# Accelerating Massive MIMO Uplink Detection on GPU for SDR Systems

Kaipeng Li*, Bei Yin*, Michael Wu*, Joseph R. Cavallaro*, and Christoph Studer†
*Dept. of Electrical and Computer Engineering, Rice University, Houston, TX, USA
†School of Electrical and Computer Engineering, Cornell University, Ithaca, NY, USA

***Abstract*—We present a reconfigurable GPU-based uplink detector for massive MIMO software-defined radio (SDR) systems. To enable high throughput, we implement a configurable linear minimum mean square error (MMSE) soft-output detector and reduce the complexity without sacrificing its error-rate performance. To take full advantage of the GPU computing resources, we exploit the algorithm's inherent parallelism and make use of efficient CUDA libraries and the GPU's hierarchical memory resources. We furthermore use multi-stream scheduling and multi-GPU workload deployment strategies to pipeline streaming-detection tasks with little host-device memory copy overhead. Our flexible design is able to switch between a high accuracy Cholesky-based detection mode and a high throughput conjugate gradient (CG)-based detection mode, and supports various antenna configurations. Our GPU implementation exceeds 250 Mb/s detection throughput for a 128×16 antenna system.**

## I. Introduction

Massive multiple-input multiple-output (MIMO) is believed to be a key technology for future 5G wireless systems. 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 [1]. In particular, data detection, which demultiplexes the spatial data streams at the BS, is among the most computationally intensive tasks. Due to the prohibitive complexity of optimal data detection methods, sub-optimal low complexity detectors, such as the linear minimum mean square error (MMSE) data detector, must be used in practice to achieve sufficiently high throughput at reasonable hardware costs.

To enable high throughput data detection for massive MIMO, recent FPGA and ASIC results in [2], [3] 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 some most complex baseband algorithms, such as LDPC decoding [4] or turbo decoding [5]. However, not much is known about the efficacy of GPGPU for data detection in massive MIMO SDR systems.

In this paper, we present—to the best of our knowledge—the first GPU implementation of a data detector for massive MIMO SDR systems. Our detector design supports various BS and user antenna configurations, and offers performance/complexity trade-offs by supporting two algorithm modes: an exact Cholesky-based mode and an approximate conjugate gradient (CG)-based mode. To improve the processing throughput and reduce the latency of our design, we judicially perform various GPGPU optimization strategies on kernel design, memory usage, and task scheduling. By running our design on two GPU devices concurrently, we achieve over 250 Mb/s throughput for a 128×16 antenna massive MIMO SDR system.

## II. Massive MIMO detection algorithms

### A. OFDM Uplink System Model

We consider a $B$ antenna BS, which serves $U \leq B$ single antenna users. In the OFDM uplink, 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 the time domain and transmitted over the wireless channel. At the receive-side, the BS computes frequency domain signals and then, performs data detection and decoding. We define $y_{b,k}$ as the received frequency-domain sample of the $b^{th}$ antenna and the $k^{th}$ subcarrier, and $x_{u,k} \in \Omega$ as the signal from the $u^{th}$ user and $k^{th}$ subcarrier. The input-output relation of the channel is modeled as $y_k = \mathbf{H}_k x_k + n_k$, where $y_k = [y_{1,k}, \ldots, y_{B,k}]^T$ and $x_k = [x_{1,k}, \ldots, x_{U,k}]^T$. The channel matrix is $\mathbf{H}_k \in \mathbb{C}^{B\times U}$, and $n_k = [n_{1,k}, \ldots, n_{B,k}]^T$ is the noise vector, where each entry $n_{b,k}$ is assumed to be i.i.d. zero-mean complex Gaussian with variance $N_0$. We set the uplink transmit power for each user to $E_s$ and define the average receive SNR as $UE_s/N_0$.

### B. Linear Soft-Output MMSE Detection

To achieve low complexity, we focus on the linear MMSE soft-output detection algorithm described in [6]. In order to compute an estimate of the transmit signal $x_k$, the MMSE detector first computes $\hat{x_k} = \mathbf{W}_k y_k$ where $\mathbf{W}_k = (\mathbf{H}_k^H \mathbf{H}_k + \frac{N_0}{E_s}\mathbf{I}_U)^{-1}\mathbf{H}_k^H$ is the MMSE equalization matrix [7]. Here, $\mathbf{H}_k^H$ denotes the adjoint of the channel matrix $\mathbf{H}_k$, and $\mathbf{I}_U$ is the $U \times U$ identity matrix. In order to avoid redundant computations, we first compute the regularized Gram matrix $\mathbf{A}_k = \mathbf{G}_k + \frac{N_0}{E_s}\mathbf{I}_U$ with $\mathbf{G}_k = \mathbf{H}_k^H \mathbf{H}_k$ and the matched-filter output $y_k^{MF} = \mathbf{H}_k^H y_k$. With this, we can compute the MMSE estimate: $\hat{x}_k = \mathbf{A}_k^{-1} y_k^{MF}$. To compute soft-output information in the form of log-likelihood ratio (LLR) values, we compute [6]

$$L_{u,k}^p = \rho_{u,k}\left(\min_{a\in\Omega_p^0}\left|\frac{\hat{x}_{u,k}}{\lambda_{u,k}} - a\right|^2 - \min_{a'\in\Omega_p^1}\left|\frac{\hat{x}_{u,k}}{\lambda_{u,k}} - a'\right|^2\right),$$

where $\Omega_p^0$ and $\Omega_p^1$ are the sets of constellation points for which the $p^{th}$ bit equals to 0 and 1, respectively, $\lambda_{u,k} = diag(\mathbf{A}_k^{-1}\mathbf{G}_k)_u$ is the effective channel gain ( $diag(\cdot)_u$ indicates the $u^{th}$ diagonal entry of a matrix ) and $\rho_{u,k} = \lambda_{u,k}^2 / v_{u,k}^2$ is the post-equalization signal-to-interference-plus-noise ratio (SINR) with $v_{u,k}^2 = E_s\lambda_{u,k} - E_s|\lambda_{u,k}|^2$.

### C. Algorithm Complexity Reduction

We next detail methods to reduce the complexity of soft-output MMSE detection, without noticeably degrading its performance.

*1) Equalization:* To calculate the MMSE estimate of the transmit signal $\hat{x}_k$, a direct way would be to first compute $y_k^{MF}$ and $\mathbf{A}_k^{-1}$, respectively, where the inverse $\mathbf{A}_k^{-1}$ can be computed via the Cholesky decomposition followed by forward and backward substitution [6]. A low-complexity alternative to obtain $\hat{x}_k$ is to solve the linear equation $\mathbf{A}_k \hat{x}_k = y_k^{MF}$ for $\hat{x}_k$ using iterative algorithms. As shown in [8], the conjugate gradient (CG) method enables one to solve such linear systems at low complexity for massive MIMO systems. Lines 7–15 of Algorithm 1 show the CG method that solves for $\hat{x}_k$ with inputs $\mathbf{A}_k$ and $y_k^{MF}$. While CG delivers the exact solution to the system of linear equations in $U$ iterations, fewer iterations are required in practice to achieve an accurate estimate.

*2) Soft-Output Computation:* With a Cholesky-based inversion approach, we can calculate the LLR values using $\mathbf{A}_k^{-1}$. CG does not require the inverse $\mathbf{A}_k^{-1}$, but does not provide information on the effective channel gain $\lambda_{u,k}$ and post-equalization SINR $p_{u,k}$ for LLR

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

**Algorithm 1** Reconfigurable soft-output MMSE MIMO detection

```
1:  Input: $\mathbf{H}_k$ and $y_k$    /*$k$ is subcarrier index*/
2:  Calculate $\mathbf{G}_k, \mathbf{A}_k, y_k^{MF}$    /*Preprocessing stage*/
3:  if Chol.-based detector then    /*Equalization stage*/
4:      Chol. decomp. & FW./BW. substitution for $\mathbf{A}_k^{-1}$
5:      Calculate $\hat{x_k} = \mathbf{A}_k^{-1} y_k^{MF}$
6:  else    /*CG-based detector*/
7:      Init: $\mathbf{x}_k^{(0)} = 0, \mathbf{r}_k^{(0)} = y_k^{MF}, \mathbf{t}_k^{(0)} = \mathbf{r}_k^{(0)}$
8:      for $i = 1, 2, \cdots, I$ do    /*$I$: CG iter. for a close approx.*/
9:          $\mathbf{s}_k^{(i-1)} = \mathbf{A}_k t_k^{(i-1)}$
10:         $\alpha_k^{(i)} = \|\mathbf{r}_k^{(i-1)}\|^2 / ((\mathbf{t}_k^{(i-1)})^H \mathbf{s}_k^{(i-1)})$
11:         $\mathbf{x}_k^{(i)} = \mathbf{x}_k^{(i-1)} + \alpha_k^{(i)} \mathbf{t}_k^{(i-1)}$
12:         $\mathbf{r}_k^{(i)} = \mathbf{r}_k^{(i-1)} - \alpha_k^{(i)} \mathbf{s}_k^{(i-1)}$
13:         $\beta_k^{(i)} = \|\mathbf{r}_k^{(i)}\|^2 / \|\mathbf{r}_k^{(i-1)}\|^2$
14:         $\mathbf{t}_k^{(i)} = \mathbf{r}_k^{(i)} + \beta_k^{(i)} \mathbf{t}_k^{(i-1)}$    /*end for*/
15:     $\hat{x_k} = \mathbf{x}_k^{(I)}$    /*end if-else*/
16: if Chol.-based detector then    /*Soft-output comp. stage*/
17:     Get $\lambda_{u,k} = diag(\mathbf{A}_k^{-1}\mathbf{G}_k)_u$ and $\rho_{u,k}, \forall u$
18: else    /*CG-based detector*/
19:     Calculate $\rho_{u,k}$ then $\lambda_{u,k}, \forall u$    /*end if-else*/
20: Calculate the LLR values
21: Output: $L_{u,k}^p, \forall p, u$
```

computation. Here, we use a simple yet effective way to approximate the LLR values without the need of explicitly computing $\mathbf{A}_k^{-1}$.

Our approximation is as follows. Since the diagonal entries of $\mathbf{A}_k^{-1}$ and $\mathbf{G}_k$, $\forall k$, are all real numbers, $\lambda_{u,k} = diag(\mathbf{A}_k^{-1}\mathbf{G}_k)_u$, $\forall u, k$, is also a real number. We furthermore have $\rho_{u,k} = \lambda_{u,k}^2 / v_{u,k}^2 = \frac{\lambda_{u,k}}{E_s - E_s\lambda_{u,k}}$ so that $\lambda_{u,k} = E_s\rho_{u,k}/(1 + E_s\rho_{u,k})$. To get $\rho_{u,k}$ and $\lambda_{u,k}$, we first approximate $\rho_{u,k}$ by the $u^{th}$ diagonal entry of $\mathbf{G}_k$ as put forward in [8]: $\rho_{u,k} \approx \mathbf{G}_k(u,u)/N_0$, which only requires the diagonal elements of $\mathbf{G}_k$ and $N_0$. We then compute the approximation for $\lambda_{u,k}$ as derived above based on the approximated $\rho_{u,k}$. Finally, we can compute approximate LLR values. Algorithm 1 summarizes our algorithm, which offers two modes: exact Cholesky-based MMSE detection and approximate CG-based detection.

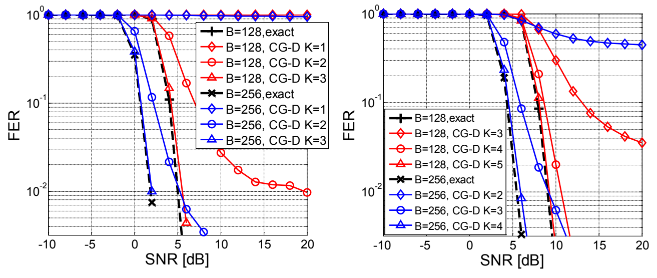
### D. *Simulation Results*

We simulate both the Cholesky-based MMSE detector and CG-based approximated MMSE detector (CG-D) discussed above in a massive MIMO 128-subcarrier OFDM system, and compare the frame error rates (FERs) under different BS and user antenna number configurations. The information bits are encoded with a 5/6-rate convolutional code, modulated to 16-QAM, and transmitted through a channel matrices obtained from the WINNER-Phase-2 model [9]. The BS is equipped with a linear antenna array at an antenna spacing of $10m/128 \approx 0.0781m$, and incorporated with our soft-output MMSE detector and a soft-input max-log Viterbi decoder. Figure 1(a)(b) shows the FER performance at for 16 and 32 user antennas.

## III. GPU Implementation

### A. *Reconfigurable detector architecture overview*

We implement our detector on a GPU according to Algorithm 1 with compute unified device architecture (CUDA) [10]. Figure 2 shows the reconfigurable architecture of our unified soft-output MMSE detector, which includes two modes: an exact Cholesky-based detection mode and a CG-based approximate detection mode. Both modes have three major computation stages: (i) a preprocessing stage to compute the matched filter $y_k^{MF} = \mathbf{H}_k^H y_k$, the Gram matrix $\mathbf{G}_k$, and the matrix $\mathbf{A}_k$; (ii) an equalization stage to get the exact or approximated estimation of transmit signal $\hat{x}_k$ using the Cholesky decomposition or CG; (iii) a soft-output calculation stage to compute LLR values. In what follows, $k \in \{1, 2, \ldots, N_{scr}\}$, where $N_{scr}$ indicates the number of subcarriers in an OFDM symbol. We define $N_{sym}$ as the number of $N_{scr}$-subcarrier OFDM symbols. At the input of the



(a) 16 users    (b) 32 users

Figure 1: Frame error rate (FER) performance comparison at 16QAM (B:basestation antenna number, K: CG iteration number).

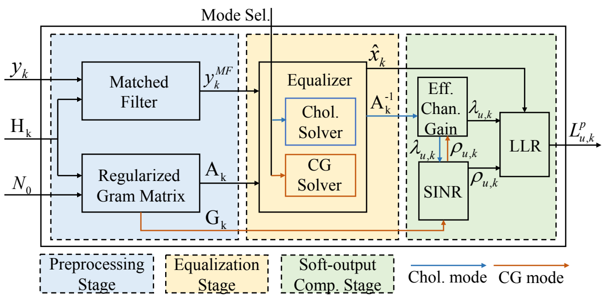


Figure 2: Detector architecture and dataflow.

detector, 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 receiver (RX), each $y_k$ including $B$ samples corresponding to $B$ antennas of BS. We also prepare $N_{sym}$ sets of channel matrices $\{\mathbf{H}_1, \mathbf{H}_2, \ldots, \mathbf{H}_{N_{scr}}\}$ for that frame, as well as the control signals for mode switching. At the output, our implementation generates LLR values for every user at a time for a certain streaming frame. 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.

### B. *Kernel Design and Optimization*

*1) Preprocessing Stage:* We calculate the matrices $\mathbf{G}_k$ and $\mathbf{A}_k$, and the vector $y_k^{MF}$ in a per-subcarrier basis. Since calculating $\mathbf{G}_k$, $\mathbf{A}_k$, and $y_k^{MF}$ exhibits no data dependency for each subcarrier, we fetch $N_{CR} = N_{sym} \times N_{scr}$ channel matrices $\mathbf{H}_k$ and vectors $y_k$ from the GPU's global memory, and calculate a batch ($batchSize = N_{CR}$) of $\mathbf{G}_k, \mathbf{A}_k$ 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 $\mathbf{G}_k$ and $y_k^{MF}$ are similar, i.e., matrix-matrix (vector) multiplication, which can be efficiently implemented by the $cuBLAS$ [10] library, a GPU accelerated Basic Linear Algebra Subprograms (BLAS) library. Specifically, we choose $cublasCgemmBatched()$ function, which is fine tuned for computing a batch of small complex matrix multiplications in the format of $\mathbf{C}_i = \mathbf{A}_i\mathbf{B}_i$, where $i$ is the matrix index, and $\mathbf{A}, \mathbf{B}, \mathbf{C}$ are complex matrices or complex vectors. To avoid unnecessary matrix transpose data movement, and to correctly utilize the $cuBLAS$ function, we directly store the $\mathbf{H}_k$ and $y_k$ in column-major format in GPU global memory as the input of the function. After a batch of matrices $\mathbf{G}_k$ have been computed, we can easily form the matrix $\mathbf{A}_k$ by a light kernel function, which adds a constant $N_0/E_s$ to each diagonal entry of $\mathbf{G}_k$ matrix in parallel for all $N_{CR}$ subcarriers.

*2) Equalization Stage:* In this stage, our detector supports a Cholesky-based inversion, which explicitly calculates $\mathbf{A}_k^{-1}$, as

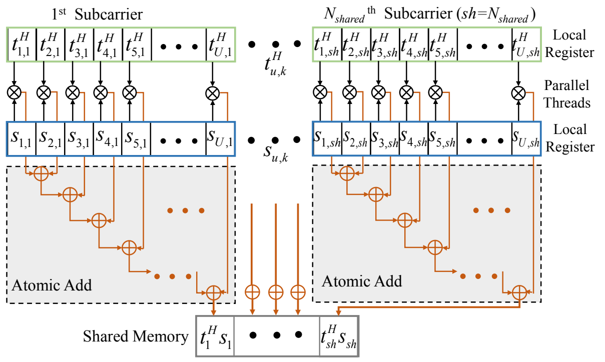


Figure 3: Reduction to shared memory by atomic operations.

well as a CG-based equalizer which approximates the MMSE estimate $\hat{x}_k$ without ever forming $\mathbf{A}_k^{-1}$. 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 $\mathbf{A}_k$, and $cublasCgetriBatched()$ to perform the forward and backward substitutions to get $\mathbf{A}_k^{-1}$. Then we use $cublasCgemmBatched()$ again to calculate estimated $\hat{x}_k$ by $\hat{x}_k = \mathbf{A}_k^{-1} y_k^{MF}$. 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 CG-based equalizer, most of the computations are vector additions or multiplications, except for the calculation of $\mathbf{s}_k^{(i)}$ at the beginning of every $i^{th}$ iteration. For $\mathbf{s}_k^{(i)}$, we can calculate $batchSize = N_{CR}$ number of $\mathbf{s}_k^{(i)} = \mathbf{A}_k \mathbf{t}_k^{(i)}$ 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 $\mathbf{t}_k^{(i)}, \mathbf{r}_k^{(i)}$ and the approximate $\mathbf{x}_k^{(i)}$ 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, $\mathbf{t}_k^{(i)}$ is a vector constructed as $[t_{1,k}^{(i)}, t_{2,k}^{(i)}, \cdots, t_{U,k}^{(i)}]^T$, where the element $t_{u,k}^{(i)}$ 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 $\mathbf{x}_k^{(i)} = \mathbf{x}_k^{(i-1)} + \alpha_k^{(i)} \mathbf{t}_k^{(i-1)}$ 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 $(\mathbf{t}_k^{(i)})^H \mathbf{s}_k^{(i)}$ or squares of vector magnitude such as $\|\mathbf{r}_k^{(i)}\|^2$, which requires data sharing between multiple parallel threads. For example, to calculate $(\mathbf{t}_k^{(i)})^H \mathbf{s}_k^{(i)}$, 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 $N_{CR}/N_{shared}$ thread blocks with $N_{shared} \times U$ threads in each block. Here, $N_{shared}$ 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 $atomic$ 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 $Maxwell$ architecture, atomic operations have been further enhanced for shared memory [10]. We take advantage of this new feature and perform $atomicAdd$ 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 $N_{shared}$ sets of such shared units are deployed for handling $N_{shared} \times U$ threads in each thread block. We finally have $N_{shared}$ 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. Figure 3 shows an example of the reduction process by serialized atomic operations on certain shared memories within a thread block when calculating $(\mathbf{t}_k^{(i)})^H \mathbf{s}_k^{(i)}$. The CG iteration index $i$ is omitted for convenience in the figure.

*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 $\mathbf{A}_k^{-1}$ for the CG-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.

## C. Multi-Stream Scheduling

We have discussed how to design and optimize various computation kernels for their efficient execution, but usually the CPU-GPU memory copy of input or output data will 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.

Consider that we have $N_{frame}$ number of streaming frames received continuously in a real-time uplink receiver, for a certain frame, the payload data has $N_{sym}$ frequency domain OFDM symbols, each including $N_{scr}$ subcarrier signals $\{y_1, y_2, \ldots, y_{N_{scr}}\}$. 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

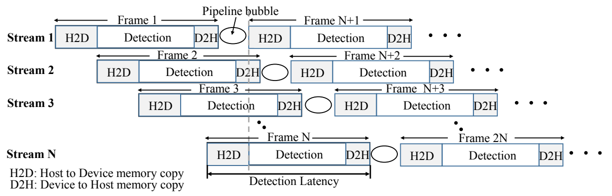


Figure 4: Multi-stream scheduling on processing streaming frames.

Table I: Single Stream Throughput Performance

| $128 \times 16$ (in Mb/s) | | | $256 \times 32$ (in Mb/s) | | |
|---|---|---|---|---|---|
| $N_{sym}$ | Chol | CG@3iter | $N_{sym}$ | Chol | CG@4iter |
| | K/(K+M) | K/(K+M) | | K/(K+M) | K/(K+M) |
| 8 | 88.92/61.80 | 106.74/71.58 | 2 | 36.98/16.96 | 39.10/20.31 |
| 16 | 106.56/78.39 | 127.87/88.86 | 4 | 43.63/24.98 | 49.16/28.97 |
| 32 | 118.19/90.74 | 144.51/102.65 | 8 | 52.14/34.02 | 57.76/39.37 |
| 64 | 130.45/102.51 | 155.02/113.93 | 16 | 54.15/40.82 | 58.09/45.78 |

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

need to copy all the $N_{sym}$ OFDM payload symbols from host, we only need to copy $N_{scr}$ channel matrices $\{\mathbf{H}_1, \mathbf{H}_2, \ldots, \mathbf{H}_{N_{scr}}\}$ 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 $Device\ To\ Device\ Memcpy$ 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 $N_{frame}$ RX frames onto $N_{stream}$ streams, the task of $f^{th}$ frame will be dispatched to the $(f \bmod N_{stream})^{th}$ stream, with the Hyper-Q feature enabled in the current Maxwell architecture GPU to avoid false dependencies between multiple streams [10]. 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. Figure 4 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 $N_{stream}$ experimentally to reduce pipeline bubbles for near-optimal pipeline performance.

### *D. 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 multi-GPU extension in our detector design, 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 tasks can be deployed evenly and performed concurrently on multiple GPUs, further improving the detection throughput.

## IV. Experimental Results and discussions

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 CUDA Toolkit 7.0 running on Linux 64-bit OS for the design implementation, debugging and performance profiling. The experimental results are based on an Nsight release version of the design with O3 compiler optimization.

In Table I, we record the 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 CG-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 utilization rate of GPU computing resources with more workloads fed. 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% throughput 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 CG-based detector with a proper iteration number, e.g., minimum iteration for <1dB SNR loss at $10^{-2}$ FER compared to Cholesky-based detector, achieves better throughput at different configurations of antenna numbers and detection workloads, so that one can trade-off error performance with throughput by reconfiguring the detection mode of our design.

Table II: Multi-stream & Multi-GPU Enhanced Performance

| $N_{GPU}$ | $N_{stream}$ | $N_{sym}$ | Detector mode | T (Mb/s) | L (ms) |
|---|---|---|---|---|---|
| 1 | 8 | 8 | $128 \times 16$ Chol. | 131.62 | 3.98 |
| | | | $128 \times 16$ CG@3iter | 148.67 | 3.54 |
| 2 | 4 | 16 | $128 \times 16$ Chol | 265.94 | 3.94 |
| | | | $128 \times 16$ CG@3iter | 286.57 | 3.66 |

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

In Table II, 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 experiments, we record a proper combination of $N_{stream}$ and $N_{sym}$ for a near optimal task 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 I. By using two GPUs, the total throughput can be enhanced by around two times, and our CG-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. 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.

## V. Conclusion

We have designed a GPU-based uplink detector targeting real-time massive MIMO SDR systems. Our MMSE soft-output detector supports various antenna configurations and performance/complexity trade-offs by supporting two detection modes. Our design achieves over 250 Mb/s throughput with less than 4 ms latency by using two of the latest GPUs for supporting a $128 \times 16$ antenna system, demonstrating the efficacy of GPU-accelerated massive MIMO SDR systems.

## References

[1] F. Rusek and et al., "Scaling up MIMO: Opportunities and challenges with very large arrays," *IEEE SPM*, pp. 40–60, Jan 2013.

[2] M. Wu and et al., "Large-scale MIMO detection for 3GPP LTE: Algorithms and FPGA implementations," *IEEE J-STSP*, pp. 916–929, Oct 2014.

[3] B. Yin and et al., "A 3.8Gb/s large-scale MIMO detector for 3GPP LTE-advanced," in *ICASSP*, May 2014, pp. 3879–3883.

[4] G. Wang and et al., "High throughput low latency LDPC decoding on GPU for SDR systems," in *GlobalSIP*, Dec 2013, pp. 1258–1261.

[5] M. Wu and et al., "Implementation of a 3GPP LTE turbo decoder accelerator on GPU," in *SIPS*, Oct 2010, pp. 192–197.

[6] C. Studer and et al., "ASIC implementation of soft-input soft-output MIMO detection using MMSE parallel interference cancellation," *IEEE JSSC*, pp. 1754–1765, July 2011.

[7] A. Paulraj and et al., "Introduction to space-time wireless communications," in *New York, USA: Cambridge University Press, 2008.*

[8] B. Yin and et al., "Conjugate gradient-based soft-output detection and precoding in massive MIMO systems," in *GLOBECOM*, Dec 2014, pp. 3696–3701.

[9] *WINNER Phase II Model*. [Online]. Available: http://www.ist-winner.org/WINNER2-Deliverables/D1.1.2v1.1.pdf

[10] *Nvidia CUDA programming guide*. [Online]. Available: http://docs.nvidia.com/cuda/