Noname manuscript No.
(will be inserted by the editor)

GPU Acceleration of a Configurable N-Way MIMO Detector for Wireless Systems

Michael Wu · Bei Yin · Guohui Wang · Christoph Studer · Joseph R. Cavallaro

Received: date / Accepted: date

Abstract Multiple-input multiple-output (MIMO) wireless is an enabling technology for high spectral efficiency and has been adopted in many modern wireless communication standards, such as 3GPP-LTE and IEEE 802.11n. However, (optimal) maximum a-posteriori (MAP) detection suffers from excessively high computational complexity, which prevents its deployment in practical systems. Hence, many algorithms have been proposed in the literature that trade-off performance versus detection complexity. In this paper, we propose a flexible N-Way MIMO detector that achieves excellent error-rate performance and high throughput on graphics processing units (GPUs). The proposed detector includes the required QR decomposition step and a tree-search detector, which exploits the massive parallelism available in GPUs. The proposed algorithm performs multiple tree searches in parallel, which leads to excellent error-rate performance at low computational complexity on different GPU architectures, such as Nvidia Fermi and Kepler. We highlight the flexibility of the proposed detector and demonstrate that it achieves higher throughput than existing GPU-based MIMO detectors while achieving the same or better error-rate performance.

1 Introduction

Multiple-input multiple-output (MIMO) wireless is a key technology used in many modern communication standards, such as 3GPP-LTE, WiMAX, and IEEE 802.11n. The use of multiple antennas at both ends of the wireless link enables significant improvements (compared to single-antenna systems) in terms of spectral efficiency. Most standards employing MIMO use spatial multiplexing, which transmits multiple independent data streams concurrently and within the same frequency band. The received signal mixture (caused by multi-path propagation) necessitates a MIMO detector, which separates the received mixture of transmitted data streams. Traditionally, corresponding detector solutions are implemented on ASICs or FPGAs, due to the excessive complexity of MIMO detection. In this paper, we aim to utilize off-the-shelf graphical processing units (GPUs) in order to perform high throughput MIMO detection. The massive amount of computational power provided by GPUs is particularly useful for platforms in need of great flexibility, such as software defined radios, or to speed up MIMO system simulations.

Michael Wu · Bei Yin · Guohui Wang · Joseph R. Cavallaro
Rice University, Houston, TX, USA, 77005
E-mail: {mbw2, by2, gw2, cavallar}@rice.edu
Christoph Studer
Cornell University, Ithaca, NY, USA, 14850
E-mail: studer@cornell.edu

1.1 MIMO Detection

Among different MIMO detection schemes, soft-output maximum a-posteriori (MAP) detection is the optimal detection scheme in coded systems. This nonlinear detector requires an exhaustive search over all candidate vectors, which results in prohibitive computational complexity, even for MIMO systems transmitting a few spatial streams. Consequently, exact MAP detection is impractical as wireless systems typically have stringent hardware constraints (silicon area and power consumption), as well as challenging throughput and latency requirements. As a result, most practical solutions to

MIMO detection rely on suboptimal but low complexity algorithms [2–4, 10, 11, 16, 18, 23].

Suboptimal detection algorithms can mainly be categorized into linear detection algorithms and non-linear tree-search-based methods. Although the complexity of linear detection methods is very low, the associated error-rate performance is rather poor in practical systems [4, 11, 23]. Non-linear tree-search-based detection methods, however, are capable of achieving excellent error-rate performance at low computational complexity. Such detectors can mainly be divided into two categories: (i) depth-first tree-search algorithms, such as sphere decoding [3, 16], and (ii) breadth-first search algorithms, such as the K-best algorithm [18]. For both approaches, the search space, i.e., the set of all possible transmit vectors, can be represented as a tree. To reduce the computational complexity of data detection, these detectors use heuristics to eliminate useless branch extensions during the tree-search process.

Depth-first sphere detectors traverse the tree from top to bottom recursively and prune branches with large partial distances during backtracking. As the traversal path is not deterministic, parallel implementations of depth-first search require load balancing to achieve high efficiency [8]; this requires global synchronization which is inefficient on GPUs. In addition to the random runtime of depth-first tree-search algorithms, they usually evaluate a large and small number of tree branches at low and high SNR, respectively. Breadth-first search algorithms, such as the K-best algorithm, reduce the complexity by pruning branches level by level. Although the K-best algorithm has a deterministic run-time, the algorithm requires a global sort at each tree level to find the best K nodes, which is the main bottleneck for corresponding software implementations [7].

More recently, the authors in [10] proposed the selective spanning with fast enumeration (SSFE) MIMO detection algorithm, which can be viewed as a sort-free approximation to the K-best algorithm; related MIMO detection algorithms were also proposed in [2, 5]. The SSFE method, however, results in a substantial error-rate performance loss. In order to recover part of this performance loss, one can run a small number of instances of SSFE in parallel, where each instance operates with a different permuted detection order [13, 20]. Since these instances perform the same set of operations but work on different input data, this improved algorithm maps very well onto GPU architectures [14, 22].

1.2 Contributions

Our contributions are as follows. We propose a flexible MIMO detector that achieves excellent error-rate performance and high throughput on GPUs. We show that the proposed design achieves a wide range of trade-offs between throughput and error-rate performance, and is able to approach the error-rate performance of the optimal soft-output MAP detector within 0.25 dB. We then describe our complete implementation, including both the required QR decomposition and the MIMO detection algorithm. We optimize our QR decomposition kernel for processing on many small dense matrices in parallel, which is different from conventional decomposition methods, such as the one in [9]. In addition, we improve the throughput of our previous work in [22] using a variety of optimizations. We furthermore characterize the achieved throughput of our kernel on both Fermi and Kepler GPUs. Finally, we show that our implementation on Nvidia GPUs achieves a substantially higher throughput than existing soft-output MIMO detectors implemented on GPUs [14, 19].

1.3 Outline of the Paper

This paper is organized as follows. Section 2 introduces the MIMO system model. Section 3 describes the proposed detection algorithm and its GPU implementation. Section 4 characterizes the performance and complexity of the detector. We conclude in Section 5.

2 MIMO System Model

The considered MIMO system transmits N_t independent data streams, and the destination receives signals on N_t antennas. At the transmit-side, given a binary-valued vector $\mathbf{x} = [x_0, \ldots, x_{L-1}]^T$ with $L = N_t \log_2 M$, the modulation function maps the vector $\mathbf{x}$ to $\mathbf{s} = [s_0, \ldots, s_{N_t-1}]^T$, where s_i is a complex number in a finite constellation alphabet $\mathbf{\Omega}$ with cardinality M. For example, the constellation alphabet for QPSK is $\{-1-j, -1+j, 1-j, 1+j\}$ with $M = 4$. The source then transmits the modulated signal vector $\mathbf{s}$ over N_t antennas. The received symbols can be modeled as

$$\mathbf{y} = \mathbf{H}\mathbf{s} + \mathbf{n}, \tag{1}$$

where $\mathbf{y} = [y_0, \ldots, y_{N_t-1}]^T$ is the received symbol vector, $\mathbf{H} = [\mathbf{h}_0, \ldots, \mathbf{h}_{N_t-1}]$ is the $N_t \times N_t$ channel matrix. We consider a Rayleigh fading channel model, where each entry of $\mathbf{H}$, denoted by h_{ij}, is modeled by an i.i.d. circularly symmetric complex Gaussian (ZM-CSCG) random variable with variance σ_h^2 per complex dimension. Each element of the additive noise vector, $\mathbf{n} = [n_0, \ldots, n_{N_t-1}]^T$, is assumed to be i.i.d. ZMCSCG with variance σ_n^2 per complex dimension.

2.1 Modified Real-Valued Decomposition

To perform MIMO detection in the real domain instead of the complex domain, we first perform a real-valued decomposition of the input-output relation (1). Specifically, we rewrite (1) as

$$\begin{pmatrix} \Re(\mathbf{y}) \\ \Im(\mathbf{y}) \end{pmatrix} = \begin{pmatrix} \Re(\mathbf{H}) & -\Im(\mathbf{H}) \\ \Im(\mathbf{H}) & \Re(\mathbf{H}) \end{pmatrix} \begin{pmatrix} \Re(\mathbf{s}) \\ \Im(\mathbf{s}) \end{pmatrix} + \begin{pmatrix} \Re(\mathbf{n}) \\ \Im(\mathbf{n}) \end{pmatrix}, \quad (2)$$

where $\Re(x)$ and $\Im(x)$ denote the real and imaginary part of the complex variable x, respectively. In order to improve the error-rate performance of the proposed detector, we deploy the modified real-valued decomposition (MRVD) put forward in [1]. In particular, we permute the vector and matrix elements such that the real and imaginary part of the same complex entry are adjacent to each other. With this, the resulting input-output relation is given by

$$\begin{pmatrix} \Re(y_0) \\ \Im(y_0) \\ \Re(y_1) \\ \Im(y_1) \\ \vdots \\ \Re(y_{N_R-1}) \\ \Im(y_{N_R-1}) \end{pmatrix} = \widetilde{\mathbf{H}} \begin{pmatrix} \Re(s_0) \\ \Im(s_0) \\ \Re(s_1) \\ \Im(s_1) \\ \vdots \\ \Re(s_{N_R-1}) \\ \Im(s_{N_R-1}) \end{pmatrix} + \begin{pmatrix} \Re(n_0) \\ \Im(n_0) \\ \Re(n_1) \\ \Im(n_1) \\ \vdots \\ \Re(n_{N_R-1}) \\ \Im(n_{N_R-1}) \end{pmatrix}$$

which we abbreviate by $\widetilde{\mathbf{y}} = \widetilde{\mathbf{H}}\hat{\mathbf{s}} + \widetilde{\mathbf{n}}$.

Compared to the original (complex-valued) system model in (1), MRVD doubles the number of elements in each vector and quadruples the dimensionality of $\widetilde{\mathbf{H}}$. Furthermore, each element of $\widetilde{s}_i$ is drawn from a smaller (real-valued) alphabet, $\mathbf{\Omega}'$, which has cardinality $Q = \sqrt{M}$. For example, with QPSK, the MRVD-equivalent constellation alphabet is $\{-1, +1\}$ with $Q = 2$.

2.2 Soft-Output MIMO Detection

Given the received vector after performing the MRVD, $\widetilde{\mathbf{y}}$, and the MRVD-equivalent channel matrix $\widetilde{\mathbf{H}}$, the soft-output MIMO detector at the receiver computes the a-posteriori probability log-likelihood (log-APP) ratio, L_D^k, for each bit. Assuming equally likely transmitted bits, the log-likelihood ratio (LLR) of the k^{th} transmitted bit can be approximated via the max-log approximation [6]:

$$L_D^k \approx \frac{1}{2\sigma_n^2} \left(\min_{\mathbf{x}\in\mathbb{X}_{k,0}} \left\| \widetilde{\mathbf{y}} - \widetilde{\mathbf{H}}\widetilde{\mathbf{s}} \right\|^2 - \min_{\mathbf{x}\in\mathbb{X}_{k,1}} \left\| \widetilde{\mathbf{y}} - \widetilde{\mathbf{H}}\widetilde{\mathbf{s}} \right\|^2 \right). \quad (3)$$

Here, $\mathbb{X}_{k,0}$ is the set of all binary-valued vectors with the k^{th} bit equals to 0, and $\mathbb{X}_{k,1}$ is the set of all binary vectors with the k^{th} bit equal to 1. For the sake of brevity, the vector $\widetilde{\mathbf{s}}$ corresponds to the modulated binary vector $\mathbf{x}$. The number of binary vectors in $\mathbb{X}_{k,0}$ and $\mathbb{X}_{k,1}$ scales exponentially with N_t. As a result, the computational complexity of evaluating (3) scales exponentially with N_t.

To reduce the complexity of (3), we can approximate L_D^k with a reduced set of transmit vectors, or a candidate list, $\mathcal{L}$. This candidate list, $\mathcal{L}$, is generated by excluding transmit vectors with large Euclidean distances. We then split $\mathcal{L}$ into two sublists for $\mathcal{L}_{k,0}$ and $\mathcal{L}_{k,1}$. The list $\mathcal{L}_{k,0}$ contains the candidates with the k^{th} bit equal to 0 while the list $\mathcal{L}_{k,1}$ contains the candidates with the k^{th} bit equal to 1. We then approximate L_D^k via (4), where the list $\mathcal{L}_{k,0}$ is used for computing the 0-hypothesis part of the L_D^k, while the list $\mathcal{L}_{k,+1}$ is used for computing the 1-hypothesis part of the L_D^k.

$$L_D^k \approx \frac{1}{2\sigma_n^2} \left(\underbrace{\min_{\mathbf{x}\in\mathcal{L}_{k,0}} \left\| \widetilde{\mathbf{y}} - \widetilde{\mathbf{H}}\widetilde{\mathbf{s}} \right\|^2}_{\text{0-hypothesis}} - \underbrace{\min_{\mathbf{x}\in\mathcal{L}_{k,1}} \left\| \widetilde{\mathbf{y}} - \widetilde{\mathbf{H}}\widetilde{\mathbf{s}} \right\|^2}_{\text{1-hypothesis}} \right). \quad (4)$$

3 Soft-Output *N*-Way MIMO Detector on Nvidia GPU

Most wireless standards employ orthogonal frequency division multiplexing (OFDM), which simplifies equalization by dividing the available bandwidth into multiple orthogonal subcarriers. With OFDM, each subcarrier corresponds to an independent MIMO detection problem. As a result, the receiver needs to perform MIMO detection on every subcarrier. Since many wireless standards use a large number of subcarriers and GPUs consist of many independent processing cores, GPUs are very suitable for this application as hundreds of independent MIMO detection problems can run in parallel on GPUs, which allows one to achieve high throughput.

Since Nvidia GPUs can be viewed as multi-core SIMD processors, a suitable algorithm needs to be data parallel to maximize the use of the available execution units. In addition, the device memory latency is high on GPUs. To reduce the memory latency, a small amount of on-chip resources, such as registers and shared memory, can be used. As a result, a good algorithm needs to have a memory footprint small enough to fit into the available on-chip memory to reduce the number of expensive device-memory accesses.

We implemented the soft-output N-way MIMO detector on GPU using the CUDA C programming lan-

Algorithm 1 Modified Gram-Schmidt CUDA Kernel: computations performed at the k^{th} thread.

```
1.  Input: $\mathbf{y}, \mathbf{H}$
2.  Initialization:
    (a) $s = 0$, where $s$ is stored in shared memory
    (b) Fetch $\mathbf{y}$ and $\mathbf{H}$ to construct $\mathbf{V} = [\widetilde{\mathbf{H}}|\widetilde{\mathbf{y}}]$ in shared memory
3.  for step $i = 0$ to $2N_t - 1$ do
4.      if $(k = i)$
5.          $e_{i,i} = \|\mathbf{v}_i\|^2$
6.          $s = 1/\sqrt{e_{i,i}}$
7.      end if
8.      __syncthreads()
9.      $v_{k,i} = v_{k,i} \cdot s$
10.     __syncthreads()
11.     if $(k \geq i)$
12.         $e_{i,k+1} = \mathbf{v}_i^H \mathbf{v}_{k+1}$
13.         $\mathbf{v}_{k+1} = \mathbf{v}_{k+1} - \mathbf{v}_i \cdot e_{i,k+1}$
14.     end if
15. end for
```

guage.[1] In this programming model, the programmer specifies a kernel, or a set of computations. At runtime, threads execute the same set of computations specified by the kernel on different input data to perform the task. The proposed MIMO detector implementation consists of two kernels. The first kernel performs a QR decomposition on the channel matrix. The second kernel searches for likely transmit vectors via a generated candidate list. Then, the algorithm uses this candidate list to compute soft-output information for each transmitted bit. In the following two sections, these two kernels are described in detail.

3.1 QR Decomposition

To reduce the complexity of the candidate search, we first perform QR decomposition on $\widetilde{\mathbf{H}}$. With this, we can rewrite (4) as follows:

$$L_D^k \approx \frac{1}{2\sigma_n^2}\Bigg(\underbrace{\min_{\mathbf{x}\in\mathcal{L}_{k,0}} \|\hat{\mathbf{y}} - \mathbf{R}\widetilde{\mathbf{s}}\|^2}_{\text{0-hypothesis}} - \underbrace{\min_{\mathbf{x}\in\mathcal{L}_{k,1}} \|\hat{\mathbf{y}} - \mathbf{R}\widetilde{\mathbf{s}}\|^2}_{\text{1-hypothesis}}\Bigg). \quad (5)$$

Here, $\mathbf{R}$ is the upper-triangular matrix obtained from the QR decomposition applied to $\widetilde{\mathbf{H}}$, and $\hat{\mathbf{y}}$ is the effective received vector obtained from $\mathbf{Q}^T\widetilde{\mathbf{y}}$.

To achieve low computational complexity and good numerical stability, we implemented the Modified Gram-Schmidt QR factorization [17] on the GPU. Algorithm 1 shows the pseudo-code for the modified Gram-Schmidt CUDA kernel function, which corresponds to a parallel implementation of the modified Gram-Schmidt algorithm. At runtime, $2N_t$ threads execute the set of instructions defined in Algorithm 1 to perform one QR decomposition. The kernel function can be summarized as follows. In the initialization part, $2N_t$ threads fetch the complex-valued inputs, the received signals $\mathbf{y}$ and the channel $\mathbf{H}$, from device memory. To perform MRVD, the threads use the input data to construct a real-valued extended matrix, $\mathbf{V} = [\widetilde{\mathbf{H}}|\widetilde{\mathbf{y}}] = [\mathbf{v}_0, \ldots, \mathbf{v}_{2N_t}]$, in shared memory. The vector $\mathbf{v}_i$ is the i^{th} column of $\mathbf{V}$ and the scalar $v_{k,i}$ is the k^{th} entry of $\mathbf{v}_i$.

Next, $2N_t$ threads perform QR decomposition on the extended matrix $\mathbf{V}$. The result is an extended upper triangular matrix $\mathbf{E} = [\mathbf{R}|\hat{\mathbf{y}}]$ which is stored in device memory, where the scalar $e_{k,i}$ is the k^{th} element of the i^{th} column of $\mathbf{E}$. The whole process consumes $2N_t$ iterations. Each iteration consists of a set of serial operations and a set of parallel operations.

The serial operations are summarized in Lines 4–7 of Algorithm 1, where we compute $\|\mathbf{v}_i\|^2$, the squared ℓ_2 norm of the i^{th} column of $\mathbf{V}$, and the corresponding scaling factor s. These serial computations are handled by one thread. In our implementation, these serial computations are handled by the i^{th} thread.[2]

Subsequent computations are done in parallel. In Line 9, $2N_t$ threads compute the i^{th} orthogonal projection, $\mathbf{v}_i$, in parallel. In Lines 11-14, we assign one thread to each column of $\mathbf{V}$ and the number of columns (of $\mathbf{V}$) updated decreases by one after each iteration. For the i^{th} iteration, only threads $k \geq i$ update the columns of $\mathbf{V}$, which leads to the condition $k \geq i$ in Line 11. Line 12 constructs the remaining elements in the i^{th} row of $\mathbf{E}$ in parallel. In Line 13, the matrix $\mathbf{V}$ is updated for subsequent iterations. In this instance, the k^{th} thread updates $\mathbf{v}_{k+1}$ by subtracting the projection of $\mathbf{v}_{k+1}$ on to the $\mathbf{v}_i$ from $\mathbf{v}_{k+1}$.

The matrix $\mathbf{V}$ is stored in shared memory as elements of $\mathbf{V}$ are accessed repeatedly. Storing $\mathbf{V}$ in shared memory is much faster than storing the matrix in device memory. Storing the entire matrix in shared memory is possible as the number of elements in the matrix $\mathbf{V}$ is small for typical MIMO systems. Furthermore, the memory access pattern of this kernel is very regular. A column-major layout for $\mathbf{V}$ results in bank-conflict-free shared memory accesses.

3.2 1-Way MIMO Detection

Given $\hat{\mathbf{y}}$ and $\mathbf{R}$, the MIMO detector computes LLR values in two steps. The first step finds candidate vectors

[1] We assume the reader is familiar with CUDA. A detailed description and explanation can be found in [12].

[2] The serial computations can be handled by any thread. For example, it is possible to always pick the 1st thread to compute the squared ℓ_2-norm.

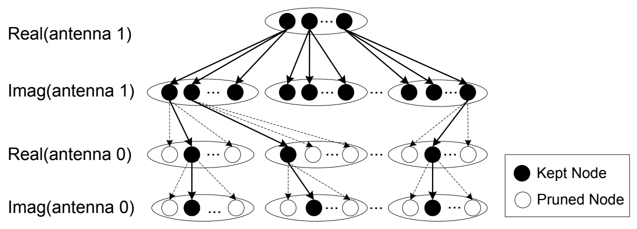


Figure 1: An example of the search process for a 2×2 16-QAM MIMO system

with small distances. The second step computes an LLR value for each transmitted bit using the candidate list.

3.2.1 Candidate Search

The search algorithm is essentially an SSFE MIMO detector [2, 5, 10] operating in the real domain. The search algorithm attempts to find candidate vectors with small distances in a greedy fashion. As $\mathbf{R}$ is an upper triangular matrix, the search algorithm evaluates transmitted symbols in reverse order, from antenna $N_t - 1$ to antenna 0. The procedure is equivalent to a tree traversal. For example, a complete tree search for a 2×2 MIMO system using 16-QAM is shown in Figure 1. This search keeps all branches of antenna 1 by fully expanding the first two levels of the tree. For the subsequent tree levels, the branches of the tree are pruned by keeping the best outgoing paths. All surviving paths at the end of the procedure are in our candidate list which in this example would be sixteen.

Let the k^{th} path be $\mathbf{p}^k = [p^k_{2N_t-1}, \dots, p^k_t]$, the set of nodes along the path from the root node to p^k_t, a node on level t. The best outgoing path can be found using Schnorr-Euchner enumeration [15]. The partial distance, the distance from p^k_t to the i^{th} node on level $t-1$, $w^{\langle t-1\rangle}_{k,i}$, can be computed as

$$w^{\langle t-1\rangle}_{k,i} = ||\hat{y}_{t-1} - \sum_{j=2N_t-1}^{t} r_{k,j}p^k_j - r_{t-1,t-1}s_i||^2, \qquad (6)$$

$$= ||b^k_{t-1} - r_{t-1,t-1}s_i||^2, \qquad (7)$$

where $r_{k,i}$ is the k^{th} row of the i^{th} column of $\mathbf{R}$ and $\hat{y}_t$ is the t^{th} row of $\hat{\mathbf{y}}$. To expand this path, the best node in level $t-1$ that minimizes $w^{\langle t-1\rangle}_{k,i}$ is simply the closest constellation point in $\mathbf{\Omega'}$ to $\gamma^j = (r_{t-1,t-1})^{-1}b^k_{t-1}$, the zero-forcing solution. If node i is the best node found at level $t-1$, the k^{th} distance can be updated by adding the partial distance, $w^{\langle t-1\rangle}_{k,i}$, to the distance from the previous level t as follows:

$$d_k = d_k + w^{\langle t-1\rangle}_{k,i}. \qquad (8)$$

Algorithm 2 Candidate search CUDA kernel: the k^{th} thread search for the k^{th} candidate

```
1. Input: $\mathbf{E} = [\mathbf{R}|\hat{\mathbf{y}}]$
2. Initialization:
   (a) $Q = \sqrt{M}$
   (b) $\mathbf{p}^k = [0, 0, \dots, 0, \Im(\Omega_k), \Re(\Omega_k)]$
3. $d_k = (\hat{y}_{2N_t-1} - r_{2N_t-1,2N_t-1} \cdot p^k_{2N_t-1})^2$,
4. $d_k = d_k + \left(\hat{y}_{2N_t-2} - \sum_{i=2N_t-2}^{2N_t-1} r_{2N_{t-2},i} \cdot p^k_i\right)^2$
5. for step $i = 2N_t - 3$ to $0$ do
6.     $b^k_i = \hat{y}_i$,
7.     for step $j = 2N_t - 1$ to $i + 1$
8.         $b^k_i = b^k_i - r_{i,j} \cdot p^k_j$
9.     end for
       // Find the best outgoing node
10.    $\gamma_k = b^k_i / r_{i,i}$
11.    $p^k_i = \text{round}\left(\frac{1}{2}(\gamma_k + Q - 1)\right) \cdot 2 - Q + 1$
12.    if $(|p^k_i| > Q - 1)$ $p^k_i = \text{sign}(p^k_i) \cdot (Q - 1)$
       // Update the distance of the $k^{th}$ path
13.    $d_k = d_k + (b^k_i - r_{i,i} \cdot p^k_i)^2$
14. end for
```

The search algorithm is implemented with one kernel. The corresponding CUDA kernel function is shown in Algorithm 2. At runtime, M threads execute the set of instructions defined in Algorithm 2 in parallel to perform the candidate search. Each thread is assigned to one modulation point, where the k^{th} thread is assigned to the k^{th} modulation point in the finite alphabet $\mathbf{\Omega}$. The first two levels of the tree are fully expanded as shown in Lines 3-4. For the subsequent levels, level $2N_t - 3$ to level 0, the algorithm first computes partial distances and then prunes outgoing branches by keeping the best outgoing paths. The partial distance for the k^{th} path is computed in Lines 6-9. Line 10 computes γ_k, which is used to find the best node at level i in lines 11-12. The best node is selected with a simple round function followed by a threshold function on γ_k. With the best node found, Line 13 updates the k^{th} distance by adding the partial distance of the best node. At the end of the loop, the path $\mathbf{p}^k$ is a candidate in our candidate list.

3.2.2 0-Hypothesis and 1-Hypothesis Generation

With the candidate list, the 0-hypothesis and 1-hypothesis for each transmitted bit can be computed. The computations for the k^{th} thread are summarized in Algorithm 3. As shown in Line 1, the k^{th} path, $\mathbf{p}^k$, is first demodulated into a binary vector $\mathbf{b}^k$ which is then stored in shared memory. The corresponding distance for the k^{th} path, d_k, is also stored in shared memory. We use $N_t \log(M)$ threads, one thread per bit, to compute the 0-hypothesis and the 1-hypothesis for each bit. In lines 5-11, the k^{th} thread scans the

Algorithm 3 LLR computation CUDA kernel: The k^{th} thread updates the k^{th} 0-hypothesis and the k^{th} 1-hypothesis

```
 1. $\mathbf{b}^k = \text{demod}(\mathbf{p}^k)$, $d_k = d$
 2. __syncthreads()
 3. if $(k < N_t \log_2(M))$
 4.     Initialization: $h_k^1 = \infty$ and $h_k^0 = \infty$.
 5.     for step $j = 0$ to $M - 1$ do
 6.         if ($k^{\text{th}}$ bit of $\mathbf{b}^j = 1$) and ($d_j \leq h_k^1$)
 7.             $h_k^1 = d_j$
 8.         else if ($k^{\text{th}}$ bit of $\mathbf{b}^j = 0$) and ($d_j \leq h_k^0$)
 9.             $h_k^0 = d_j$
10.         end if
11.     end for
12. end if
13. $L_D^k = h_k^0 - h_k^1$
```

binary vectors one by one to find the 0-hypothesis, h_k^0, and the 1-hypothesis, h_k^1, for the k^{th} bit. Finally, the LLR for the k^{th} bit is the difference between h_k^0 and h_k^1.

3.2.3 Optimizations

Beyond the above procedures, we also improve throughput in several ways.

1. We unroll the loops in both algorithms to reduce the total number of instructions.
2. As there are no data dependencies between the threads in the search process, it is possible to store the path history $\mathbf{P}$ in registers instead of using shared memory to eliminate shared memory read/load instructions and memory address computation. Storing path history in registers also reduces the number of device memory accesses as computation is carried out with on-chip resources.
3. On Nvidia GPUs, the SIMD instructions (or WARP instructions) are 1024 bit wide, where each instruction operates on 32 elements. At runtime, 32 threads share the same instruction. As a result, multiple MIMO detections are packed into one thread block to ensure that each thread block consists of an integer multiple of 32 threads to improve efficiency. For example, for a 4×4 MIMO 16-QAM system, QR decomposition uses 8 threads and MIMO detection uses 16 threads. We pack at least 4 QR decompositions into one thread block and at least two 16-QAM detectors into one thread block to ensure there are 32 threads within a thread block.

3.3 N-Way Parallel MIMO Detection

As described in the previous section, the soft-output SSFE detector consists of a single QR decomposition, a single candidate search and a single LLR generator. However, the error-rate performance of the soft-output SSFE detection is significantly worse than that of the soft-output max-log-MAP detection, as shown in Section 4.1.

We now describe a simple algorithm which improves upon the error-rate performance of the SSFE detector. We perform several tree searches with different antenna detection orders in parallel to improve error-rate performance of the detector. This improves performance as multiple tree searches generate a larger candidate list which results in more reliable LLRs. In our design, we run N parallel candidate searches, where $1 \leq N \leq N_t$, to generate N parallel candidate lists, each with M candidates. We then generate LLR values from the combined candidate list, which consists of MN candidates.

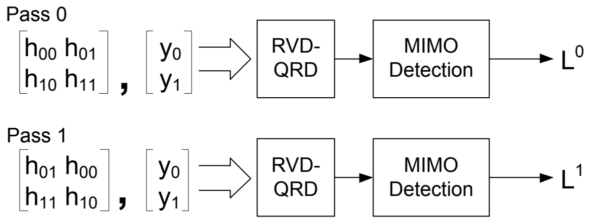

Pass 0
$\begin{bmatrix} h_{00} & h_{01} \\ h_{10} & h_{11} \end{bmatrix}, \begin{bmatrix} y_0 \\ y_1 \end{bmatrix}$
RVD-QRD
MIMO Detection
L^0
Pass 1
$\begin{bmatrix} h_{01} & h_{00} \\ h_{11} & h_{10} \end{bmatrix}, \begin{bmatrix} y_0 \\ y_1 \end{bmatrix}$
RVD-QRD
MIMO Detection
L^1

Figure 2: Proposed detector for a 2×2 MIMO System.

For example, the proposed algorithm for a 2×2 MIMO 16-QAM system is shown in Fig. 2. The example consists of two parallel detectors. The inputs consist of the received vector $\mathbf{y}$ and the channel matrix $\mathbf{H}$. A different antenna detection order can be obtained by a simple circular rotation of columns of $\mathbf{H}$. Each detector performs QR decomposition followed by candidate search to generate a candidate list. The results from the two detectors, two candidate lists, are then used to generate LLR values.

In our implementation, we spawn $2NN_t$ threads for an input pair $\mathbf{H}$ and $\mathbf{y}$. Each set of $2N_t$ threads constructs a permuted version of $\mathbf{H}$ in shared memory by reading the input data in a different order. Each set of $2N_t$ threads then performs QR decompositions on its permuted channel matrix. We then spawn MN threads to perform N parallel MIMO detections and LLR generation. To reduce communication among cores on the GPU, all threads corresponding to an instance of a MIMO detection problem reside within the same thread block.

3.3.1 LLR Computation

The threads generate LLRs using the larger combined candidate list, which consists of MN candidates. We can use one thread to scan through all MN candidates to find the 0-hypothesis, h_k^0 , and 1-hypothesis, h_k^1, for the k^{th} bit. In this case, we use $N_t \log_2(M)$ threads, one thread per transmitted bit, to compute LLRs for all transmitted bits.

Since our thread block consists of MN threads, we can improve the efficiency of the LLR generator by parallelizing the workload further. Instead of one thread per transmitted bit, we split the workload of finding the h_k^0 and h_k^1 among N threads. Since there are N lists where each list consists of M candidates, we assign one list to each thread. Using the assigned list, each thread attempts to find the 0-hypothesis and the 1-hypothesis. As a result, there is one set of N 0-hypotheses and one set of N 1-hypotheses per bit. To find the 0-hypothesis and the 1-hypothesis, we use two threads to scan through the N 0-hypotheses and N 1-hypotheses to find the minimum of each set. The two minimums correspond to the 0-hypothesis and 1-hypothesis respectively. The difference between these values is L_D^k, the LLR for the k^{th} bit. In total, we use $NN_t \log_2(M)$ threads to compute LLRs for all transmitted bits.

The complexity of the LLR generator is directly proportional to the number of candidates in the candidate list, which is MN. The parallel searches do not necessarily generate unique candidates. As a result, there may be duplicates in the candidate list. Since (4) consists of $\min(\cdot)$ operators, duplications do not affect the result. Although it is possible to reduce the complexity of the LLR generator by eliminating duplications in the candidate list, the number of unique candidates is not fixed, which leads to indeterminate runtime. As a result, we do not eliminate duplicate candidates in the list.

4 Performance

In this section, we first show the bit error rate (BER) performance simulation results of our proposed detector. We then investigate and analyze the detector's throughput performance on Fermi and Kepler graphics cards for various different configurations. We then compare and show the advantages of our detector to other GPU-based soft-output MIMO detector implementations.

4.1 BER Performance

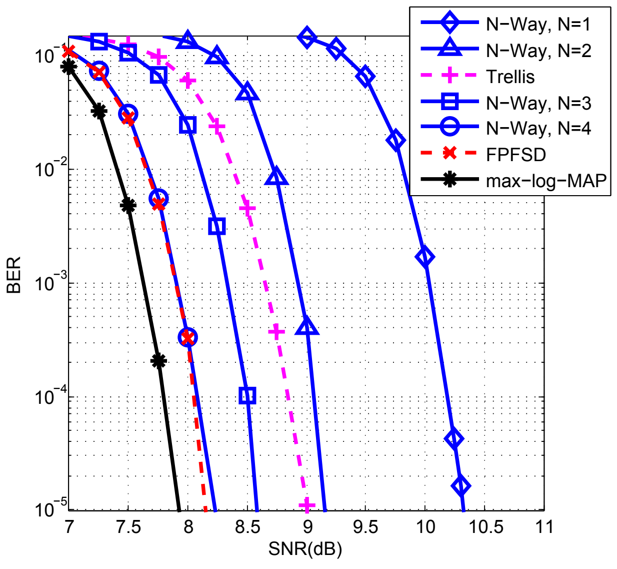
We compared the BER performance of the N-way parallel MIMO detector against several other soft-output detectors, including soft-output trellis-based MIMO detector [19], fully parallel fixed complexity-sphere detector (FPFSD) [14] and soft-output max-log-MAP detector. The soft-output max-log-MAP detector computes LLR values using the set of all possible transmit vectors (i.e. a direct implementation of (4)) and serves as the performance bound.

In our BER simulation, we first generate a random binary information vector which is then encoded by a rate 1/3 3GPP LTE turbo encoder where $K = 6144$. We then modulate the coded binary vector onto MIMO symbols. The symbols are transmitted through a Rayleigh fading channel with additive white Gaussian noise. The detector performs QR decomposition on the channel matrix and then performs soft-output detection once to generate LLRs. The soft-output of the detector is then fed to a 3GPP Turbo decoder [21] which performs up to 8 turbo decoding iterations. The detectors use an LLR clipping value of 8 for all the detector configurations with the exception of $N = 4$ where LLR clipping is not required. An iterative detection and decoding scheme is not considered in this paper as our implementation does not iteratively exchange LLRs between the MIMO detector and the channel decoder. We perform soft-output MIMO detection once followed by turbo decoding.

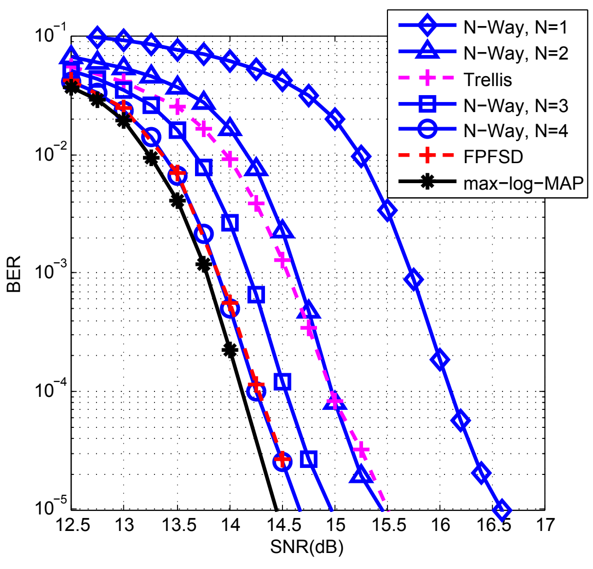
Figure 3 compares the BER performance of detectors for 16-QAM and 64-QAM. The trends are similar in both plots. The N-way parallel MIMO detector is equivalent to SSFE when $N = 1$. For $N = 1$, the BER performance of the N-way MIMO detector is worse than that of the other soft-output detectors. As we increase N, the performance of the detector improves as a larger candidate list increases the probability of finding the smallest 0-hypothesis and the smallest 1-hypothesis for each transmitted bit. For $N = 4$, the detector's performance is within 0.25 dB of the soft-output max-log-MAP detector. We note that the computational complexity difference between these two cases is significant—the number of leaf nodes visited is NM for the proposed algorithm compared to M^N for the soft-output max-log-MAP detector.

For $N \geq 3$, the N-way MIMO detector outperforms the soft-output trellis-based MIMO detector. The FPFSD MIMO detector is similar to the $N = 4$ case except that the FPFSD MIMO detector performs detection in the complex domain. In addition, the PFSD detector uses column-norm reordering preprocessing to improve performance. Nevertheless, we found the FPFSD detector performs similarly to the $N = 4$ N-way detector despite their differences.[3]

[3] The column-norm reordering processing, however, is an effective way of improving the $N = 1$ case. Nevertheless, the BER performance of the $N = 1$ case with column norm re-



(a) 16-QAM



(b) 64-QAM

Figure 3: BER Performance of soft-output 4×4 MIMO detectors in Rayleigh fading channels.

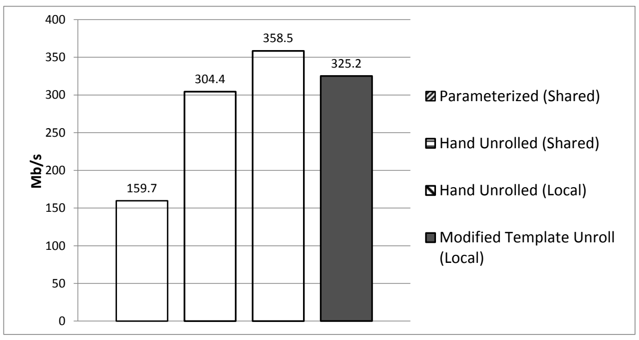


Figure 4: Effect of optimizations on 4×4, 64-QAM MIMO detectors, $N = 1$, 8192 subcarriers.

4.2 Throughput Performance

To measure the throughput performance of our implementation, we used two types of graphics cards. We first used an NVIDIA GeForce GTX 470 graphics card (Fermi) with 448 shaders running at 1215 MHz and with 1280 MB of GDDR5 with a peak bandwidth of 133.9 GB/s. In addition, we used one GPU on NVIDIA GeForce GTX 690 graphics card (Kepler), which has 1536 shaders running at 915 Mhz and has 2GB of GDDR5 with a peak bandwidth of 192.3 GB/s. This setup is equivalent to a GTX 680 graphic card downclocked by 10%. In our benchmark, the reported execution time is averaged over 1000 runs.

We first analyze the runtime of N-way detection without considering transport time, the time required to copy data from host memory to GPU and vice versa. We then look at the runtime of QR decomposition without considering transport time. Finally we report the runtime of the entire design considering transport time.

4.2.1 MIMO Detection Kernel

Effect of optimizations on kernel performance: We first present the effect of different optimization techniques on the performance of the detector. The nested loop within Algorithm 2 depends on N_t (the number of antennas), while the loop within Algorithm 3 depends on N_t as well as M (the modulation order). Unrolling these loops reduces the number of instructions which increases the throughput of the MIMO detector. Similarly, locality of the data also affects performance of the detector. We tried different unrolling techniques in conjunction with different data placement. As an illustrative example, we consider the peak throughput of a 4×4 64-QAM detector where $N = 1$. We present the results on the Fermi GPU as the trend is similar for Kepler. The BER performances of different cases are presented in Figure 4. We now describe and explain each case in detail:

1. For the initial case (labeled as "parameterized (shared)" in Figure 4), we put the path history in shared memory and attempt to use NVIDIA unroll directives to unroll these loops. Since both N_t and M are input parameters, the MIMO detector kernel is designed as a template function where N_t and M

ordering preprocessing is still worse than that of the $N = 2$ case without column-norm reordering.

are template parameters. This enables the compiler to generate different instances of the function for different combinations of N_t and M. For the nested loop within Algorithm 2, we notice that the compiler only unrolls the inner loop and does not unroll the outer loop even though unroll directives are used on both loop levels.
2. For the second case (labeled as "hand unrolled (shared)"), we unrolled loops manually by hand, which results in significant improvement.
3. For the third case (labeled as "hand unrolled (local)"), the array access pattern is now completely deterministic as the loops are completely unrolled. As a result, we put the path history into local memory instead of shared memory. Since memory access patterns are deterministic at compile time, the compiler stores the path vectors into registers (instead of device memory), which eliminates shared memory load/store instructions and increases throughput.
4. For the fourth case (labeled as "modified template unroll (local)"), we note that manually unrolling is not practical as we need to manually generate N_tM instances of the MIMO detector functions, one instance per MIMO configuration. We use C++ templates to perform compile-time transformations to force the compiler to unroll these loops. Although the number of instructions is more than manual unrolling, it is significantly faster than our initial cases and does not require multiple copies of the kernel code. Therefore, we use this configuration in the subsequent performance results.

MIMO detection kernel performance: Table 1 shows the throughput of the N-way detector for different MIMO configurations in a system with 8192 subcarriers. We packed up to 8 different detection problems per thread block for 16-QAM MIMO configurations. Since the number of threads required for the MIMO detector scales linearly with N, runtime of the N-way detector is directly proportional to N when M and N_t are fixed values. In the case where N and M are fixed values, the computation required to generate LLR values, as show in Algorithm 3, is a constant overhead that depends on M but does not depend on N_t. As a result, runtime is not directly proportional to N_t. Consequently, runtime of the 2×2 64-QAM MIMO detector is not half of the 4×4 64-QAM MIMO detector. The detector achieved higher throughput on Kepler for the majority of the cases. For the computationally intensive cases such as the 4×4 64-QAM configuration where $N = 4$, Kepler achieved a speedup of 1.7× over Fermi.

We used a large number of subcarriers to report peak throughput. However, we do not need an extremely large number of subcarriers to achieve throughput close to the peak performance. Figure 5 shows throughput as a function of the number of subcarriers. We see that in all of these cases, the throughput starts to plateau after 2048 subcarriers.

4.2.2 QR Decomposition Kernel

An N-way Parallel MIMO Detection requires N QR decompositions on an input $\mathbf{y}$ and $\mathbf{H}$ to generate inputs for the detector. Table 2 shows the results for 2×2 and 4×4 MIMO configurations with different values of N. We packed up to 8 different QR decomposition problems in one thread block. The results show that the QR decomposition kernel is not the bottleneck as the runtime is much smaller than that of MIMO detection. Nevertheless, we also used C++ templates to fully unroll the loops in Algorithm 1 to reduce runtime.

4.2.3 Performance of the Complete Design

Table 3 shows the total runtime of the complete design measured with the CPU timer. The behavior of the detector is similar to the runtime of the detector. For example, in the case where N_t and M are fixed values, runtime increases as N increases. However, due to constant overhead such as transport time, runtime is no longer linearly proportional to N. We note that the total runtime results are pessimistic. Using CUDA streams, data transfers can be overlapped with computation to further increase the throughput.

4.2.4 Comparison with Existing Work

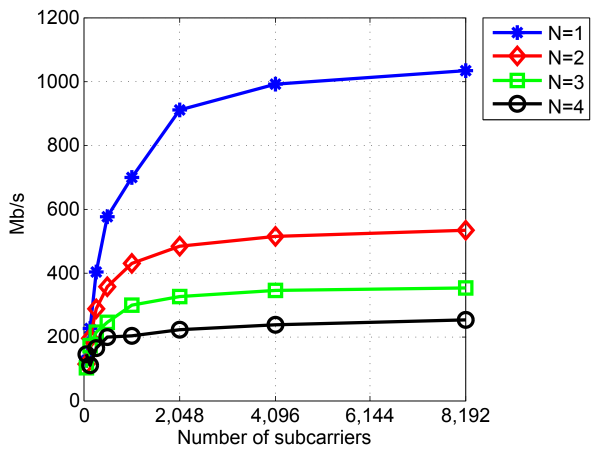
We compare our kernel time of the MIMO detector with the kernel time of other soft-output MIMO detection implementations.

We compared the $N = 4$ case against the soft-output trellis-based MIMO detector [19] and the fully parallel fixed complexity-sphere detector (FPFSD) [14]. As shown in Section 4.1, the error-rate performance of the $N = 4$ case is better than the BER performance of soft-output trellis MIMO detector and is similar to the BER performance of FPFSD. To compare these detectors, we also provided throughput normalized by core count and core frequency.

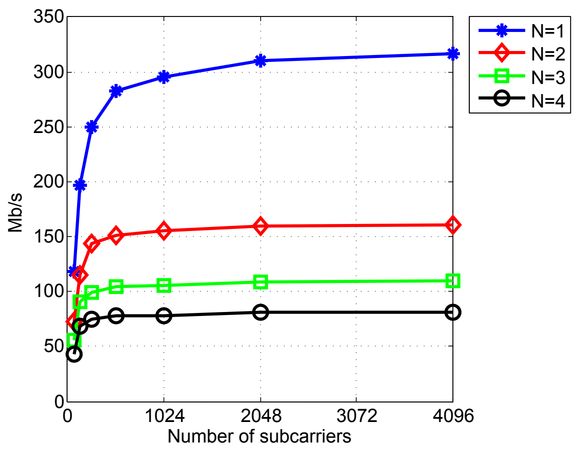
Compared to N-way MIMO detection, trellis based MIMO detection [19] requires more instructions. For trellis based MIMO detection, the number of instructions required to find the best path out of M possible paths scales with modulation order M. By comparison, N-way detection uses a round and threshold function to find the best outgoing path. As a result, the number of instructions required to find the best outgoing path

Table 1: MIMO Detection kernel time for 8192 MIMO symbols

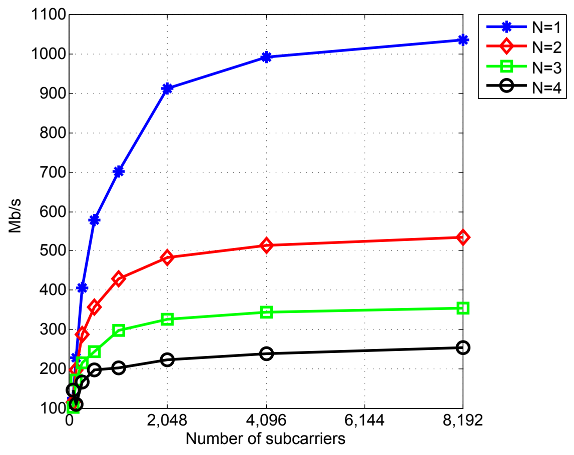
GPU	Configuration	$N = 1$	$N = 2$	$N = 3$	$N = 4$
Fermi	2×2, 16-QAM	0.069 ms/946.8 Mb/s	0.127 ms/511.5 Mb/s	-	-
	4×4, 16-QAM	0.117 ms/1122.7 Mb/s	0.323 ms/406.0 Mb/s	0.433 ms/302.2 Mb/s	0.547 ms/239.7 Mb/s
	2×2, 64-QAM	0.480 ms/204.2 Mb/s	0.953 ms/102.8 Mb/s	-	-
	4×4, 64-QAM	0.602 ms/325.80 Mb/s	1.190 ms/164.7 Mb/s	1.75 ms/111.7 Mb/s	2.560 ms/76.5 Mb/s
Kepler	2×2, 16-QAM	0.049 ms/1315.1 Mb/s	0.093 ms/701.1 Mb/s	-	-
	4×4, 16-QAM	0.126 ms/1036.8 Mb/s	0.245 ms/533.7 Mb/s	0.372 ms/351.7 Mb/s	0.515 ms/254.2 Mb/s
	2×2, 64-QAM	0.231 ms/423.7 Mb/s	0.477 ms/205.5 Mb/s	-	-
	4×4, 64-QAM	0.378 ms/519.2 Mb/s	0.713 ms/274.9 Mb/s	1.203 ms/162.92 Mb/s	1.467 ms/133.61 Mb/s



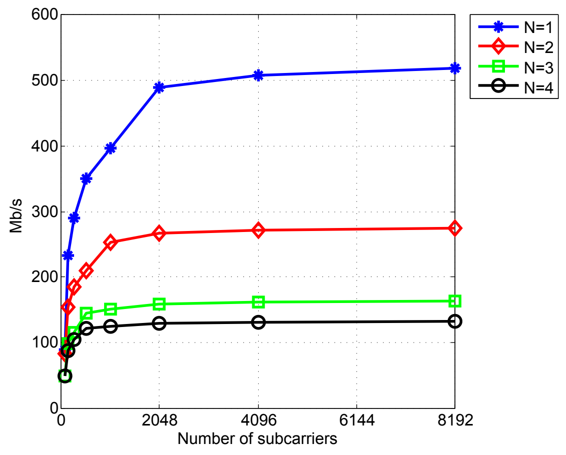
(a) 16-QAM, Fermi



(b) 64-QAM, Fermi



(c) 16-QAM, Kepler



(d) 64-QAM, Kepler

Figure 5: Throughput of 4×4 MIMO detectors vs. workload size.

is constant and does not depend on M. Consequently, particularly for higher modulation orders, the N-way MIMO detector achieves higher normalized throughput than that of the trellis-based MIMO detector.

The work that is the most similar to ours is the FPFSD in [14], which uses a similar detector on the same GPU. We emphasize, however, that the throughput of FPFSD is lower than that of our work. In our design, we perform detection in the real domain through RVD, while FPFSD performs MIMO detection in the complex domain. We do not expect this to cause a large throughput difference, as RVD does not reduce the computation complexity. We believe there are two key differences between our detector and the FPFSD. First, in our design, we store the candidate list in registers to reduce the number of device memory accesses. For

Table 2: QR decomposition kernel time for 8192 MIMO symbols.

GPU	Configuration	$N = 1$	$N = 2$	$N = 3$	$N = 4$
Fermi	2×2	0.037 ms	0.057 ms	–	–
	4×4	0.168 ms	0.323 ms	0.433 ms	0.547 ms
Kepler	2×2	0.035 ms	0.053 ms	–	–
	4×4	0.178 ms	0.326 ms	0.483 ms	0.596 ms

Table 3: Total runtime for 8192 MIMO symbols including data transport time.

GPU	Configuration	$N = 1$	$N = 2$	$N = 3$	$N = 4$
Fermi	2×2, 16-QAM	0.340 ms/191.1 Mb/s	0.440 ms/147.7 Mb/s	–	–
	4×4, 16-QAM	0.890 ms/147.2 Mb/s	1.130 ms/115.9 Mb/s	1.360 ms/ 96.3 Mb/s	1.640 ms/79.9 Mb/s
	2×2, 64-QAM	0.820 ms/119.5 Mb/s	1.350 ms/72.6 Mb/s	–	–
	4×4, 64-QAM	1.560 ms/125.6 Mb/s	2.180 ms/89.9 Mb/s	2.950 ms/66.4 Mb/s	3.870 ms/0.6 Mb/s
Kepler	2×2, 16-QAM	0.290 ms/224.3 Mb/s	0.220 ms/145.3 Mb/s	–	–
	4×4, 16-QAM	0.698 ms/188.0 Mb/s	0.925 ms/141.6 Mb/s	1.209 ms/108.3 Mb/s	1.460 ms/89.7 Mb/s
	2×2, 64-QAM	0.489 ms/200.5 Mb/s	0.779 ms/125.8 Mb/s	–	–
	4×4, 64-QAM	0.986 ms/198.9 Mb/s	1.432 ms/136.9 Mb/s	2.068 ms/ 94.8 Mb/s	2.460 ms/79.6 Mb/s

Table 4: Throughput comparison of the MIMO detection kernel with other GPU MIMO detectors.

Detector Type	GPU	Configuration	Throughput [Mb/s]	Normalized Throughput [(Mb/s)/(Core×GHz)]
Trellis-based [19]	Telsa C1060	4×4, 16-QAM	122.03	0.422
		4×4, 64-QAM	12.05	0.042
FPFSD [14]	Tesla C2070	4×4, 16-QAM	92.39	0.179
		4×4, 64-QAM	17.20	0.033
N-way, N = 4	GTX 470	4×4, 16-QAM	239.7	0.440
		4×4, 64-QAM	76.50	0.141

FPFSD, the candidate list is stored in device memory which increases the number of slow device memory accesses. Second, we reduce the number of instructions by aggressively unrolling loops in our kernels. This is not straightforward; we used C++ templates to automatically unroll loops for different MIMO detector configurations (see Section 4.2.1). These optimizations were not done for the FPFSD detector. As a result, we achieved higher normalized throughput than that of the FPFSD detector.

5 Conclusion

In this paper, we have presented a novel parallel high throughput MIMO detector algorithm that maps well onto Nvidia GPU architectures. We have implemented the proposed N-way MIMO detector on both the NVIDIA Fermi and Kepler GPUs, and have shown that our proposed detector outperforms existing GPU-based MIMO detectors in terms of detection throughput. Furthermore, we have shown that by changing the number of parallel candidate searches, our MIMO detector provides a wide range of detection options: from a design that provides excellent error-rate performance (within 0.25 dB of the soft-output max-log-MAP detector) to an extreme high-throughput design that achieves several hundred Mb/s to Gb/s detection throughput.

Acknowledgements This work was supported in part by Renesas Mobile, Texas Instruments, Xilinx, Samsung, Huawei, and by the US National Science Foundation under grants CNS-1265332, ECCS-1232274, EECS-0925942 and CNS-0923479.

References

1. Amiri K, Cavallaro JR, Dick C, Rao RM (2011) A high throughput configurable SDR detector for multi-user MIMO wireless systems. Journal of Signal Processing Systems 62:233–245
2. Barbero LG, Thompson JS (2006) A fixed-complexity MIMO detector based on the complex sphere decoder. In: IEEE International Workshop on Signal Processing Advances in Wireless Communications (SPAWC), IEEE

3. Burg A, Borgmann M, Wenk M, Zellweger M, Fichtner W, Bölcskei H (2005) VLSI implementation of MIMO detection using the sphere decoding algorithm. IEEE Journal of Solid-State Circuits 40:1566–1577
4. Burg A, Haene S, Perels D, Luethi P, Felber N, Fichtner W (2006) Algorithm and VLSI architecture for linear MMSE detection in MIMO-OFDM systems. In: IEEE International Symposium on Circuits and Systems (ISCAS), IEEE, pp 4012–4105
5. Hess C, Wenk M, Burg A, Luethi P, Studer C, Felber N, Fichtner W (2007) Reduced-complexity MIMO detector with close-to ML error rate performance. In: Proc. 17th ACM Great Lakes Symposium on VLSI, pp 200–203
6. Hochwald B, ten Brink S (2003) Achieving near-capacity on a multiple-antenna channel. IEEE Transactions on Communications 51:389–399
7. Janhunen J, Silven O, Juntti M, Myllyla M (2008) Software defined radio implementation of K-best list sphere detector algorithm. In: International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), pp 100–107
8. Karypis G, Kumar V (1994) Unstructured tree search on SIMD parallel computers. IEEE Transactions on Parallel and Distributed Systems 5(10):1057–1072
9. Kerr A, Campbell D, Richards M (2009) QR decomposition on GPUs. In: Proceedings of 2nd Workshop on GPGPU, ACM, pp 71–78
10. Li M, Bougard B, Lopez E, Bourdoux A, Novo D, Van Der Perre L, Catthoor F (2008) Selective spanning with fast enumeration: A near maximum-likelihood MIMO detector designed for parallel programmable baseband architectures. In: IEEE International Conference on Communications (ICC), IEEE, pp 737–741
11. Michalke C, Zimmermann E, Fettweis G (2006) Linear MIMO receivers vs. tree search detection: A performance comparison overview. In: IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC), IEEE
12. NVIDIA Corporation (2008) CUDA Compute Unified Device Architecture Programming Guide. URL http://www.nvidia.com/object /cuda_develop.html
13. Qi Q, Chakrabarti C (2010) Parallel high throughput soft-output sphere decoder. In: IEEE Workshop on Signal Processing Systems (SiPS), IEEE, pp 50 – 55
14. Roger S, Ramiro C, Gonzalez A, Almenar V, Vidal A (2012) Fully parallel GPU implementation of a fixed-complexity soft-output MIMO detector. IEEE Transactions on Vehicular Technology 61:3796 – 3800
15. Schnorr C, Euchner M (1993) Lattice basis reduction: Improved practical algorithms and solving subset sum problems. In: Mathematical Programming, vol 66, pp 181–191
16. Studer C, Burg A, Bolcskei H (2005) Soft-output sphere decoding: Algorithms and VLSI implementation. IEEE Journal on Selected Areas in Communications 26(2):290–300
17. Trefethen L, Bau D (1997) Numerical Linear Algebra. SIAM: Society for Industrial and Applied Mathematics
18. Wong K, Tsui C, Cheng R, Mow W (2002) A VLSI architecture of a K-best lattice decoding algorithm for MIMO channels. In: IEEE International Symposium on Circuits and Systems (ISCAS), vol 3, pp 273–276
19. Wu M, Sun Y, Gupta S, Cavallaro JR (2010) Implementation of a high throughput soft MIMO detector on GPU. Journal of Signal Processing Systems pp 123–136
20. Wu M, Dick C, Sun Y, Cavallaro J (2011) Improving MIMO sphere detection through antenna detection order scheduling. In: Software Defined Radio Forum (SDR-WInnComm), pp 280–284
21. Wu M, Sun Y, Wang G, Cavallaro J (2011) Implementation of a high throughput 3GPP turbo decoder on GPU. Journal of Signal Processing Systems pp 171–183
22. Wu M, Yin B, Cavallaro JR (2012) Flexible N-way MIMO detector on GPU. In: IEEE Workshop on Signal Processing Systems (SiPS), IEEE, pp 318–323
23. Wubben D, Bohnke R, Kuhn V, Kammeyer KD (2004) Near-maximum-likelihood detection of MIMO systems using MMSE-based lattice reduction. In: IEEE International Conference on Communications, IEEE, vol 2, pp 798–802