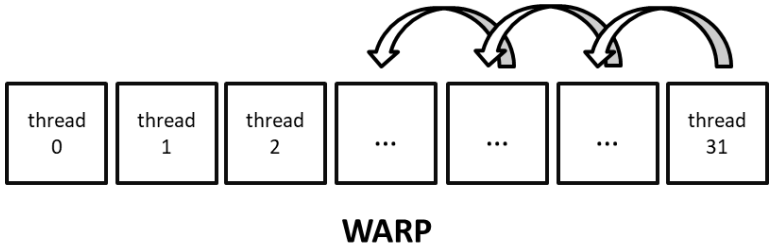

Figure 6. Warp shuffle operations.

Chapter 4

4 Results

4.1 Calculations

Shown in [8] [10] [11], Krylov Subspace Methods and the CR algorithm are of lower computational complexity than direct methods like Cholesky decomposition $\mathcal{O}(n^3)$. Compared to the Conjugate Gradient it requires one less matrix-vector multiplication but one more vector update [36]. Shown in table 4, the number of read-write to memory instructions for each operation that is required during the execution of the CR algorithm are calculated when a kernel is executing only a single operation and when a fused kernel is executing multiple operations for matrices of size n×n and vectors of size n.

	Operation	**Read**	**Write**
Single	Matvec	$2n^2$	N
	Dot	n	1
	Norm	n	1
	Axpy	2n	N
Fused	axpy + norm	2n	n + 1
	matvec + dot	$2n^2$	n + 1
	matvec + dot + norm	$2n^2$	n + 2

Table 4. Read-write operations for single and fused kernel implementations.

For the CR algorithm three iterations are sufficient to reach a good approximation of the exact solution in the case of m-MIMO according to [8] [10] [11]. Shown in table 5 the number of read-write instructions that would be required for the CR algorithm after three iterations if it is implemented as a sequence of separate kernels each executing a single operation and if it is implemented as a single fully fused with early exit kernel when the input matrix is of size n×n and the vectors of size n.

Separate	$8n^2 + 54n + 12$
fully fused + early exit	$6n^2 + 21n$

Table 5. Total number of read-write operations for the CR algorithm after 3 iterations.

The total number of complex multiplications and additions for the CR algorithm after k iterations when the input matrix is of size n×n and the vectors of size n is given in table 6.

Operation	
Addition	$n^2 + k(n^2 + 7n)$
Multiplication	$n^2 + k(n^2 + 7n)$

Table 6. Complex operations for the CR algorithm.

*k = number of iterations, n = number of elements

In an uplink m-MIMO system with 128 BS antennas and 32 users, **A** would be 32x32:

$$\boldsymbol{H}^{H}_{32\times128} \cdot \boldsymbol{H}_{128\times32} = \boldsymbol{G}_{32\times32} = \boldsymbol{A}_{32\times32}$$

vector **r** would be 32x1:

$$H^{H}_{32\times128} \cdot \boldsymbol{y}_{128\times1} = \boldsymbol{y}^{32\times1}_{MF} = \boldsymbol{r}^{32\times1}$$

To calculate matrix **A** in a per-subcarrier basis and since there are no data dependencies between subcarriers then in a massive MIMO OFDM system where the number of subcarriers, $N_{subcarrriers}$, can reach 128 and the number of symbols, $N_{symbols}$, can reach 64 then the number of matrices **A** to be inverted in parallel can reach up to:

$$N_{subcarrriers} \times N_{symbols} = 128 \times 64 = 8192$$

For a GPU with 65536 bytes of available shared memory and 2048 active threads per SM, then the number of instances of matrix **A** or vector **r** that can be fitted into shared memory and the ratio of used threads over the available is given in table 7.

	Size (bytes)	Instances	Threads	Thread Ratio
A	8192	8	256	12.5%
r	256	64	2048	100%

Table 7. Calculation of used shared memory and thread ratio.

4.2 Simulation Results

Shown in table 8, the execution time in 10^{-3} seconds for an ascending number $(N_{sub} \times N_{sym})$ of input matrices $\mathbf{A}^{32\times32}$ that must be processed in parallel, when starting from an implementation of the CR algorithm as a sequence of separate built-in library kernels and gradually replacing the separate kernels with custom fused until reaching a single fully fused kernel, under the thread strategy of one warp per matrix/vector.

$\mathbf{A}^{32\times32}$							
$N_{sub} \times N_{sym}$	Execution Time						
	Separate	Axpy	Axpy +norm	matvec + dot	matvec+dot +norm	early exit	single kernel
128	0.55	0.53	0.52	0.52	0.51	0.39	0.20
256	1.17	1.16	1.16	1.16	1.16	0.88	0.47
512	2.31	2.29	2.31	2.31	2.31	1.76	0.93
1024	4.81	4.87	4.86	4.78	4.76	3.60	1.82
2048	10.23	10.26	10.12	9.92	9.87	7.41	3.63
4096	21.28	20.82	20.68	20.19	19.97	15.18	7.24
8192	42.54	41.68	41.25	40.34	39.85	30.35	14.16

Table 8. Execution times in 10^{-3} seconds for different fusion levels.

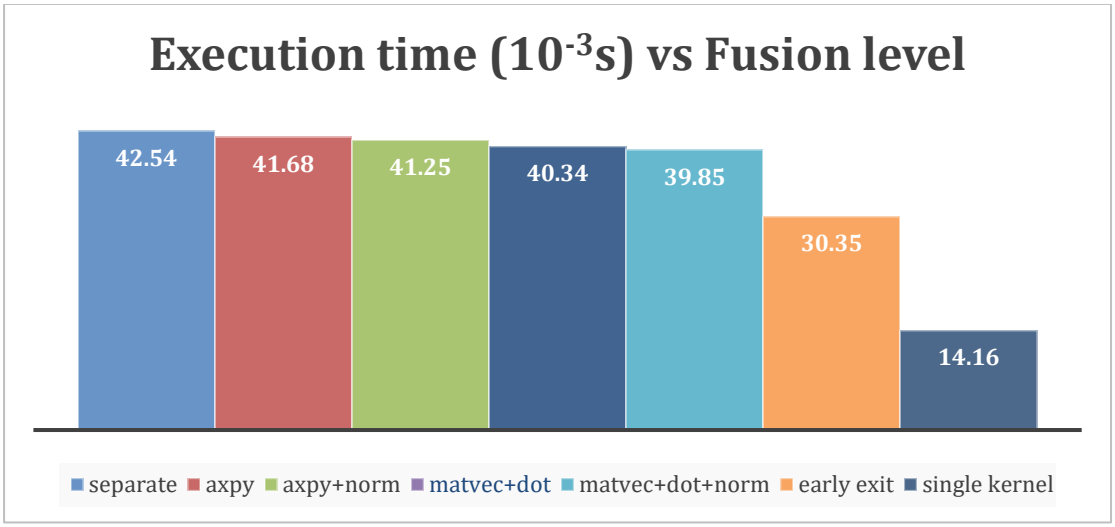

Figure 7. Execution time in 10^{-3}s for different fusion levels.

Shown in table 9, the execution time in 10^{-3} seconds for an ascending number ($N_{sub} \times N_{sym}$) of input matrices $\mathbf{A}^{32\times32}$ that must be processed in parallel, when the shared memory is not used, when matrix **A** is loaded in shared memory, and when vector **r** is loaded in shared memory, for a fully fused kernel under the thread strategy of one thread per matrix/vector.

$\mathbf{A}^{32\times32}$			
$N_{sub} \times N_{sym}$	Time		
	Shared Memory		
	-	**A**	**r**
128	0.97	1.68	0.70
256	2.06	3.37	1.30
512	4.36	6.71	2.64
1024	8.79	13.41	5.56
2048	17.96	26.69	10.89
4096	36.17	53.38	21.50
8192	72.69	106.52	42.13

Table 9. Execution times in 10^{-3} seconds for one thread per matrix/vector, with and without using shared memory.

Shown in table 10, the execution time in 10^{-3} seconds for an ascending number ($N_{sub} \times N_{sym}$) of input matrices $\mathbf{A}^{32\times32}$ that must be processed in parallel, when the shared memory is not used, when matrix **A** is loaded in shared memory, and when vector **r** is loaded in shared memory, for a fully fused kernel under the thread strategy of one warp per matrix/vector.

$\mathbf{A}^{32\times32}$			
$N_{sub} \times N_{sym}$	Time		
	Shared Memory		
	-	**A**	**r**
128	0.20	0.27	0.19
256	0.47	0.53	0.46
512	0.93	1.16	0.87
1024	1.82	2.12	1.69
2048	3.63	3.87	3.32
4096	7.24	7.29	6.51
8192	14.16	14.36	12.84

Table 10. Executions times in 10^{-3} seconds for one warp per matrix/vector, with and without using shared memory.

On table 11 and on figure 8 the results are presented from the NVIDIA Visual Profiler for the implementation without the use of shared memory.

Variable	Achieved	Theoretical	Device Limit
Occupancy Per SM			
Active Blocks		3	32
Active Warps	44.73	48	64
Active Threads		1536	2048
Occupancy	69.9%	75%	100%
Registers			
Registers/Thread		35	65536
Shared Memory			
Shared Memory/Block		0	65536
Block Limit		0	32

Table 11. GPU Utilization for the implementation without the use of shared memory.

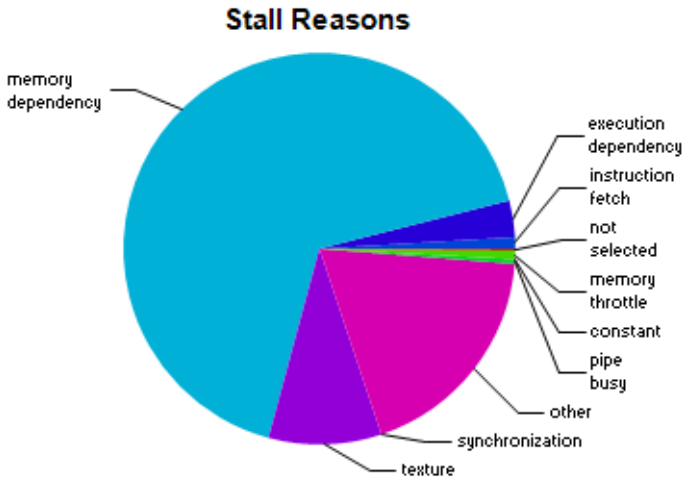

Figure 8. Stall reasons for the implementation one warp per matrix/vector without the use of shared memory.

On table 12 and on figure 9 the results are presented from the NVIDIA Visual Profiler for the implementation with matrix **A** loaded in shared memory.

Variable	Achieved	Theoretical	Device Limit
Occupancy Per SM			
Active Blocks		2	32
Active Warps	7.98	8	64
Active Threads		256	2048
Occupancy	12.5%	12.5%	100%
Registers			
Registers/Thread		34	65536
Shared Memory			
Shared Memory/Block		8192	65536
Block Limit		8	32

Table 12. GPU Utilization for the implementation with matrix A loaded in shared memory.

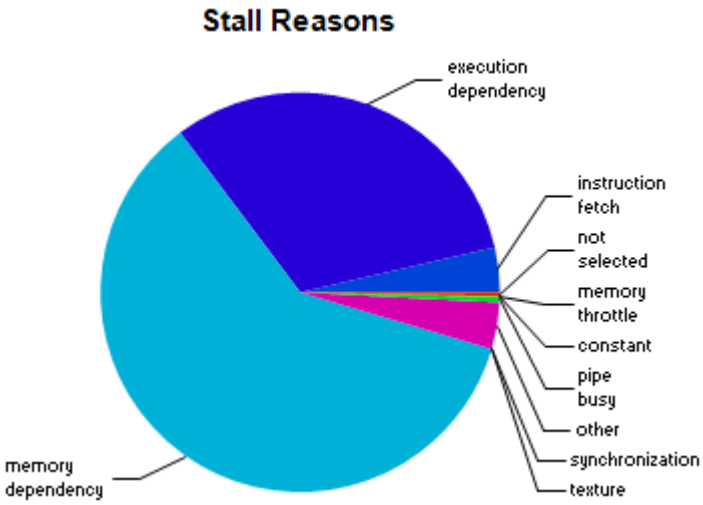

Figure 9. Stall reasons for the implementation one warp per matrix/vector with matrix A in shared memory.

On table 13 and on figure 10 the results are presented from the NVIDIA Visual Profiler for the implementation with vector **r** loaded in shared memory.

Variable	Achieved	Theoretical	Device Limit
Occupancy Per SM			
Active Blocks		6	32
Active Warps	40.83	48	64
Active Threads		1536	2048
Occupancy	63.8%	75%	100%
Registers			
Registers/Thread		35	65536
Shared Memory			
Shared Memory/Block		2048	65536
Block Limit		32	32

Table 13. GPU Utilization for the implementation with vector r loaded in shared memory.

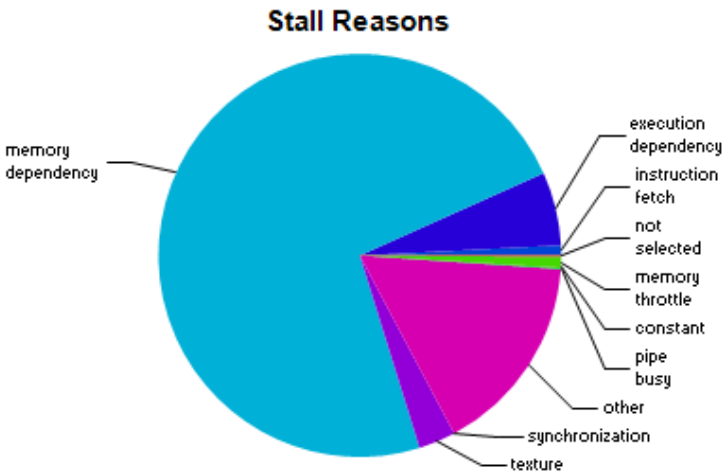

Figure 10. Stall reasons for the implementation one warp per matrix/vector with vector **r** in shared memory.

On figure 11 the plot of the results is shown for number of matrices versus time of execution for all implementations of a fully fused kernel under the thread strategy of one thread assigned per matrix/vector.

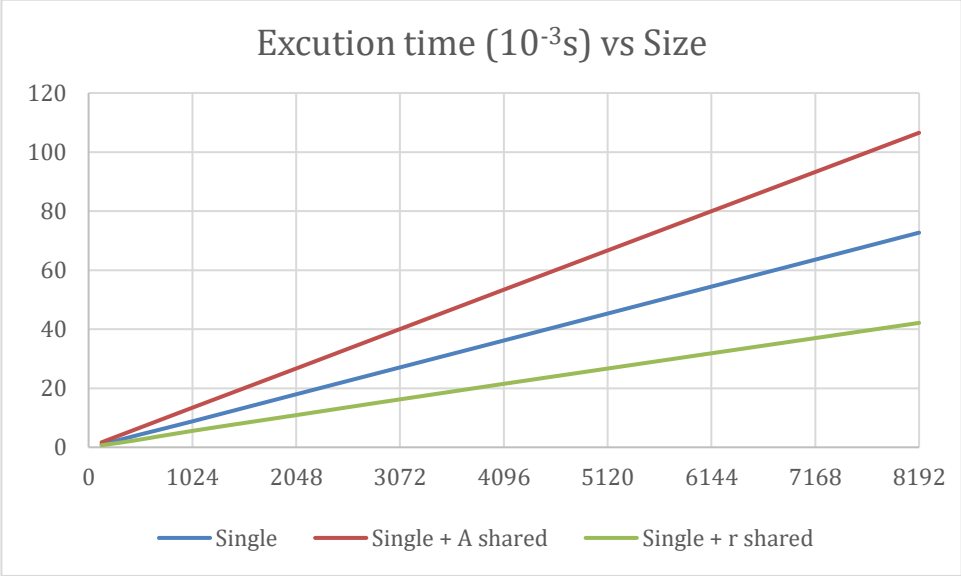

Figure 11. Plot number of matrices versus time of execution in milliseconds for the implementations where a single thread is assigned to a matrix/vector.

On figure 12 the plot of the results is shown for number of matrices versus time of execution for all implementations of a fully fused kernel under the thread strategy of assigning one warp per matrix/vector.

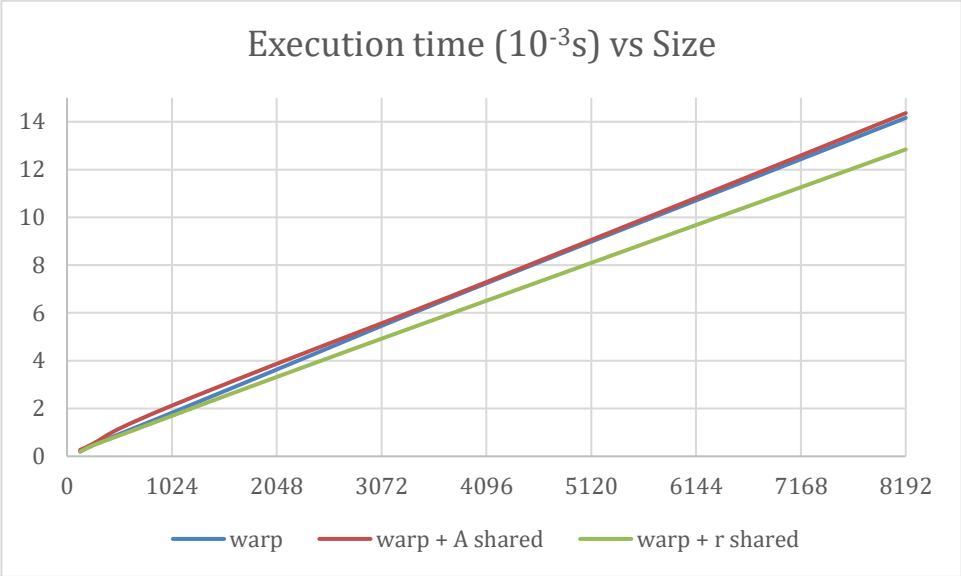

Figure 12. Plot number of matrices versus time of execution in 10^{-3} seconds for the implementations where a warp is assigned to a matrix/vector.

Results are considered only for the runtime of the CR kernel and do not include the pre-processing stage (calculation of Gram matrix and matched filter vector).

5 Conclusions

By merging kernels, the total number of read/write operations, as seen from the calculations in tables 4 and 5, can be dramatically reduced. The results of fusion can be seen in table 8 and figure 7 where the execution time starting from 42.54 milliseconds for a separate kernel implementation gradually drops to 14.16 milliseconds for a fully fused. The findings align with previous related work in [14] [37] [41].

Complete unrolling of the algorithm cuts down the overhead of the loop iteration (eliminating instructions that control the loop), allows the compiler in the background to optimize the code further, and any if-else statements or temporary variables that would be used between iterations are skipped.

Fusing multiple axpy operations increases the workload assigned to each thread. The gain is higher when the vector norm is to be computed after the vector update. The number of loads is reduced by n (n the length of the vector), while the compute intensity added from the calculation of the norm helps hiding the memory latency of the GPU.

For the matrix/vector+dot fusion, the computational intensity assigned to every thread is increased and the number of loads is reduced by 2n. The gain is higher in the case where the vector **r** is loaded into shared memory and can be fetched at higher speed. At the beginning of the first iteration the vector norm can also be added (matvec+dot+norm), the number of loads is then reduced by 3n.

For every fusion between two or more kernels, the added overhead from separate kernel launches is avoided, the number of global synchronizations, and the number of data movements from GPU to CPU is reduced.

A thread strategy of assigning one thread per matrix/vector is the simplest way of merging the algorithm in a single kernel. Results in table 9 show that this strategy is not optimal and even with the use of shared memory does not meet the massive MIMO requirements for under 10 milliseconds coherence time.

Assigning one warp per matrix/vector and making extensive use of warp shuffle commands enables total fusion and ease of control of the groups of matrix/vector that are computed. The calculation of the dot products and vector norms are done using less compiler instructions and without the need of shared memory.

The implementation without the use of shared memory uses 35 registers per thread, reaches 1536 active threads, and results in 14.16 milliseconds execution time. Shown in tables 10 and 12, the implementation with matrix **A** loaded in shared memory fits exactly eight (8) instances of matrix **A**, uses 34 registers per thread, reaches 256 active threads, and executes in 14.36 milliseconds. Execution time between these two implementations is quite close which is a hint that if shared memory could fit more

matrices, then the kernel would reach more active threads and possibly lower execution time.

Best performance is achieved for the implementation with vector **r** loaded in shared memory. Shown in tables 10 and 13, forty-eight (48) instances of vector **r** are loaded in shared memory, 35 registers per thread are used, 1536 active threads are reached, and the execution time is 12.84 milliseconds.

From the above implementations, none achieves max thread level parallelism. Limiting factors are either the size of available shared memory, case where matrix **A** is loaded in shared memory, or the number of registers per thread. For a machine with 65536 available registers and 2048 threads, the maximum number of registers per thread is 32. Register allocation explains the difference seen in calculations on table 7 and actual results in table 13. Developers do not have full control over the number of registers per thread therefore different compiler settings or versions should be tested. On the other hand, as explained in [42] max thread level parallelism does not always yield optimal performance.

Compared to work done in [8] the algorithm is fully fused into a single kernel, shared memory is used for preloading matrix **A** or vector **r**, not for sharing intermediate results of dot products or vector norms. Results cannot be compared directly since the GPUs used differ greatly, here a small laptop GPU, and further investigation using different GPUs and compiler settings is required.

The cuBLAS implementation of matrix-vector product is highly optimized, beyond the reach of the implementation in this work, but unfortunately the source code is private. It would of great interest if such an implementation could be incorporated to this work.

As proposed in [41], the algorithm is fully unrolled into a single kernel and a barrier for synchronization and data exchange while a block is active, is implemented. This was achieved using warp shuffle commands and matches the case where the number of users in massive MIMO detection is 32. Loading vector **r** into shared memory yields the best performance. Results show that kernel fusion is also beneficial for the case of massive MIMO (many relatively small matrices that must be calculated in parallel) and an approach for the Conjugate Residual algorithm is presented.

The programming model of CUDA has a programmer friendly interface, with lots of tools for visualising and debugging code, but still programming GPUs is error prone. Unrolling the algorithm and hardcoding often leads to unreadable code. Tuning kernels for memory efficiency is a complex task that degrades the productivity with most of the working hours eventually being consumed in debugging. All these could also possibly explain why the authors in previous works decided to combine generic kernels from libraries with custom kernel implementations.

Future work could be the fusion of the preconditioned CR algorithm. Performance gains by using techniques presented in [43] for the multiplication of a matrix by its transpose and the affect it would have on the numerical stability of the algorithm could be explored. The kernel fusion could then begin at the pre-processing stage of the equalizer. Great attention has been drawn on implementations using mixed precision and half precision data types for iterative solvers [44]. A half precision

implementation could be tested for further performance gains and how much it affects the numerical stability of the algorithm.

Disclaimer

This thesis work is available to all readers and researchers for study without regard to race, color, religion, sex, sexual orientation, marital status, pregnancy, parental status, national origin, ethnic background, age, disability, political opinion, social status, veteran status, union membership or genetics.

The contents of these pages are provided as an information guide only. They are intended to enrich information regarding the subject matter covered. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. The author shall not be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

Reasonable efforts have been made to publish reliable data and information, but the author cannot assume responsibility for the validity of all materials or the consequences of their use. No responsibility is accepted by or on behalf of the author for any errors, omissions or misleading statements on these pages or any site to which these pages connect.

If any copyright material has not been acknowledged, please contact so it may be rectified in the future.

Notation

Upper-case boldface letters are used to denote matrices (e.g., **X, Y**), while column or row vectors are denoted with lower-case boldface letters (e.g., **x, y**). Scalars are denoted by lower/upper-case italic letters (e.g., *x, y, X, Y*) and sets by calligraphic letters (e.g., X, Y).
The following mathematical notations are used:

$\mathbb{R}, \mathbb{R}^{n}, \mathbb{R}^{m \times n}$	Set of real numbers, n-vectors, m × n matrices
$\mathbb{C}, \mathbb{C}^{n}, \mathbb{C}^{m \times n}$	Set of complex numbers, n-vectors, m × n matrices
$x \in S$	x is a member of the set S
$\mathbf{x_i}$	The ith element of a vector x
$\mathbf{A_{ij}}$	The (i, j)th element of a matrix A
$\mathbf{A}^{*}$	The complex conjugate of A
$\mathbf{A}^{\mathbf{T}}$	The transpose of A
$\mathbf{A}^{\mathbf{H}}$	The conjugate transpose of A
$\mathbf{A}^{-1}$	The inverse of a square matrix A
$\Re(x)$	Real part of x
$\Im(x)$	Imaginary part of x
$\|\|x\|\|^2$	The L2-norm
$\mathbf{xy}$	Dot product of vectors x and y
$\mathbf{I}_{MM}$	The M × M identity matrix

Acknowledgments

I would like to thank Tiago Fernandes Cortinhal for his continuous support and encouragement during my thesis work.

6 Bibliography

[1] E. Björnson, J. Hoydis och S. L., Massive MIMO Networks: Spectral, Energy, and Hardware Efficiency, Publishers Inc., 2017.

[2] V. Madisetti, Wireless, Networking, Radar, Sensor Array Processing, and Nonlinear Signal Processing, CRC Press, 2018.

[3] S. Malkowsky, ”Massive MIMO: Prototyping, Proof-of-Concept and Implementation,” Doctoral dissertation, University of Lund, 2019.

[4] A. Paulraj, R. Nabar och D. Gore, Introduction to space-time wireless communications, 2003: Cambridge university press.

[5] L. Liu, G. Peng och S. Wei, Massive MIMO Detection Algorithm and VLSI Architecture, Springer Singapore, 2019.

[6] R. Couturier, Designing scientific applications on GPUs, CRC Press, 2013.

[7] M. Wu, C. Dick, J. R. Cavallaro och C. Studer, ”FPGA design of a coordinate descent data detector for large-scale MU-MIMO,” *2016 IEEE International Symposium on Circuits and Systems (ISCAS),* pp. 1894-1897, 2016.

[8] K. Li, B. Yin, M. Wu, J. R. Cavallaro och C. Studer, ”Accelerating massive MIMO uplink detection on GPU for SDR systems,” *IEEE dallas circuits and systems conference (DCAS),* pp. 1-4, 2015.

[9] C. Tarver, M. Tonnemacher, H. Chen, J. C. Zhang och J. R. Cavallaro, ”GPU-based LDPC decoding for vRAN systems in 5G and beyond,” *IEEE International Symposium on Circuits and Systems (ISCAS),* pp. 1-5, October 2020.

[10] B. Yin, ”Low complexity detection and precoding for massive MIMO systems: Algorithm, architecture, and application.,” Doctoral dissertation, 2014.

[11] C. Zhang, Y. Yang, S. Zhang, Z. Zhang och X. You, ”Residual-Based Detections and Unified Architecture for Massive MIMO,” *Journal of Signal Processing Systems,* vol. 91, nr 9, pp. 1039-1052, 2019.

[12] M. M. Dehnavi, D. M. Fernández och D. Giannacopoulos, ”Enhancing the performance of conjugate gradient solvers on graphic processing units,” *IEEE Transactions on Magnetics,* vol. 5, nr 47, pp. 1162-1165, 2011.

[13] J. Filipovič, M. Madzin, J. Fousek och L. Matyska, ”Optimizing CUDA code by kernel fusion: application on BLAS.,” *The Journal of Supercomputing,* vol. 10, nr 71, pp. 3934-3957, 2015.

[14] H. Anzt, S. Tomov, P. Luszczek, W. Sawyer och J. Dongarra, ”Acceleration of GPU-based Krylov solvers via data transfer reduction,” *The International Journal of High Performance Computing Applications,* vol. 3, nr 29, pp. 366-383, 2015.

[15] Y. Huang, S. Li, Y. T. Hou och W. Lou, ”GPF: A GPU-based Design to Achieve~ 100 μs Scheduling for 5G NR,” *Proceedings of the 24th Annual International*

Conference on Mobile Computing and Networking, pp. 207-222, 2018.

[16] P. P. Campo, V. Lampu, A. Meirhaeghe, J. Boutellier, L. Anttila och M. Valkama, ”Digital predistortion for 5G small cell: GPU implementation and RF measurements,” *Journal of Signal Processing Systems,* pp. 1-12, 2019.

[17] C. Tarver, A. Singhal och J. R. Cavallaro, ”GPU-Based Linearization of MIMO Arrays,” *IEEE Workshop on Signal Processing Systems (SiPS),* pp. 1-5, October 2020.

[18] K. Li, A. Ghazi, C. Tarver, J. Boutellier, M. Abdelaziz, L. Anttila, M. Juntti, M. Valkama och J. R. Cavallaro, ”Parallel digital predistortion design on mobile GPU and embedded multicore CPU for mobile transmitters,” *Journal of Signal Processing Systems,* vol. 89, nr 3, pp. 417-430, 2017.

[19] F. L. Luo och C. J. Zhang, Signal Processing for 5G: Algorithms and Implementations, John Wiley & Sons, 2016.

[20] H. Malepati, Digital media processing: DSP algorithms using C, Newnes, 2010.

[21] L. J. Hennessy och A. D. Patterson, Computer Architecture A Quantitative Approach Fifth Edition, Elsevier Inc., 2012.

[22] J. Cheng, M. Grossman och T. McKercher, Professional CUDA C Programming, John Wiley & Sons Inc., 2014.

[23] D. Storti och M. Yurtoglu, CUDA for engineers: an introduction to high-performance parallel computing, Addison-Wesley Professional, 2015.

[24] NVIDIA, ”cuBLAS,” [Online]. Available: https://developer.nvidia.com/cublas.

[25] NVIDIA, ”cuFFT,” [Online]. Available: https://developer.nvidia.com/cufft.

[26] NVIDIA, ”cuSOLVER,” [Online]. Available: https://developer.nvidia.com/cusolver.

[27] C. R. Sánchez, A. M. V. Maciá och A. G. Salvador, ”Efficient soft-output detectors: Multi-core and GPU implementations in MIMOPack library,” *International Conference on Pervasive and Embedded Computing and Communication Systems (PECCS),* pp. 1-10, 2015.

[28] A. Li, B. Zheng, G. Pekhimenko och F. Long, ”Automatic horizontal fusion for GPU kernels,” *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO),* pp. 14-27, April 2022.

[29] M. Geshi, The Art of High Performance Computing for Computational Science, Vol. 1, Springer, 2019.

[30] T. Soyata, GPU parallel program development using CUDA, CRC Press, 2018.

[31] W. H. Press, S. A. Teukolsky, W. T. Vetterling och B. P. Flannery, Numerical Recipes The Art of Scientific Computing Third Edition, Cambridge University Press, 2007.

[32] D. Bertaccini och F. Durastante, Iterative methods and preconditioning for large and sparse linear systems with applications, CRC Press, 2018.

[33] S. Hashima och M. Osamu, ”Fast matrix inversion methods based on Chebyshev

and Newton iterations for zero forcing precoding in massive MIMO systems," *EURASIP Journal on Wireless Communications and Networking*, nr 1, pp. 1-12, 2020.

[34] M. A. Albreem, M. H. Alsharif och S. Kim, "A robust hybrid iterative linear detector for massive MIMO uplink systems," *Symmetry*, vol. 2, nr 12, p. 306, 2020.

[35] H. Prabhu, "Hardware implementation of baseband processing for massive MIMO," Lund University, Doctoral dissertation, 2017.

[36] Y. Saad, Iterative methods for sparse linear systems, Society for Industrial and Applied Mathematics, 2003.

[37] G. Wang, Y. Lin och Y. Wei, "Kernel Fusion : an Effective Method for Better Power Efficiency on Multithreaded," *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pp. 344-350.

[38] A. Li, B. Zheng, G. Pekhimenko och F. Long, "Automatic horizontal fusion for GPU kernels," *arXiv preprint arXiv:2007.01277*, 2020.

[39] R. Nath, S. Tomov, T. T. Dong och J. Dongarra, "Optimizing symmetric dense matrix-vector multiplication on GPUs.," *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1-10, 2011.

[40] B. Crovella, "Oak Ridge National Laboratory," [Online]. Available: https://www.olcf.ornl.gov/wp-content/uploads/2019/12/05_Atomics_Reductions_Warp_Shuffle.pdf. [Använd May 2022].

[41] S. Tarsa, T. H. Lin och H. Kung, "Performance Gains in Conjugate Gradient Computation with Linearly Connected GPU Multiprocessors," *USENIX HotPar*, nr 12, 2012.

[42] X. Xie, Y. Liang, X. Li, Y. Wu, G. Sun, T. Wang och D. Fan, "CRAT: Enabling coordinated register allocation and thread-level parallelism optimization for GPUs.," *IEEE Transactions on Computers*, vol. 67, nr 6, pp. 890-897, 2017.

[43] V. Arrigoni, F. Maggioli, A. Massini och E. Rodolà, "Efficiently Parallelizable Strassen-Based Multiplication of a Matrix by its Transpose," i *50th International Conference on Parallel Processing*, 2021.

[44] A. e. a. Abdelfattah, "A survey of numerical linear algebra methods utilizing mixed-precision arithmetic," *The International Journal of High Performance Computing Applications*, vol. 35, nr 4, pp. 344-369, 2021.

[45] T. F. Collins, R. Getz, D. Pu och A. M. Wyglinski, Software-Defined Radio for Engineers, Artech House, 2018.

[46] D. B. Kirk och W. H. Wen-Mei, Programming Massively Parallel Processors A Hands-on Approach Second Edition, Morgan Kaufmann, 2016.

[47] J. R. Shewchuk, An introduction to the conjugate gradient method without the agonizing pain, 1994.