RICE UNIVERSITY

GPU Accelerated Reconfigurable Detector and Precoder for Massive MIMO SDR Systems

by

Kaipeng Li

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Joseph Cavallaro, Chair
Professor of Electrical and Computer Engineering and Computer Science

Behnaam Aazhang
J.S Abercrombie Professor of Electrical and Computer Engineering

Lin Zhong
Associate Professor of Electrical and Computer Engineering and Computer Science

HOUSTON, TEXAS
DECEMBER 2015
RICE UNIVERSITY

GPU Accelerated Reconfigurable Detector and Precoder for Massive MIMO SDR Systems

by

Kaipeng Li

A Thesis Submitted
in Partial Fulfillment of the Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Joseph Cavallaro, Chair
Professor of Electrical and Computer Engineering and Computer Science

Behnaam Aazhang
J.S. Abercrombie Professor of Electrical and Computer Engineering

Lin Zhong
Associate Professor of Electrical and Computer Engineering and Computer Science

Houston, Texas
December 2015
We present a reconfigurable GPU-based unified detector and precoder for massive MIMO software-defined radio systems. To enable high throughput, we implement the linear minimum mean square error detector/precoder and further reduce the algorithm complexity by numerical approximation without sacrificing the error-rate performance. For efficient GPU implementation, we explore the algorithm’s inherent parallelism and take advantage of the GPU’s numerous computing cores and hierarchical memories for the optimization of kernel computations. We furthermore perform multi-stream scheduling and multi-GPU workload deployment to pipeline multiple detection or precoding tasks on GPU streams for the reduction of host-device memory copy overhead. The flexible design supports both detection and precoding and can switch between Cholesky based mode and efficient conjugate gradient (ECG) based mode for accuracy and complexity tradeoff. The GPU implementation exceeds 250 Mb/s detection and precoding throughput for a 128×16 antenna system.
ACKNOWLEDGEMENTS

I would like to thank my advisor Prof. Joseph Cavallaro for his thoughtful guidance and continuous support during my thesis work. I would also like to thank other committee members: Prof. Behnaam Aazhang, Prof. Lin Zhong for their insightful suggestions and comments. I would also like to thank my families for their support and encouragement, and thank my teammates and friends for their favorable help: Guohui Wang, Yin Bei, Michael Wu, Christoph Studer, Chance Tarver and Aida Vosoughi.

This work was supported in part by the US National Science Foundation under grants CNS-1265332, ECCS-1232274, ECCS-1408370, and ECCS-1408006.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>iii</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Contribution</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Thesis Outline</td>
<td>3</td>
</tr>
<tr>
<td><strong>2 Background and Related Work</strong></td>
<td>5</td>
</tr>
<tr>
<td>2.1 Software-defined Radio Systems</td>
<td>5</td>
</tr>
<tr>
<td>2.2 MIMO Detection and Precoding</td>
<td>7</td>
</tr>
<tr>
<td>2.3 General Purpose Computing on GPU</td>
<td>13</td>
</tr>
<tr>
<td><strong>3 Low Complexity Detection and Precoding Algorithms</strong></td>
<td>19</td>
</tr>
<tr>
<td>3.1 Cholesky-based Detection and Precoding</td>
<td>19</td>
</tr>
<tr>
<td>3.2 Algorithm Complexity Reduction by Efficient Conjugate Gradient (ECG) Method</td>
<td>21</td>
</tr>
<tr>
<td>3.3 Block Error Rate Performance</td>
<td>24</td>
</tr>
<tr>
<td>3.4 Complexity Analysis</td>
<td>32</td>
</tr>
<tr>
<td>3.5 BLER Performance Comparison</td>
<td>34</td>
</tr>
<tr>
<td>3.6 Arithmetic Flow of the Unified Design</td>
<td>36</td>
</tr>
<tr>
<td><strong>4 Design Implementation and Optimization on GPU</strong></td>
<td>38</td>
</tr>
<tr>
<td>4.1 Reconfigurable Unified Architecture</td>
<td>38</td>
</tr>
</tbody>
</table>
## List of Figures

2.1 MIMO-OFDM Uplink Model ........................................... 9  
2.2 MIMO-OFDM Downlink Model ........................................ 10  
2.3 GPU Architecture ...................................................... 15  
2.4 GPU Programming Model ................................................ 18  
3.1 BLER Performance at U=16 ........................................... 26  
3.2 BLER Performance at U=32 ........................................... 27  
3.3 BLER Performance at U=64 ........................................... 28  
3.4 BLER Performance of $68(B) \times 4(U)$ Experimental Setup by Argos .......................................................... 30  
3.5 BLER Performance of $84(B) \times 3(U)$ Experimental Setup by Argos .......................................................... 31  
3.6 Complexity Comparison Between ECG-D and Chol-D .............. 32  
3.7 Reduction Rate of ECG-D Compared to Chol-D ...................... 34  
3.8 BLER Comparison Between MMSE-P(exact and approx.) and MF-P .......................................................... 35  
4.1 Unified and Reconfigurable Architecture ............................. 39  
4.2 Reduction to Shared Memory by Atomic Operations ............... 42  
4.3 Multi-stream Scheduling ............................................... 46  
4.4 Multi-GPU Wordload Deployment ..................................... 48  
5.1 Throughput/Latency Tradeoff ........................................... 53  
5.2 Speedup Rate: GPU VS Multicore CPU ............................... 54  
5.3 Effective Capacity Comparison of MF and MMSE precoders for $128 \times 16$ System ................................................... 57
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>GPU architecture comparison</td>
<td>17</td>
</tr>
<tr>
<td>3.1</td>
<td>Minimum CG iterations for $&lt;1$dB SNR loss at $10^{-2}$ BLER</td>
<td>33</td>
</tr>
<tr>
<td>5.1</td>
<td>Single Stream Throughput Performance of Detector</td>
<td>51</td>
</tr>
<tr>
<td>5.2</td>
<td>Multi-stream &amp; Multi-GPU Enhanced Performance of Detector</td>
<td>51</td>
</tr>
<tr>
<td>5.3</td>
<td>Multi-stream &amp; Multi-GPU Enhanced Performance of Precoder</td>
<td>52</td>
</tr>
<tr>
<td>5.4</td>
<td>Performance Comparison with Previous Work</td>
<td>54</td>
</tr>
<tr>
<td>5.5</td>
<td>Parameters of Systematical Performance Model</td>
<td>56</td>
</tr>
</tbody>
</table>
1.1 Motivation

Massive multiple-input multiple-output (MIMO) is believed to be a key technology for future 5G wireless systems[1][2][3]. By equipping the base station (BS) with hundreds of antennas while serving tens of users, improved spectral efficiency and link reliability compared to small-scale MIMO systems can be obtained. Massive MIMO, however, entails high baseband processing complexity when the antenna number goes to hundreds, thousands, even to infinity [4]. In particular, uplink data detection, which demultiplexes the spatial data streams at the BS[5], and downlink beamforming (precoding), which allows the BS to transmit data streams independently to multiple terminals within the same resource block, are among the most computationally intensive tasks. Due to the prohibitive complexity of optimal data detection and precoding methods, sub-optimal low complexity detectors and precoders, such as the linear minimum mean square error (MMSE) data detector and precoder, must be used in practice to achieve sufficiently high throughput at reasonable hardware costs.

To enable high throughput data detection and precoding for massive MIMO, recent FPGA and ASIC results in [6][7][8][9] use a novel low-complexity Neumann series
(NS) data-detection algorithm for the LTE-A uplink. While corresponding hardware implementations achieve high throughput, they offer only limited flexibility with respect to antenna configurations and performance/complexity trade-offs. As an alternative, general purpose computing on graphics processing unit (GPGPU) technology is shown to offer both high performance and high degrees of reconfigurability for accelerating most complex baseband algorithms, such as LDPC decoding [10] or Turbo decoding [11]. Currently, most previous work on GPU accelerated baseband modules targets single-input-single-output (SISO) systems or small scale MIMO systems to realize low design efforts and high design flexibility in the context of software-defined radio (SDR) systems. However, not much is known about the efficacy of GPGPU for serving as a centralized baseband accelerator in massive MIMO SDR systems. With the cutting edge architecture and increasing computing power of GPU, it is worth exploring the design space of a GPU-based massive MIMO SDR system to offer both high data rate and high design flexibility for potential applications in the coming 5G wireless systems. Specifically, data detection and precoding are among the most representative baseband modules in terms of computational complexity in a massive MIMO basestation, so how to design and implement an efficient and flexible GPU-based detector and precoder is a key step of realizing such a massive MIMO SDR system.

1.2 Contribution

In this thesis, we present an unified GPU-accelerated reconfigurable uplink data detector and downlink precoder for massive MIMO SDR systems, which can achieve both high data throughput and design flexibility. This work—to the best of the author’s knowledge—is the first GPU implementation of detector and precoder targeting massive MIMO systems.
To obtain high data throughput, both low complexity algorithms and efficient implementation are required. First, we use efficient conjugate gradient (ECG) based methods to reduce the computational complexity of the linear minimum mean square error (MMSE) detection and precoding algorithm without losing much error-rate performance. When mapping our low-complexity algorithms on GPU, we judiciously perform various GPGPU optimization strategies on kernel design, memory utilization, and task scheduling to improve the processing efficiency and reduce the overhead. By running our design on two GPU devices concurrently, we achieve over 250 Mb/s detection and precoding throughput for a 128×16 antenna massive MIMO SDR system.

With regard to flexibility, we design a reconfigurable architecture with mode selection input signals and software-defined kernel function parameters. Our unified design can switch the functionality between uplink detection and downlink precoding. Both the detector and precoder designs can support various BS and user antenna configurations, and can tradeoff between accuracy and complexity by supporting two modes: a high-accuracy Cholesky-based mode and a high-throughput ECG-based mode. One can easily configure the design according to these requirements on error-rate performance and data-rate performance.

Results from this thesis are recently published and in press[12][13][14].

1.3 Thesis Outline

Chapter 2 discusses necessary backgrounds and related work, including software-defined radio systems, MIMO detection and precoding algorithms, GPU hardware architecture and programming models. Chapter 3 illustrates the efficient conjugate gradient-based low complexity detection and precoding algorithms, focusing on how to realize the accuracy and complexity tradeoff based on organized simulation results. We show the architecture, design details and optimization strategies of our GPU-
based unified detector/precoder design in Chapter 4 and the latency and throughput performance evaluation results in Chapter 5. Chapter 6 concludes the thesis work and proposes future work.
2.1 Software-defined Radio Systems

Software-defined radio (SDR) systems have developed rapidly in recent years with the ever-growing needs of low cost, high flexibility and high efficiency design for next generation wireless communication systems. By enabling some or all of the physical layer functions for a radio in a fully programmable way, SDR systems can be easily adapted to a variety of wireless communication standards and specific user requirements by simply reconfiguring the software-defined system parameters, which allows researchers to fast prototype and test more powerful baseband algorithms for future wireless communication systems and standards with lower development cost. SDR systems are believed to be the basis of evolution to future cognitive radio systems considering the high adaptability and flexibility of SDRs.

In SDR systems, the baseband processing is usually accelerated by modifiable or programmable hardware, such as field programmable gate arrays (FPGA), general purpose processors (GPP) such as CPUs, digital signal processors (DSP), or programmable System on Chip (SoC), etc. Some FPGA-based SDRs systems, such as 802.11 reference design on wireless open-access research platform (WARP) developed
by Rice University, are able to achieve real-time baseband processing with the high computing capability of FPGA. There are also some GPP-based SDR systems, such as GNU radio[15], Microsoft Sora[16] platform and WARPLab[17], enabling more flexible baseband design by high level programming language such as C or Matlab and efficient data communication with front end platforms such as USRP, WARP or Sora radios. Recently, general purpose computing on GPU (GPGPU) technology has been seen an another alternative of baseband accelerator considering the GPU’s TFLOPS level peak computing capability and high programmability by CUDA or OpenCL[18]. Some GPU-based baseband components or SDR systems have also been developed and evaluated, showing the capability and efficiency of using GPU as baseband accelerators.

As we know, FPGAs are constructed by low level gates and multipliers which can capture the data path efficiently. With the pre-programmed property, the overhead of instruction fetching and decoding can be neglected in FPGA, in contrast to CPU or GPU. Furthermore, the memory resources on FPGA are placed close to the modules which need them. All above architectural properties contribute to the low latency and high performance of FPGA. Relatively, GPUs have some different features. For example, the instruction fetching and decoding on GPU can introduce extra overhead, and the memory system on GPU is more restrictive than FPGA. Also, the computation tasks on GPU should have high degree of inherent parallelism, and can be divided into independent small portions so that they can be processed on thousands of GPU’s simple computing cores for improving the total throughput. Besides, GPUs are good at floating point computations rather than fixed point computations, which can offer higher precision. Another key advantage of GPU is its high programmability, which enables lower design efforts in the context of SDR systems. Compared to conventional high performance FPGA-based SDRs which may suffer from high design cost and long
design period, or flexible GPP-based SDRs which may have limited performance on computationally intensive workloads, GPU-based SDRs have the potential to strike the balance between the design flexibility and system data-rate performance for fast and real-time prototyping of a variety of baseband components which have inherent arithmetic parallelism and can be mapped in an architecture-aware manner.

### 2.2 MIMO Detection and Precoding

#### 2.2.1 Large Scale MIMO Systems

Multiple-input multiple-output (MIMO) technology was proposed in recent years to increase the data rate of wireless communication by exploring the communication resources at the spacial dimension within the same time and frequency band\[19\][20]. MIMO technology has already been widely adopted in many modern wireless standards, such as IEEE 802.11n\[21\][22] and 3GPP LTE-A\[23\][24][25], etc. By equipping many-antenna array, the basestation can process the array signals coherently to stream and focus the transmitted signal to specific mobile terminals or recover the received signal of multiple users from interference and noise, by performing downlink precoding or uplink data detection, respectively. More antenna numbers at BS can result in finer spatial focusing and signal recovery. Compared to small-scale MIMO systems, large-scale MIMO (or massive MIMO) systems, where the BS is equipped with hundreds of antennas to serve tens of users, can further improve the spectrum efficiency, channel capacity and link reliability\[26\][27]. Several previous massive MIMO testbeds have been developed to show the feasibility and capability of massive MIMO system, such as Rice Argos\[28\][29][30][31] built by WARP boards, NI massive MIMO SDR system\[32\], LUND basestation by USRP\[33\] and Microsoft BigStation based on CPU clusters\[34\].
Orthogonal frequency-division multiplexing (OFDM) is another widely adopted technology in modern standards. By performing OFDM, the signals can be divided and mapped on to different subcarriers, and transmitted over their corresponding narrow frequency band to overcome the intersymbol interference (ISI) and fading due to multipath propagation. FFT and IFFT can be used to transform the baseband signals between time and frequency domain for efficient implementation of OFDM system.

MIMO-OFDM systems can take advantage of both MIMO and OFDM technologies[35]. We describe the uplink and downlink MIMO-OFDM system models which are used in later Chapters in the following part.

In an uplink OFDM system, as shown in Fig. 2.1, we consider a basestation equipped with \( B \) antennas, which serves \( U \leq B \) single antenna users. Each user encodes the information bits and maps the encoded bits onto constellation points in a finite alphabet \( \Omega \). The frequency domain modulated OFDM symbols are then transformed to time domain symbols by IDFT and transmitted over the wireless channel. At the receiver side, the basestation uses DFT to transform back the received time domain symbols to frequency domain for the data detection and decoding stages. Define \( y_{b,k} \) as the transformed frequency domain sample of the \( b^{th} \) antenna and \( k^{th} \) subcarrier at BS, and \( x_{u,k} \) as the transmitted frequency domain sample from the \( u^{th} \) user and \( k^{th} \) subcarrier, then:

\[
y_k = \mathbf{H}_k x_k + n_k,
\]

for the \( k^{th} \) subcarrier, where \( y_k \in \mathbb{C}^B \) is a vector constructed as \([y_{1,k}, y_{2,k}, \ldots, y_{B,k}]^T\), \( x_k \in \Omega^U \) is a vector \([x_{1,k}, x_{2,k}, \ldots, x_{U,k}]^T\), \( \mathbf{H}_k \in \mathbb{C}^{B \times U} \) is the \( B \times U \) complex channel matrix and \( n_k \in \mathbb{C}^B \) is the noise vector \([n_{1,k}, n_{2,k}, \ldots, n_{B,k}]^T\) with each entry \( n_{b,k} \) assumed to be an i.i.d zero-mean complex Gaussian random variable with variance \( N_0 \). The
uplink transmit power for each user is denoted as $E_s$, so the average uplink SNR can be computed as $UE_s/N_0$.

In a downlink OFDM system, as shown in Fig. 2.2., the basestation serves as transmitter, encoding the information bits and mapping the encoded bits onto constellation points in a finite alphabet $\Omega$ to form the transmit vector $j_k \in \Omega^U$ for $k^{th}$ subcarrier of all users. The transmit vector $j_k$ will then be filtered by the precoding matrix $P_k \in \mathbb{C}^B \times U$ as

$$m_k = P_k j_k,$$  \hspace{1cm} (2.2)

where $m_k \in \mathbb{C}^B$ is the precoded vector and needs to be further normalized as

$$x'_k = E_s m_k / \|m_k\|,$$  \hspace{1cm} (2.3)

where $x'_k \in \mathbb{C}^B$ is the precoded and normalized transmit vector in the frequency domain. Similar to uplink system, we can form the input-output relationship of the
downlink channel $\mathbf{H}'_k$ as

$$y'_k = \mathbf{H}'_k x'_k + n'_k,$$

where $y'_k \in \mathbb{C}^U$ is the receive vector with each element corresponding to a certain user for $k^{th}$ subcarrier. When the system operates in time-division duplex (TDD) mode, the uplink and downlink channels are assumed to be nearly reciprocal, that is $\mathbf{H}'_k = \mathbf{H}_k^H$, where $(\cdot)^H$ indicates the conjugate transpose of a matrix.

### 2.2.2 Uplink Detection

In the uplink MIMO system, the basestation receives the frequency domain baseband signal $y_k$ as denoted in Eq. (2.1). To unmix the received data streams from different users and recover the transmit signal $x_k$, data detection process is required. There are different types of data detection methods for MIMO systems with different arithmetic complexity and detection performance. The maximum-likelihood (ML) detector achieves optimal detection performance, but entails prohibitive computational com-
plexity for efficient hardware implementation with the increase of antenna numbers, especially in the scenario of massive MIMO systems. Some sub-optimal detectors, such as sphere detection and decoding scheme, are thus adopted in small-scale MIMO systems, but are still impractical for massive MIMO systems. Linear detectors, such as matched filter (MF), zero-forcing (ZF), or minimum mean square error (MMSE) detectors, are believed to be promising candidates for massive MIMO detection considering their manageable complexity for practical implementation. Compared to MF and ZF detectors, linear MMSE detector can strike better balance between suppressing noise and suppressing interference in received signals.

In this thesis, we focus on the linear soft-output MMSE detection targeting massive MIMO scenarios. The soft-output detector can not only equalize the received signal $y_k$, but also can generate the soft information denoted as log-likelihood ratios (LLRs) of each information bit for each user. The soft-output of the detector can be then fed into the following soft-input channel decoder to realize an iterative detection and decoding scheme to achieve better system error-rate performance.

To estimate the transmit signal $x_k$, the MMSE equalizer gives $\hat{x}_k = W_k y_k$ where $W_k$ is the equalization matrix:[36]:

$$W_k = (H_k^H H_k + (E_s/N_0)^{-1} I_U)^{-1} H_k^H.$$

(2.5)

Here, $H_k^H$ is the conjugate transpose of channel matrix $H_k$, and $I_U$ is the $U \times U$ identity matrix. To avoid redundant computations [37], we define the Gram matrix $G_k = H_k^H H_k$, regularized Gram matrix $A_k = G_k + (E_s/N_0)^{-1} I_U$ and matched-filter output $y_k^{MF} = H_k^H y_k$, then:

$$\hat{x}_k = A_k^{-1} y_k^{MF}.$$

(2.6)

For the computation of LLR, we model the estimated transmit sample for the $u^{th}$
user as \( x_{u,k} = \lambda_{u,k} x_{u,k} + e_{u,k} \), where \( \lambda_{u,k} = w_{u,k}^H h_{u,k} = \text{diag}(A_k^{-1} G_k)_u \) is the effective channel gain, and \( e_{u,k} = \sum_{q \neq u} w_{u,k}^H h_{q,k} x_{q,k} + w_{u,k}^H n_k \) is the post-equalization noise-plus-interference (NPI) with variance \([6]\):

\[
v_{u,k}^2 = \mathbb{E}(|e_{u,k}|^2) \approx E_s \lambda_{u,k} - E_s |\lambda_{u,k}|^2.
\] (2.7)

Here, \( w_{u,k}^H \) and \( h_{q,k} \) are the \( u \)th row of \( W_k \) and the \( u \)th column of \( H_k \), respectively, and \( \text{diag}(\cdot)_u \) indicates the \( u \)th diagonal entry of a matrix. The max-log approximation of LLR of bit \( p \) for a certain \( x_{u,k} \) can be computed as follows with low complexity \([37]\):

\[
L_p^{u,k} = \rho_{u,k} \left( \min_{a \in \Omega_0^p} \left| \frac{x_{u,k}}{\lambda_{u,k}} - a \right|^2 - \min_{a' \in \Omega_1^p} \left| \frac{x_{u,k}}{\lambda_{u,k}} - a' \right|^2 \right).
\] (2.8)

Here, \( \rho_{u,k} \) is the post-equalization signal-to-interference-plus-noise ratio (SINR) represented as:

\[
\rho_{u,k} = \frac{\lambda_{u,k}^2}{v_{u,k}^2},
\] (2.9)

and \( \Omega_0^p \) and \( \Omega_1^p \) indicate the sets of constellation samples of which the \( p \)th bit equals to 0 and 1, respectively.

### 2.2.3 Downlink Precoding

In the downlink MIMO system, the basestation precodes the transmit constellation symbols as denoted by Eq. (2.2) so that the precoded and normalized transmit signal \( x_k' \) can reach the desired mobile terminal with finer spatial focusing. Similar to uplink detection, linear precoders, such as MF, ZF, and MMSE precoder, have better potential to be adopted in massive MIMO systems considering their good performance and acceptable complexity.

In this thesis, we focus on the linear MMSE precoder, which can maximize the
signal-to-interference-plus-noise-ratio (SINR) and usually outperforms MF and ZF in most situations. The precoding matrix $P_k$ of linear MMSE precoder can be formulated as

$$P_k = H_k^H (H_k H_k^H + (E_s/N_0)^{-1} I_U)^{-1} = H_k^H A_k^{-1},$$

where $A_k = H_k H_k^H + (E_s/N_0)^{-1} I_U$ is the regularized Gram matrix for downlink precoding.

From the above analysis, we can find that the dominant computational complexity of both uplink detection and downlink precoding comes from the matrix inversion $A_k^{-1}$ or $A_k'^{-1}$, which requires $O(U^3)$ complexity.

### 2.3 General Purpose Computing on GPU

Graphics processing unit (GPU) is equipped in most commercial computers and mobile devices. While the initial usage of GPU is to offer high computing power for accelerating graphics applications, in recent years, the numerous computing resources of modern GPUs are better exploited and generalized for some general purpose computing problems such as digital signal processing, molecular dynamics simulations, machine learning and data mining, and so on. General purpose computing on GPU (GPGPU) has become a new trend and hot topic in high performance computing (HPC) and parallel computing area[38]. For wireless communication applications, some previous work presents the efficacy of using GPU as baseband accelerators to accelerate some computationally intensive baseband modules, such as LDPC decoding[10], Turbo decoding[11], and small scale MIMO detectors[39], etc. Thousands of parallel computing cores and hierarchical memory resources in modern GPUs are able to provide TFLOPS peak computing power for a variety of computationally intensive problems, but to optimize the acceleration performance on GPU requires
both careful exploration of inherent parallelism and efficient architecture-aware mapping of specific algorithms.

2.3.1 GPU Hardware Architecture

We introduce the architecture of modern GPUs in this section. Without losing generality, we discuss the GPU devices produced by Nvidia company, which leads the trend in GPU computing, as case studies. In 2006, Nvidia released G80 GPU and GTX 8800 card, bringing the Compute Unified Device Architecture (CUDA) and better normalization of GPGPU to the world. During the past decade, the Nvidia GPU architecture has experienced several generations, from Fermi, to Kepler, and to the latest Maxwell architecture, significantly improving the computing performance and power efficiency.
Fig. 2.3 shows the detailed hardware architecture of a latest GM204 GPU[40],

Figure 2.3: GPU Architecture
which is manufactured with Maxwell architecture and integrated in GTX 980 and GTX 970 graphics cards and can represent the general architecture hierarchy of most modern GPUs. As shown in Fig. 2.3, the GPU consists several graphics processing clusters (GPCs), each containing multiple streaming multiprocessors (SMs), with each SM composed of hundreds of computing cores. Different GPUs may have different number of SMs and computing cores. Generally speaking, with more SMs and computing cores, a GPU can achieve better peak computing power, but the computing performance can also be affected by the richness and hierarchy of memory resources on the GPU.

In a GPU card, the slowest but largest (several GBs) memory is global memory, or device memory which can directly communicate with CPU memory via PCIe bus and share the data between different SMs. Within global memory, there is a special and limited memory area called constant memory, which can be used to store some constant variables for an application and be fast broadcast when those variables need to be fetched, achieving lower memory transaction overhead than general global memory. Furthermore, L2 caches are also used to store frequently used values to reduce the memory transactions with slower global memory. Inside a SM, as shown in the zoom-in subfigure of Fig. 2.3, the shared memories and L1 caches are used to realize the efficient data sharing within a SM with higher memory bandwidth than global memory, and the local registers, which are closest to GPU cores and thus fastest, are used to store the local variables for a certain thread running on a certain core. We should also understand that although local registers and shared memories are faster, they are also a very scarce memory resource (several tens of KB in a SM), so usually optimization of the utilization of those resources is essential to obtain high performance of GPU acceleration.

In Table 2.1, we show the comparison of architectural specifications of two rep-
<table>
<thead>
<tr>
<th></th>
<th>GTX 680 (Kepler)</th>
<th>GTX 980 (Maxwell)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMs</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>CUDA cores</td>
<td>1536</td>
<td>2048</td>
</tr>
<tr>
<td>Clock Freq.</td>
<td>1006MHz/1058MHz (boosted)</td>
<td>1126MHz/1216MHz (boosted)</td>
</tr>
<tr>
<td>GFLOPS</td>
<td>3090</td>
<td>4612</td>
</tr>
<tr>
<td>Memory Clock</td>
<td>6000MHz</td>
<td>7000MHz</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>192GB/sec</td>
<td>224GB/sec</td>
</tr>
<tr>
<td>L2 Cache Size</td>
<td>512KB</td>
<td>2048KB</td>
</tr>
<tr>
<td>TDP</td>
<td>195Watts</td>
<td>165Watts</td>
</tr>
<tr>
<td>Transistors</td>
<td>3.5 billion</td>
<td>5.2 billion</td>
</tr>
<tr>
<td>Manufacture</td>
<td>28nm</td>
<td>28nm</td>
</tr>
</tbody>
</table>

2.3.2 GPU Programming Model

A key advantage of GPGPU is that we can program a GPU with high-level programming languages such as CUDA or OpenCL with mature development tools, formalized programming models and numerous libraries to achieve significant speedup of computation with low programming efforts. Considering CUDA and OpenCL have similar programming methodologies, so here we only discuss the CUDA programming model[41] as illustrated in Fig. 2.4.

The total computational workloads of a specific application usually include several computational stages, each of which can be designed as a certain kernel function to represent the computing flow and inherent parallelism of that stage. At runtime, the CPU process will invoke those kernel functions with thousands of threads to perform the computations in parallel. For a certain kernel function, those parallel threads will be grouped into several thread blocks within a whole thread grid for better thread organization. One can specify the number of the thread blocks within a grid and

representative Nvidia GPU cards[40], the GTX 680 with Kepler architecture, and the GTX 980 with Maxwell architecture.
the number of threads within a block as directives of kernel function according to the parallelism degree of the corresponding computation stage. As shown in Fig. 2.4, each thread processes an independent and small portion of the total computation workloads with related variables stored in corresponding local registers, different threads within a block can communicate by the shared memories or L1 caches, and all the threads from different blocks within a grid can share data in the global memories or L2 caches. When deploying those threads onto GPU cores to execute, every 32 threads within a block will be grouped into a *warp* as a workload unit to be dispatched onto the GPU cores by *warp* schedulers in SMs. To fully utilize thousands of GPU cores, we usually need to have enough computational workload so that we can generate enough parallel threads to keep high occupancy of GPU cores and achieve high computing throughput.
To bridge the gap between the theory and practice of massive MIMO detection and precoding, both low complexity algorithms and efficient implementations are required. In this chapter, we focus on the discussion of algorithm complexity reduction by performing numerical approximations without sacrificing much error-rate performance.

### 3.1 Cholesky-based Detection and Precoding

Both the calculation of equalization matrix $W_k$ in the uplink detection and the calculation of precoding matrix $P_k$ in the downlink precoding require a $U \times U$ matrix inversion, i.e., $A_k^{-1}$ (detection) or $A_k'^{-1}$ (precoding) according to Eq. (2.5) or Eq. (2.10), respectively, which are among the most computationally intensive parts in MMSE detection or precoding with the increase of the number of users.

To calculate $A_k^{-1}$ or $A_k'^{-1}$, a straightforward way is to perform matrix decomposition. Considering matrices $A_k$ and $A_k'$ are both symmetric and positive definite matrices, we can resort to *Cholesky* decomposition with lower complexity than other
direct matrix inversion methods such as LU or QR decomposition: we first decompose $A_k$ as $A_k = L_k L_k^H$, where $L_k$ is a lower triangular matrix with real and positive values on the diagonal; then we solve the linear equation $L_k f_u = \epsilon_u, u = 1, 2 \cdots U$ ($\epsilon_u$ is $u^{th}$ unit vector) for $f_u$ and $L_k^H z_u = f_u$ for $z_u$ via forward and backward substitution, respectively. In this way, $A_k^{-1}$ can be finally represented as $A_k^{-1} = [z_1, z_2 \cdots z_U]$.

In the Cholesky-based detection and precoding scheme, we can explicitly obtain the exact $A_k^{-1}$ or $A_k'^{-1}$, and then the exact $W_k$ or $P_k$. While the following calculation of precoding is only to get the precoded vector $x_k'$ according to Eq. (2.2) and (2.3), the detection still requires another computation stage, i.e, the soft-output computation stage, which can be performed according to Eq. (2.8) based on equalized signal $\hat{x}_k$ and $A_k^{-1}$.

The Cholesky decomposition has $O(U^3)$ complexity for explicitly calculating $A_k^{-1}$ or $A_k'^{-1}$, which dominates the complexity of the Cholesky-based detector or precoder when $U$ is large. Some previous work proposed low complexity numerical methods to approximate $A_k^{-1}$ or $A_k'^{-1}$, such as Neumann Series approximation[6] or conjugate gradient based approximation[42]. The reason that we need exact or approximate matrix inversion is that the calculation of LLR for uplink detection requires $A_k^{-1}$. Considering that such calculation of $A_k^{-1}$ or $A_k'^{-1}$ introduces extra complexity even with low complexity approximations, we propose an efficient conjugate gradient (ECG) based method which can totally avoid the matrix inversion, and can approximate LLR without the need of $A_k^{-1}$ in the following section.
3.2 Algorithm Complexity Reduction by Efficient Conjugate Gradient (ECG) Method

3.2.1 CG-based Iterative Linear Solver

In the uplink detection, the matrix inversion $A_k^{-1}$ is used for calculation of equalized signal $\hat{x}_k$ by $\hat{x}_k = A_k^{-1}y_{MF}$ according to Eq. (2.6). Actually, the calculation of $\hat{x}_k$ can be regarded as the solution of a linear equation $A_k\hat{x}_k = y_{MF}$.

In the downlink precoding, the matrix inversion $A_k'^{-1}$ is used to calculate the precoded vector $m_k$ by $m_k = H_k'^H A_k'^{-1}j_k$ according to Eq. (2.2) and (2.10). Similar to detection, we can first solve the linear equation $A_k' n_k = j_k$, where $n_k = (H_k'^H)^{-1}m_k$ to get the solution $n_k$, and then the precoded vector can be obtained by $m_k = H_k'^H n_k$.

Iterative methods are effective techniques and usually require lower complexity on solving large linear equations than direct methods. Here, we use the conjugate gradient (CG) iterative method, to solve the above linear equations $A_k\hat{x}_k = y_{MF}$ and $A_k' n_k = j_k$ without explicit matrix inversions to reduce the algorithm complexity.

Assume we have a linear equation $\Lambda \psi = b$, where $\Lambda \in \mathbb{C}^{U \times U}$ is a positive definite matrix, and $\psi \in \mathbb{C}^U$ and $b \in \mathbb{C}^U$ are vectors, the conjugate gradient method can approximate the solution $\psi$ in iterations by solving the optimization problem

$$\hat{\psi} = \arg \min_{\psi \in \mathbb{C}^U} \|b - \Lambda \hat{\psi}\|.$$

(3.1)

Given the input $\Lambda$ and $b$, and initial value of solution $\psi^{(0)} = 0$, the conjugate gradient linear solver (CGSolver) can update the approximation of solution $\psi^{(i)}$ in the $i^{th}$ iteration according to Algorithm 1.

For CGSolver, while we can perform $U$ iterations to obtain the exact solution of $\hat{\psi}$, usually in practice, we only need a smaller iteration number $I < U$ to achieve a close
Algorithm 1: CG-based iterative linear solver

1: **Init:** \( ψ^{(0)} = 0, r^{(0)} = b, t^{(0)} = r^{(0)} \)

2: **for** \( i = 1, 2, \cdots, I \) **do** /*I: CG iter. for a close approx.* /

3: \( s^{(i-1)} = Λ t^{(i-1)} \)

4: \( α^{(i)} = \|r^{(i-1)}\|^2 / ( (t^{(i-1)})^H s^{(i-1)} ) \)

5: \( ψ^{(i)} = ψ^{(i-1)} + α^{(i)} t^{(i-1)} \)

6: \( r^{(i)} = r^{(i-1)} - α^{(i)} s^{(i-1)} \)

7: \( β^{(i)} = \|r^{(i)}\|^2 / \|r^{(i-1)}\|^2 \)

8: \( t^{(i)} = r^{(i)} + β^{(i)} t^{(i-1)} \) /*end for*/

9: \( \hat{ψ} = ψ^{(I)} \)


22

enough approximation of the exact solution with lower computational complexity compared to direct methods by explicit matrix inversion.

### 3.2.2 Efficient CG-based Detection

By using the CG linear solver, we can calculate the equalized signal \( \hat{x}_k \) with lower complexity for the equalization stage of the MMSE detector. Specifically, in the CGSolver according to Algorithm 1, we prepare the regularized Gram matrix \( A_k \) and matched filter output \( y_{k}^{MF} \) as the input \( Λ \) and \( b \), respectively, to approximate the solution \( \hat{x}_k \) as \( \hat{ψ} \) in iterations. To get \( A_k^{-1} \), [42] presents a method to calculate the approximation of \( A_k^{-1} \) during CG iterations, however, it will introduce extra complexity in each CG iteration. In the following, we discuss the efficient conjugate gradient (ECG)-based detection, which removes the calculation of \( A_k^{-1} \) from CG iterations for lower complexity, and approximates the LLR without the need of \( A_k^{-1} \).

As a baseline for comparison, with Cholesky-based equalization, we can calculate exact LLRs based on an explicit \( A_k^{-1} \) according to Eq. (2.7-2.9).

In the ECG-based detector, we directly approximate the equalized signal \( \hat{x}_k \) according to Algorithm 1 without forming \( A_k^{-1} \) during CG iterations. For the LLR calculation as the soft-output of the detector, we need to calculate the effective channel gain \( λ_{u,k} \) and SINR \( ρ_{u,k} \), which are all based on \( A_k^{-1} \) according to Eq. (2.7-2.9).
Here, we show a simple but effective way to approximate LLRs without the need of $A_k^{-1}$ to avoid such extra complexity for the ECG-based detector.

Considering the diagonal entries of $A_k^{-1}$ and $G_k$, $\forall k$, are both real numbers, $\lambda_{u,k} = \text{diag}(A_k^{-1}G_k)_u$, $\forall u,k$, is also a real number. According to Eq. (2.7) and (2.9), we have:

$$\rho_{u,k} = \frac{\lambda_{u,k}}{E_s - E_s \lambda_{u,k}}.$$  \hfill (3.2)

Alternatively, when $B/U$ is large, all of $A_k$, $A_k^{-1}$ and $G$ are diagonally dominant, so we can replace them with the matrices each of which only contains their corresponding diagonal entries. In this way, we can approximate the effective channel gain $\lambda_{u,k}$ as $\lambda_{u,k} = \text{diag}(A_k^{-1}G_k)_u \approx \text{diag}(A_k^{-1})_u \text{diag}(G_k)_u$. Similarly, we have $v_{u,k}^2 = \text{Var}(\Sigma_{j\neq u}(A_k^{-1}G_k)_{ju}x_j + \Sigma_{j=1}^U(A_k^{-1}H_k^H)_{ju}n_j)$ which can be approximated as $v_{u,k}^2 \approx N_0(\text{diag}(A_k^{-1})_u \text{diag}(G_k)_u \text{diag}(A_k^{-1})_u)$. Finally we can approximate $\rho_{u,k} = \lambda_{u,k}^2/\nu_{u,k}^2$ by the $u^{th}$ diagonal entry of $G_k$:

$$\rho_{u,k} = \frac{\text{diag}(G_k)_u}{N_0}.$$  \hfill (3.3)

In this way, we can first find the approximated $\rho_{u,k}$ according to Eq. (3.3), which only relies on the $G_k$ matrix and $N_0$, and then we calculate the approximated $\lambda_{u,k}$ as

$$\lambda_{u,k} = \frac{E_s \rho_{u,k}}{1 + E_s \rho_{u,k}}.$$  \hfill (3.4)

according to Eq. (3.2). Finally we can arrive at an approximated LLR according to Eq. (2.8) with little error performance loss based on simulation results discussed next.
3.2.3 Efficient CG-based Precoding

The ECG-based precoder is more straightforward since we do not need to calculate the soft information. To solve the linear equation $A'_{k}n_{k} = j_{k}$ and get the solution $n_{k}$ for the precoder, where $n_{k} = (H_{k}^{H})^{-1}m_{k}$, we can prepare the regularized Gram matrix $A'_{k}$ and transmit symbol vector $j_{k}$ as the input $\Lambda$ and $b$ in the CGSolver, respectively, to approximate the solution $n_{k}$ as $\hat{\psi}$ in iterations. Then we can obtain the precoded vector $m_{k}$ as $m_{k} = H_{k}^{H}n_{k}$. Finally, we calculate the normalized precoded vector $x'_{k}$ according to Eq. (2.3).

3.3 Block Error Rate Performance

3.3.1 Simulation Setup

To verify the effect on error-rate performance of massive MIMO systems by above algorithm complexity reductions, we perform a batch of simulations for both uplink and downlink under various basestation and user antenna configurations. The low complexity efficient conjugate gradient-based detector (ECG-D) and efficient conjugate gradient-based precoder (ECG-P) are incorporated in an uplink and downlink MIMO-OFDM system, respectively, and the block error rate (BLER) performance results are recorded at different CG iterations. As a baseline for comparison, we also simulate the exact Cholesky-based detector (Chol-D) and precoder (Chol-P) in the uplink and downlink MIMO-OFDM system with the same configurations. In the MIMO-OFDM system, each OFDM symbol has 128 subcarriers. The information bits are encoded with a 5/6-rate convolutional code, modulated to 16-QAM. The channel state information (CSI) is modeled by the WINNER-Phase-2 model [43] or extracted from channel measurement by Argos in real environments. In the WINNER-Phase-2 model, the BS is equipped with a linear antenna array at an antenna spacing of
10m/128 \approx 0.0781m, while the Argos massive MIMO basestation is equipped with antennas arranged in a 2-D array.

### 3.3.2 BLER Under WINNER-II Modelled CSI

We show the simulation results under WINNER-II Modelled CSI in this section with different base station antenna numbers B (128/256/512) and user antenna numbers U (16/32/64). Fig. 3.1 (a) shows the BLER performance of ECG-D at different iterations and Chol-D with U=16 and different BS antenna numbers: B=128, B=256 and B=512. Fig. 3.1 (b) shows the BLER performance of ECG-P and Chol-P when U=16 and B=128,256,512. Similarly, we show the BLER performance when U=32 in Fig. 3.2(a)(b) and the BLER performance when U=64 in Fig. 3.3 (a)(b).
Figure 3.1: BLER Performance at U=16
Figure 3.2: BLER Performance at $U=32$
From the results, we can find that with the increase of CG iterations, the BLER performance of ECG-D and ECG-P can approximate more closely to the exact Chol-D and Chol-P.
3.3.3 BLER Under Real CSI Measured by Argos

We extract the channel matrices from the CSI measured by Argos in a real over-the-air environment. Fig 3.4 (a) shows the BLER performance of ECG-D and Chol-D with the experimental setup of $B=68$, $U=4$ in an uplink system, while Fig 3.4 (b) shows the BLER performance of ECG-P and Chol-P in a downlink system with the same antenna configuration. Fig 3.5 (a)(b) show the BLER performance for the uplink system and downlink system under the antenna setup of $B=84$ and $U=3$. 
Figure 3.4: BLER Performance of 68($B$) × 4($U$) Experimental Setup by Argos
From the results, we can find that the ECG-D and ECG-P can also converge well to the exact Chol-D and Chol-P in the real over-the-air channel environments.
3.4 Complexity Analysis

In this section, we analyze the complexity of the ECG-based detector and precoder and how to tradeoff between accuracy and complexity by performing a certain number of CG iterations.

In Fig. 3.6(a)(b)(c), we show the complexity comparison between the ECG based detector and the Cholesky based detector, for $128 \times 16$, $128 \times 32$, $128 \times 64$ antenna systems, respectively. We omit the complexity of the precoder since the results are quite similar to the detector.

![Graphs showing complexity comparison between ECG-D and Chol-D for different antenna systems.](image)

(a) $128 \times 16$ system  
(b) $128 \times 32$ system  
(c) $128 \times 64$ system

Figure 3.6: Complexity Comparison Between ECG-D and Chol-D

From the above results, we find that with more users, ECG based detector can
Table 3.1: Minimum CG iterations for <1dB SNR loss at $10^{-2}$ BLER

<table>
<thead>
<tr>
<th>B = 128</th>
<th>U=8</th>
<th>U=16</th>
<th>U=32</th>
<th>U=64</th>
</tr>
</thead>
<tbody>
<tr>
<td>B = 256</td>
<td>I=3</td>
<td>I=3</td>
<td>I=5</td>
<td>I=8</td>
</tr>
<tr>
<td>B = 512</td>
<td>I=3</td>
<td>I=3</td>
<td>I=4</td>
<td>I=6</td>
</tr>
</tbody>
</table>

achieve higher complexity reduction at low iterations. The reason is that with more users, the matrix inversion by Cholesky based method will entail larger complexity scaled as $O(U^3)$, while the ECG based linear solver can approximate the solution with only $O(U^2)$ complexity at low iterations. In Table 3.1, we record the minimum CG iteration number $I$ which is required to achieve less than 1dB SNR loss at $10^{-2}$ BLER compared to the exact Cholesky based method for sufficient accuracy, for a variety of antenna configurations (B=128/256/512, U=8/16/32/64). By performing the recorded certain number $I$ of CG iterations for different antenna configurations, we plot Fig. 3.7 to show the complexity reduction rate of ECG-D compared to Chol-D. For example, for 128 $\times$ 64 antenna system, ECG-D can achieve around 40% complexity reduction compared to Chol-D.
3.5 BLER Performance Comparison

In this part, we compare the BLER performance of the exact and approximate MMSE precoding with a low complexity matched filter (MF) beamforming scheme. Figure 3.8(a)(b)(c) illustrate the precoding BLER performance comparison under antenna configurations of $128 \times 8$, $256 \times 8$ and $512 \times 8$, respectively. The results show that the MMSE precoding (both exact and approximate) can outperform the MF precoding in terms of BLER. We also find that the BLER performance of all methods can be improved under larger $B$ and $U$ ratio as expected.
Figure 3.8: BLER Comparison Between MMSE-P (exact and approx.) and MF-P
3.6 Arithmetic Flow of the Unified Design

In a practical massive MIMO system, the basestation should be able to serve as both transmitter and receiver for downlink and uplink transmission, respectively. In this part, we combine the arithmetic flow of both uplink detection and downlink precoding for a unified design, which can support various antenna configurations and also switch between Cholesky-based high accuracy mode and ECG-based low complexity mode to tradeoff between accuracy and complexity. In Chapter 4, we will detail the GPU implementation of such a unified and reconfigurable design based on the arithmetic flow shown in Algorithm 2.
Algorithm 2 Unified Massive MIMO Detector and Precoder

1: \textbf{Input}: \\
2: \quad \mathbf{H}_k \text{ and } y_k \quad (\text{detector}) \quad /* k \text{ is subcarrier index}*/ \\
3: \quad \mathbf{H}'_k \text{ and } j_k \quad (\text{precoder}) \\
4: \textbf{Preprocessing}: \\
5: \quad \text{Calculate } \mathbf{G}_k, \mathbf{A}_k, y_k^{MF} \quad (\text{detector}) \\
6: \quad \text{Calculate } \mathbf{G}'_k, \mathbf{A}'_k \quad (\text{precoder}) \\
7: \textbf{Linear Solver}: \\
8: \quad \text{if Chol.-based mode then} \\
9: \quad \quad \text{Chol. decomp. \& FW./BW. substitution for } \mathbf{A}_k^{-1} \quad (\text{det.}) \text{ or } \mathbf{A}'_k^{-1} \quad (\text{prec.}) \\
10: \quad \text{Calculate } \hat{x}^*_k = \mathbf{A}_k^{-1}y_k^{MF} \quad (\text{detector}) \\
11: \quad \text{Calculate } \hat{n}^*_k = \mathbf{A}'_k^{-1}j_k \quad (\text{precoder}) \\
12: \quad \text{else} /* ECG-based mode */ \\
13: \quad \quad \text{Init: } x^{(0)}_k = 0, r^{(0)}_k = y_k^{MF} \quad (\text{detector}) \text{ or } r^{(0)}_k = j_k \quad (\text{precoder}), \ t^{(0)}_k = r^{(0)}_k \\
14: \quad \quad \text{for } i = 1, 2, \cdots, I \text{ do } /* I: CG iter. for a close approx. */ \\
15: \quad \quad \quad s^{(i-1)}_k = \mathbf{A}_k'^{(i-1)} \quad (\text{detector}) \text{ or } s^{(i-1)}_k = \mathbf{A}'_k'^{(i-1)} \quad (\text{precoder}) \\
16: \quad \quad \quad \alpha^{(i)}_k = \|r^{(i-1)}_k\|^2 / (\langle t^{(i-1)}_k \rangle^H s^{(i-1)}_k) \\
17: \quad \quad \quad x^{(i)}_k = x^{(i-1)}_k + \alpha^{(i)}_k t^{(i-1)}_k \\
18: \quad \quad \quad r^{(i)}_k = r^{(i-1)}_k - \alpha^{(i)}_k s^{(i-1)}_k \\
19: \quad \quad \quad \beta^{(i)}_k = \|r^{(i)}_k\|^2 / \|r^{(i-1)}_k\|^2 \\
20: \quad \quad \quad t^{(i)}_k = r^{(i)}_k + \beta^{(i)}_k t^{(i-1)}_k /* \text{end for} */ \\
21: \quad \hat{x}^*_k = x^{(I)}_k \quad (\text{detector}) \\
22: \quad \hat{n}^*_k = x^{(I)}_k \quad (\text{precoder}) /* \text{end if-else for linear solver} */ \\
23: \textbf{Postprocessing for Detector}: \\
24: \quad \textbf{if Chol.-based detector then} \\
25: \quad \quad \text{Get } \lambda_{u,k} = \text{diag}(\mathbf{A}_k^{-1}\mathbf{G}_k)_u \text{ and } \rho_{u,k}, \forall u \\
26: \quad \textbf{else} /* ECG-based detector */ \\
27: \quad \quad \text{Calculate } \rho_{u,k} \text{ then } \lambda_{u,k}, \forall u /* \text{end if-else} */ \\
28: \quad \text{Calculate the LLR values} \\
29: \textbf{Output}: \\
30: \quad L^p_{u,k}, \forall p, u \quad (\text{detector}) \\
31: \quad m_k = \mathbf{H}'_k n_k, x'_k = E_k m_k / \|m_k\| \quad (\text{precoder})
Design Implementation and Optimization on GPU

4.1 Reconfigurable Unified Architecture

We implement our unified detector and precoder on a GPU according to Algorithm 2 with compute unified device architecture (CUDA)[44]. Fig. 4.1 shows the reconfigurable architecture of our unified MMSE detector and precoder, which can support both uplink detection and downlink precoding and can switch between two modes: a high accuracy Cholesky-based mode and a low complexity ECG-based mode.

For uplink detector, both modes have three major computation stages: (i) a preprocessing stage to compute the matched filter \( y_k^{MF} = H_k^H y_k \), the Gram matrix \( G_k \), and the matrix \( A_k \); (ii) an equalization stage to get the exact or approximated estimation of transmit signal \( \hat{x}_k \) using the Cholesky decomposition or ECG; (iii) a soft-output calculation stage to compute LLR values. For downlink precoder, both modes also have three computation stages: (i) a preprocessing stage to calculate \( A'_k \); (ii) an precoding stage to obtain the exact or approximated solution \( n_k \) based on Cholesky or ECG method; (iii) a postprocessing stage to get the precoded vector \( m_k \) and normalize to \( x'_k \).

In what follows, \( k \in \{1, 2, \ldots, N_{scr}\} \), where \( N_{scr} \) indicates the number of subcar-
riers in an OFDM symbol. We define $N_{sym}$ as the number of $N_{scr}$-subcarrier OFDM symbols. For the detector input, we prepare $N_{sym}$ sets of received frequency domain signals $\{y_1, y_2, \ldots, y_{N_{scr}}\}$ as the payload of a certain streaming frame at the receiver (RX), each $y_k$ including $B$ samples corresponding to $B$ antennas of the BS, while for the precoder input, we have $N_{sym}$ sets of transmit vectors $\{j_1, j_2, \ldots, j_{N_{scr}}\}$ instead, each $j_k$ including $U$ samples for $U$ user. We also prepare $N_{sym}$ sets of channel matrices $\{H_1, H_2, \ldots, H_{N_{scr}}\}$ for a frame (uplink and downlink channels are assumed reciprocal), as well as the control signals for mode switching. At the output, the detector generates LLR values for every user at a time for a certain streaming frame, while the precoder generates the normalized precoded vectors $x'_k$ for beamforming to desired terminals.

To enable real-time processing, our design is able to dispatch streaming frames onto multiple streams generated in multiple GPUs for task pipelining, and process streaming payloads continuously and efficiently. The following sections discuss the kernel design and task pipelining.
4.2 Kernel Design and Optimization

4.2.1 Detector Kernel Design

We detail the kernel implementation for the detector of the unified design in this section. Considering the computation similarity between the detector and precoder, many of the detector kernels in the unified design can be reused for precoding calculations with only the change of inputs.

4.2.1.1 Preprocessing Stage

We calculate the matrices $G_k$ and $A_k$, and the vector $y_k^{MF}$ in a per-subcarrier basis. Since calculating $G_k$, $A_k$, and $y_k^{MF}$ exhibits no data dependency for each subcarrier, we fetch $N_{CR} = N_{sym} \times N_{scr}$ channel matrices $H_k$ and vectors $y_k$ from the GPU’s global memory, and calculate a batch ($batchSize = N_{CR}$) of $G_k$, $A_k$ and $y_k^{MF}$ in parallel for all $N_{CR}$ subcarriers. With large $N_{CR}$, we obtain high utilization of the GPU’s computing resources. The computations of $G_k$ and $y_k^{MF}$ are similar, i.e., matrix-matrix (vector) multiplication, which can be efficiently implemented by the cuBLAS [44] library, a GPU accelerated Basic Linear Algebra Subprograms (BLAS) library. Specifically, we choose the `cublasCgemmBatched()` function, which is fine tuned for computing a batch of small complex matrix multiplications in the format of $C_i = A_iB_i$, where $i$ is the matrix index, and $A, B, C$ are complex matrices or complex vectors. To avoid unnecessary matrix transpose data movement, and to correctly utilize the cuBLAS function, we directly store the $H_k$ and $y_k$ in column-major format in GPU global memory as the input of the function. After a batch of matrices $G_k$ have been computed, we can easily form the matrix $A_k$ by a light kernel function, which adds a constant $N_0/E_s$ to each diagonal entry of $G_k$ matrix in parallel for all $N_{CR}$ subcarriers.
4.2.1.2 Equalization Stage

In this stage, our detector supports a Cholesky-based inversion, which explicitly calculates $A_k^{-1}$, as well as a ECG-based equalizer which approximates the MMSE estimate $\hat{x}_k$ without ever forming $A_k^{-1}$. For simplicity and efficiency, we implement the Cholesky-based equalizer by some special batch-supported cuBLAS functions. In particular, we choose `cublasCgetrfBatched()` with pivoting disabled to perform the Cholesky decomposition of $A_k$, and `cublasCgetriBatched()` to perform the forward and backward substitutions to get $A_k^{-1}$. Then we use `cublasCgemmBatched()` again to calculate estimated $\hat{x}_k$ by $\hat{x}_k = A_k^{-1}y_{MF}$. Note that all these functions are still performed on batchSize $= N_{CR}$ subcarriers in parallel, and the intermediate results can be shared in pre-allocated GPU global memories.

For the ECG-based equalizer, most of the computations are vector additions or multiplications, except for the calculation of $s^{(i)}_k$ at the beginning of every $i$th iteration. For $s^{(i)}_k$, we can calculate batchSize $= N_{CR}$ number of $s^{(i)}_k = A_k t^{(i)}_k$ first by `cublasCgemmBatched()` in parallel. For the following computations as denoted in lines 10–14 of Algorithm 1, although we can still perform those vector operations by cuBLAS vector-related functions, we resort to our customized kernel design CGsolver, where we can take full advantage of the GPU memory hierarchy within a kernel for potentially better performance, instead of sharing intermediate results between multiple kernels via slower global memory as operated by cuBLAS functions.

The functionality of our kernel CGsolver is to update the value of $t^{(i)}_k, r^{(i)}_k$ and the approximate $x^{(i)}_k$ in each iteration, so we need to store those variables in global memory for data sharing between adjacent CG iterations. Note that each of those variables is actually a vector including $U$ elements. For instance, $t^{(i)}_k$ is a vector constructed as $[t^{(i)}_{1,k}, t^{(i)}_{2,k}, \ldots, t^{(i)}_{U,k}]^T$, where the element $t^{(i)}_{u,k}$ corresponds to the $u$th user, $k$th subcarrier and $i$th CG iteration. Given a certain streaming frame to process in a certain CG
iteration, we launch the kernel with $N_{CR} \times U$ threads, each controlling the processing for each element in parallel. For efficient computations within the kernel, we first fetch each element $t_{u,k}^{(i)}$, $r_{u,k}^{(i)}$, $x_{u,k}^{(i)}$ and also the previously calculated $s_{u,k}^{(i)}$ from global memory and store them to local registers in a naturally coalesced way by each thread. While the vector addition computations such as $x_{k}^{(i)} = x_{k}^{(i-1)} + \alpha_{k}^{(i)} t_{k}^{(i-1)}$ can be performed under a per-element basis in parallel, the calculation of $\alpha_{k}^{(i)}$ and $\beta_{k}^{(i)}$ is based on vector dot product results such as $(t_{k}^{(i)})^H s_{k}^{(i)}$ or squares of vector magnitude such as $\|r_{k}^{(i)}\|^2$, which requires data sharing between multiple parallel threads. For example, to calculate $(t_{k}^{(i)})^H s_{k}^{(i)}$, we have to realize the thread communication within every group of $U$ threads, where the $u^{th}$ thread can see its own local register elements $t_{u,k}^{(i)}$ and $s_{u,k}^{(i)}$ but not others’ register elements. Here, we use shared memory, a manually controlled L1 cache-like on-chip memory for efficient data sharing and thread communication within a thread block.
As stated before, we create a total of \(N_{CR} \times U\) threads when launching the kernel, specifically, we have \(N_{CR}/N_{\text{shared}}\) thread blocks with \(N_{\text{shared}} \times U\) threads in each block. Here, \(N_{\text{shared}}\) denotes how many sets of shared units we deployed within each thread block, where each set of shared units handles the communication within a group of \(U\) threads. To calculate a vector dot product or a square of vector magnitude, we need to add a group of scalar products together, reducing them to a sum result. Such operations can be efficiently realized by \textit{atomic} addition, which adds the results from different threads to a shared location, such as shared memory or global memory, in serial with implicit synchronization locks between threads. On the latest Nvidia GPU with \textit{Maxwell} architecture, atomic operations have been further enhanced for shared memory\cite{44}. We take advantage of this new feature and perform \textit{atomicAdd} operations on the elements within every group of \(U\) threads for the reduction to a set of shared units which can be seen and used by those \(U\) threads. A total of \(N_{\text{shared}}\) sets of such shared units are deployed for handling \(N_{\text{shared}} \times U\) threads in each thread block. We finally have \(N_{\text{shared}}\) sets of \(\alpha_k^{(i)}\) and \(\beta_k^{(i)}\) results stored in the shared memory of each block, so that they can be efficiently fetched as shared coefficients during the vector addition computations such as lines 11,12,14 in Algorithm 1. Fig. 4.2 shows an example of the reduction process by serialized atomic operations on certain shared memories within a thread block when calculating \((t_k^{(i)})^H s_k^{(i)}\). The CG iteration index \(i\) is omitted for convenience in the figure.

\subsection{4.2.1.3 Soft-Output Computation Stage}

We calculate the LLR values in this stage. The kernel is launched with a total of \(N_{CR} \times U\) threads to process each I/Q sample of each subcarrier for each user in parallel. In the kernel, we can either calculate \(\lambda_{u,k}\) then \(\rho_{u,k}\) for LLRs of the Cholesky-based detector, or calculate \(\rho_{u,k}\) then \(\lambda_{u,k}\) without explicitly forming \(A_k^{-1}\) for the ECG-based
detector, depending on the detector mode. The minimum operations are performed on multiple bits corresponding to a certain modulated I/Q sample. Since our kernel is launched under a per-sample basis, each thread will calculate all the LLR values of those bits corresponding to a certain I/Q sample, with those min values reduced to a local register variable of that thread instead of any shared memory.

4.2.2 Precoder Kernel Design

Considering the similarity between the precoding computations and detection computations, many of the detector kernels can be reused to perform precoding functionality. For example, the kernels in preprocessing stage for calculating \( G'_k \) and \( A'_k \) for the precoder are actually the same with the detector, except that we only need to change the inputs to downlink channel matrices \( H'_k \) instead of uplink channel \( H_k \). For the precoding stage, the Cholesky linear solver kernels or ECG-based linear solver kernels can also be reused to calculate the exact or approximate linear solution \( n_k \) given the solver inputs \( A'_k \) and \( j_k \). For the postprocessing stages, the calculation of precoded vector \( m_k = H'_k n_k \) is realized by \( N_{CR} \)-batched \( \text{cublasCgemmBatched}() \) kernels, and the following normalization \( x'_k = \frac{E_x m_k}{\|m_k\|} \) is performed with a \( (N_{CR} \times B) \)-threaded customized kernel, where the magnitude \( \|m_k\| \) is calculated by atomic reduction in the same way as the vector dot products in the \( CGsolver \) kernel.

4.2.3 Summary of Kernel Optimization Strategies

In our design, some key strategies to optimize GPU kernel computation performance are: (i) keep high occupancy of GPU computing cores; (ii) exploit high bandwidth on-chip memory resources such as local registers and shared memories; (iii) access well aligned data in global memory in a coalesced way to reduce memory transaction overhead.
To keep high occupancy of GPU cores, we need to ensure that the workload is computationally intensive and has inherent parallelism. In the linear MMSE detection and precoding for massive MIMO system, an obvious coarse-grained parallelism is that the detection/precoding can be performed on a per-subcarrier basis in an OFDM system, that is, we can process $N_{CR}$ subcarriers in a frame in parallel. Furthermore, for the detection/precoding of each subcarrier, most of the computations are matrix multiplications, matrix inversions, and vector operations, which have inherent fine-grained parallelism. We take advantage of the hierarchical parallel programming model (grid, block, thread) to represent such hierarchical arithmetic parallelism (coarse-grained and fine-grained), efficiently mapping the processing workload onto the hierarchical parallel architecture of GPU (many parallel streaming multiprocessors, each including hundreds of parallel computing cores). To feed the streaming multiprocessors with enough thread blocks for high throughput, we need a large $N_{CR}$ subcarriers to be processed within a short period, and such computationally intensive workload is not hard to produce in a real-time communication system.

To exploit local registers and shared memory resources, we need to strike a balance between the number of parallel threads and the computation efficiency of each thread. Specifically, when each thread can use more local registers and shared memories for its corresponding computations, the computation efficiency for each thread will be enhanced. However, since the total register number and shared memory size are limited in the streaming multiprocessor (SM), the total number of parallel threads that can be generated in that SM will decrease due to higher on-chip memory utilization by each thread, which may lead to more idle computing cores and lower computing performance. Therefore, we need to carefully deploy the on-chip memory resources for each thread to make sure that each thread can compute efficiently and that enough parallel threads can be generated to keep high occupancy of computing cores.
To reduce the overhead of memory transactions with slower global memory, we need to align and store the data in the global memory with contiguous addresses so that a batch of continuous data can be fetched or written in a coalesced way within a single memory transaction. Fortunately, in our design, most of the input, intermediate and output data for the detector and precoder are matrices or vectors, which can be stored in a naturally coalesced way.

4.3 Multi-stream Concurrent Task Scheduling

We have discussed how to design and optimize various computation kernels for efficient execution, but usually the CPU-GPU memory copy of input or output data will also introduce significant overhead, especially when we have a large data set to transfer between CPU and GPU. Here, we perform multi-stream scheduling in our design for the task pipelining of CPU-GPU memory copy and kernel execution, so that the processing tasks of streaming frames can be dispatched onto multiple streams and handled concurrently.

Here, we only show the multi-stream scheduling strategy for the detection workload as an example considering we can perform such scheduling in the same way
for the precoding workload. Consider that we have $N_{\text{frame}}$ number of streaming frames received continuously in a real-time uplink receiver, for a certain frame, the payload data has $N_{\text{sym}}$ frequency domain OFDM symbols, each including $N_{\text{scr}}$ sub-carrier signals $\{y_1, y_2, \ldots, y_{N_{\text{scr}}}\}$. As the input of our detector, those payload data as well as pre-calculated channel matrices will be initially stored in page-locked host memory for faster CPU to GPU asynchronous memory copy. Note here to avoid redundant host-device memory copy for a certain frame, while we need to copy all the $N_{\text{sym}}$ OFDM payload symbols from host, we only need to copy $N_{\text{scr}}$ channel matrices $\{H_1, H_2, \ldots, H_{N_{\text{scr}}}\}$ of a certain OFDM symbol from the host. We then broadcast those channel matrices to all OFDM symbols of the frame within GPU global memory by \textit{Device To Device Memcpy} with higher memory bandwidth than PCIe bus, assuming the channel information is static across different OFDM symbols in a frame. When we schedule the processing tasks of streaming $N_{\text{frame}}$ RX frames onto $N_{\text{stream}}$ streams, the task of the $f^{th}$ frame will be dispatched to the $(f \mod N_{\text{stream}})^{th}$ stream, with the Hyper-Q feature enabled in the current Maxwell architecture GPU to avoid false dependencies between multiple streams\cite{44}. After detection, the LLR values for a frame will be asynchronously transferred from GPU global memory to pre-allocated page-locked host memory in corresponding streams. Fig. 4.3 shows the multi-stream scheduling on processing tasks of streaming frames for overlapping the CPU-GPU memory copy latency. An ideal pipeline requires two memory copy engines (H2D and D2H) and the kernel execution engine to keep working without any waiting or idle time slots (pipeline bubbles). While such a condition is hard to match exactly in most cases, we select a proper $N_{\text{stream}}$ experimentally to reduce pipeline bubbles for near-optimal pipeline performance.

For downlink precoding, we also perform such scheduling for overlapping the CPU-GPU memory copy latency with the precoding kernel executions.
4.4 Multi-GPU Workload Deployment

When the frames are generated faster than a single GPU can handle, they may stall at the host memory waiting for deployment. To support the multi-GPU extension in our unified detector and precoder design, as shown in Fig. 4.4, we create $N_{GPU}$ threads in the CPU by OpenMP APIs, each controlling the workload processing in a certain GPU in our multi-GPU system, so that the detection or precoding tasks can be deployed evenly and performed concurrently on multiple GPUs, further improving the detection and precoding throughput.
5.1 Experimental Hardware and Tools

The experimental platform includes an Intel i7-3930K six-core 3.2GHz CPU and two 1GHz Nvidia GTX 980Ti graphics cards. The GTX 980Ti has 6GB GDDR5 device memory and a Maxwell GPU with 2816 CUDA cores. The CPU and graphic cards communicate via PCIe x16 interfaces. We use the CUDA Toolkit 7.0 running on Linux 64-bit OS for the design implementation, debugging and performance profiling. The GPU benchmark results are evaluated on an Nsight release version of the design with -O3 compiler optimization. The single-core CPU and multi-core CPU benchmark results are evaluated on a GCC -O3 compiled executable program.

5.2 Benchmark Results

Table 5.1 and 5.2 show the benchmark results of the detector, and Table 5.3 shows the performance of the precoder.

In Table 5.1, we record the detection throughput performance at different configurations of BS antenna numbers ($B$) and user antenna numbers ($U$), as well as different
detection modes, i.e., Cholesky-based mode or ECG-based mode, when scheduling the
detection workload in a single stream and a single GPU. 16-QAM modulation is used.
$N_{sym}$ indicates the number of OFDM symbols in a frame, and each OFDM symbol
includes $N_{scr}=128$ subcarriers. With the increase of $N_{sym}$ and the total number of
subcarriers $N_{CR} = N_{sym} \times N_{scr}$, the throughput will also increase due to higher uti-
lization rate of GPU computing resources with a larger size of workload being fed
to the GPU. We compare the pure kernel execution throughput and the effective
throughput including the host-device memory copy overhead at different $N_{sym}$. The
results show that the memory copy overhead will introduce around 20%-30% through-
put performance loss for $128(B) \times 16(U)$ MIMO systems, and higher performance loss
for $256(B) \times 32(U)$ MIMO systems. We also show that the ECG-based detector with
a proper iteration number, e.g., minimum iteration for <1dB SNR loss at $10^{-2}$ BLER
compared to Cholesky-based detector, achieves better throughput at different config-
urations of antenna numbers and detection workloads, so that one can trade-off error
performance with throughput by reconfiguring the detection mode of our design. For
our precoder design under a single stream scenario, the throughput performance fol-
low the same rules with the detector performance: with the increase of $N_{sym}$, the
precoding throughput will also increase and the ECG-based precoder can achieve
higher throughput with <1dB SNR loss at $10^{-2}$ BLER compared to Cholesky-based
precoder.

In Table 5.2, we show the enhanced results by performing multi-stream and multi-
GPU deployment. The detection tasks of streaming frames are evenly deployed on
$N_{GPU}$ GPUs, each GPU generating $N_{stream}$ streams with each stream processing $N_{sym}$
OFDM symbols for a certain frame. By setting a typical upper bound on detection
latency (several milliseconds) for a real-time system, for example, 4ms in our experi-
ments, we record a proper combination of $N_{stream}$ and $N_{sym}$ for a near optimal task.
Table 5.1: Single Stream Throughput Performance of Detector

<table>
<thead>
<tr>
<th>$N_{sym}$</th>
<th>$128 \times 16$ (in Mb/s)</th>
<th>$256 \times 32$ (in Mb/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N_{sym}$</td>
<td>Chol</td>
<td>ECG@3iter</td>
</tr>
<tr>
<td>8</td>
<td>$88.92/61.80$</td>
<td>$106.74/71.58$</td>
</tr>
<tr>
<td>16</td>
<td>$106.56/78.39$</td>
<td>$127.87/88.86$</td>
</tr>
<tr>
<td>32</td>
<td>$118.19/90.74$</td>
<td>$144.51/102.65$</td>
</tr>
<tr>
<td>64</td>
<td>$130.45/102.51$</td>
<td>$155.02/113.93$</td>
</tr>
</tbody>
</table>

K: pure Kernel execution throughput
K+M: throughput under (Kernel execution + Memory copy overhead)

Table 5.2: Multi-stream & Multi-GPU Enhanced Performance of Detector

<table>
<thead>
<tr>
<th>$N_{GPU}$</th>
<th>$N_{stream}$</th>
<th>$N_{sym}$</th>
<th>Detector mode</th>
<th>$T$ (Mb/s)</th>
<th>$L$ (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>8</td>
<td>$128 \times 16$ Chol.</td>
<td>131.62</td>
<td>3.98</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$128 \times 16$ ECG@3iter</td>
<td>148.67</td>
<td>3.54</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>16</td>
<td>$128 \times 16$ Chol.</td>
<td>265.94</td>
<td>3.94</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$128 \times 16$ ECG@3iter</td>
<td>286.57</td>
<td>3.66</td>
</tr>
</tbody>
</table>

T: Throughput on processing streaming frames (including memory copy overhead)
L: Detection latency for each frame

Pipelining with few pipeline stalls to achieve high throughput. With multiple streams, we overlap almost all the memory copy overhead for a higher throughput which is close to the pure kernel execution throughput in Table 5.1. By using two GPUs, the total throughput can be enhanced by around two times, and our ECG-based detector achieves 286.57 Mb/s throughput with 3.66 ms latency for supporting a $128(B) \times 16(U)$ MIMO system at 16-QAM. In Table 5.3, we record the corresponding performance results for the precoder with multi-stream scheduling and multi-GPU deployment performed. Similarly, the precoder can also achieve over 250Mbps throughput with less than 4ms latency by running on two GPUs concurrently.

We can further extend our design to run on more GPUs in the future for even higher throughput, for example, by using four Nvidia 980Ti GPUs concurrently, our design is likely to achieve over 0.5 Gb/s throughput for a $128 \times 16$ MIMO system at 16-QAM.
Table 5.3: Multi-stream & Multi-GPU Enhanced Performance of Precoder

<table>
<thead>
<tr>
<th>$N_{GPU}$</th>
<th>$N_{stream}$</th>
<th>$N_{sym}$</th>
<th>Precoder mode</th>
<th>T (Mb/s)</th>
<th>L (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>8</td>
<td>$128 \times 16$ Chol.</td>
<td>125.67</td>
<td>4.17</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$128 \times 16$ CG@3iter</td>
<td>145.40</td>
<td>3.61</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>16</td>
<td>$128 \times 16$ Chol</td>
<td>258.59</td>
<td>4.06</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$128 \times 16$ CG@3iter</td>
<td>284.49</td>
<td>3.69</td>
</tr>
</tbody>
</table>

T: Throughput on processing streaming frames (including memory copy overhead)
L: Precoding latency for each frame

5.3 Latency/Throughput Tradeoff

Latency is another important criteria for evaluating the performance of design. Since the CSI can only keep static within coherence time $\tau$ (usually from several hundreds of microseconds to several milliseconds according to the BS/user mobility), we need to make sure that the processing latency $t$ of our design satisfies $t < \tau$, which is particularly essential for downlink precoding.

In our design, we process $N_{sym}$ OFDM symbols together as a workload frame to be fed into the GPU’s streaming multiprocessors in a processing cycle with latency duration $t$. On one hand, we need larger $N_{sym}$ to feed more workload to the GPU to keep high occupancy of GPU resources for higher data throughput, however, on the other hand, smaller $N_{sym}$ means shorter latency of a processing cycle, so we need to strike a balance between throughput and latency to satisfy certain throughput or latency requirements. In Fig. 5.1(a) and 5.1(b), we show the throughput/latency results for a $128 \times 16$ uplink and downlink system, respectively, in both ECG (@3iter) and Chol modes. The number beside a marker is the latency result (in ms) of the corresponding data point in the plot. The results show that with the increase of workload denoted as $N_{sym}$, the throughput of GPU will increase and converge to a saturated peak performance, but the latency will also increase, so one needs to select a proper $N_{sym}$ according to certain throughput/latency requirements or specific wireless standards.
The number besides a marker is the latency result (in ms) of the corresponding data point in the plots.

Figure 5.1: Throughput/Latency Tradeoff
Table 5.4: Performance Comparison with Previous Work

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Detector Type</th>
<th>ATN Config.</th>
<th>T (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtex 7 FPGA</td>
<td>MMSE (CGLS)</td>
<td>128 × 8</td>
<td>20</td>
</tr>
<tr>
<td>Virtex 7 FPGA</td>
<td>MMSE (Chol.)</td>
<td>128 × 8</td>
<td>603</td>
</tr>
<tr>
<td>Quadro FX1700 GPU</td>
<td>SSFE</td>
<td>2 × 2</td>
<td>36.06</td>
</tr>
<tr>
<td>Tesla C2070 GPU</td>
<td>FPFSD</td>
<td>4 × 4</td>
<td>92.31</td>
</tr>
<tr>
<td>Tesla C1060 GPU</td>
<td>Trellis-based</td>
<td>4 × 4</td>
<td>120.0</td>
</tr>
<tr>
<td>Tesla C2050 GPU</td>
<td>MMSE</td>
<td>4 × 4</td>
<td>100</td>
</tr>
<tr>
<td>This work</td>
<td>GTX 980Ti GPU×2</td>
<td>MMSE (Chol.)</td>
<td>128 × 16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MMSE (CG)</td>
<td>128 × 16</td>
</tr>
</tbody>
</table>

Figure 5.2: Speedup Rate: GPU VS Multicore CPU

5.4 Performance Comparison

In Table 5.4, we show the performance comparison of our GPU-based detector with previous work. Our design has comparable data rate with FPGA based design, and can outperform previous GPU based detectors in terms of data rate performance, even though they are targeting small scale MIMO systems. In Fig. 5.2, we compare the performance of GPU based detector (ECG mode @ 3iter, 128 × 16 antenna system) with a pure C code implementation running on a 6-core Intel i7-3930K CPU. OpenMP APIs are used to deploy the workload evenly on the multicore CPU. The results show
that evident speedup can be achieved by our GPU-based design, especially when increasing the workload denoted by \( N_{sym} \). Similar speedup rate can also be obtained by our GPU-based precoder.

### 5.5 Further Analysis: A System Perspective

In this section, we analyze the effective capacity of our MMSE detector or precoder from a system perspective. In Section 3.5, we have compared the BLER performance of the exact and approximate MMSE precoding with a low complexity matched filter (MF) beamforming scheme. Here, for simplicity but without losing generality, we focus on the discussion of the tradeoff point among our Cholesky based MMSE precoder, ECG based MMSE precoder, and a simpler matched filter (MF) based precoder.

We adopt the system performance model introduced in [50], which is derived based on some key practical factors, including environmental factors, such as channel coherence time and precoder spectrum efficiency, and system design factors, such as number of antennas and hardware capability. According to [50], the effective capacity (with channel estimation overhead considered) of MF precoding is

\[
\Theta_{MF} = \frac{Coh_t - \frac{U}{Coh_b}}{Coh_t} \times \theta_{MF} \times U,
\]

and the effective capacity (with the channel estimation overhead, transport latency, and processing latency considered) of MMSE precoding is

\[
\Theta_{MMSE} = \frac{Coh_t - \frac{U}{Coh_b} - (2 \times \left( \frac{B \times U \times N_{sub} \times N_b}{S} + L \right) + N_{sub} \times T_{prec} \})}{Coh_t} \times \theta_{MMSE} \times U,
\]

where the definitions and corresponding values of variables in our precoder design are listed in Table 5.5. For the MMSE precoder, we simulate both the Cholesky based
Table 5.5: Parameters of Systematical Performance Model

<table>
<thead>
<tr>
<th>Variable</th>
<th>Definition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Coh_t$</td>
<td>Coherence Time</td>
<td>0.1 ms – 100 ms</td>
</tr>
<tr>
<td>$Coh_b$</td>
<td>Coherence Bandwidth</td>
<td>313 kHz</td>
</tr>
<tr>
<td>$\theta$</td>
<td>Spectrum Efficiency/User</td>
<td>6.12 bps/Hz/User (MF)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8.40 bps/Hz/User (Chol-MMSE)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8.32 bps/Hz/User (ECG(@3iter)-MMSE)</td>
</tr>
<tr>
<td>$U$</td>
<td>Number of Users</td>
<td>16</td>
</tr>
<tr>
<td>$B$</td>
<td>Number of BS Antennas</td>
<td>128</td>
</tr>
<tr>
<td>$S$</td>
<td>Data Transport Throughput</td>
<td>40Gbps/100Gbps/500Gbps</td>
</tr>
<tr>
<td>$L$</td>
<td>Data Transport Hop Latency</td>
<td>1 $\mu$s</td>
</tr>
<tr>
<td>$T_{prec}$</td>
<td>MMSE Precoding Latency</td>
<td>242$\mu$s (Chol)/ 192$\mu$s (ECG@3iter)</td>
</tr>
<tr>
<td>$N_b$</td>
<td>Number of Bits per CSI</td>
<td>64 bits</td>
</tr>
<tr>
<td>$N_{sub}$</td>
<td>Number of Subcarriers</td>
<td>128</td>
</tr>
<tr>
<td>$\Theta$</td>
<td>Effective Capacity</td>
<td>(Result) bps/Hz</td>
</tr>
</tbody>
</table>

exact MMSE precoder and the ECG based approximate MMSE precoder. The $T_{prec}$
values are measured based on our GPU implementation of MMSE precoders.

Figure 5.3(a)(b)(c) show the simulation results on the effective capacity performance of MF and MMSE precoders with the variation of coherence time, for three different configurations of transport efficiency: 40Gbps, 100Gbps and 500Gbps throughput, respectively.
From the results, we can find that the MF precoder can outperform MMSE pre-
coder in terms of effective capacity when the coherence times are less than 4ms, 2ms and 1ms for 40Gbps, 100Gbps and 500Gbps throughput transport, respectively. With the increase of the transport throughput, the total transport and processing overhead for MMSE will be reduced, so the tradeoff point for MF and MMSE will shift left.

Also, the ECG based MMSE precoder can achieve higher effective capacity than Chol. based MMSE precoder with lower transport overhead before getting to their corresponding tradeoff point, which is around 5ms for all those three transport configurations. The reason is that with the reduction of transport overhead, the processing latency will play a more important role for the final effective capacity of MMSE precoder, so the ECG based precoder which has higher precoding processing efficiency can performs even better than the Cholesky based precoder with the reduction of transport overhead.
In this thesis, we present the design and implementation of a GPU accelerated reconfigurable detector and precoder targeting massive MIMO software-defined radio systems. We start from selecting proper detection and precoding algorithms, and finally decide to focus on linear MMSE detector and precoder considering the good error rate performance and manageable complexity in the massive MIMO scenarios. To further reduce the algorithm complexity, we judiciously perform numerical approximation to achieve low complexity without sacrificing much of the accuracy. Specifically, we propose the efficient conjugate gradient (ECG) based method to avoid the explicit matrix inversion, which is the dominant computation in the MMSE detector and precoder. The simulation results show that the BLER performance of the ECG-based detector and precoder converge well to the exact Cholesky-based method with a small number of iterations and evident complexity reduction. We then map our design on GPU for efficient and flexible implementation. By exploring the inherent parallelism of the algorithm and taking advantage of the numerous computing cores, parallel architecture and hierarchical memory resources of the GPU, we design and optimize the kernel functions for parallel processing of the computationally intensive workloads required by the detector and precoder. Those computation tasks of ker-
nel functions are furthermore pipelined on multiple GPU streams and deployed on multiple GPUs for reducing the host-device memory transfer overhead. Our unified and reconfigurable design can switch between detection and precoding functionalities, can tradeoff the accuracy and complexity by supporting a Cholesky based mode and a ECG mode, and can support various antenna configurations and modulation orders. Such reconfigurability of our design provides high flexibility on prototyping and testing different setups and algorithms for various proposals for the 5G wireless standards and for software-defined radios. Our design achieves over 250Mbps throughput with less than 4ms latency for supporting a $128 \times 16$ antenna system by running on two latest generation of GPUs concurrently. Higher performance can be potentially obtained by running on more GPUs.

In the future, further investigation can be done in both algorithm and implementation aspects to make the design more practical. Lower complexity algorithms are still required. Some other iterative methods besides conjugate gradient may converge faster by using specific preconditions with lower complexity. Distributed optimization algorithms, which can be used to distribute the centralized zero-forcing or MMSE processing in a cluster-based manner, may handle the bottleneck of limited bandwidth which arises when we need to aggregate data from hundreds of antennas to a centralized processor, so that the design can have better scalability. Furthermore, the integration of the soft-output detector design with soft-input decoder design for iterative detection and decoding in a massive MIMO receiver can potentially further improve the error rate performance of the whole system.
References


uplink,” in *Circuits and Systems (ISCAS), 2013 IEEE International Symposium on*, May 2013, pp. 2155–2158. 1.1


[22] “IEEE Standard for Information technologyTelecommunications and information exchange between systemsLocal and metropolitan area networksSpecific requirements-Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 3: Enhancements for Very High Throughput in the 60 GHz Band.” 2012. 2.2.1


