J Sign Process Syst (2011) 64:123–136
DOI 10.1007/s11265-010-0523-4

Implementation of a High Throughput Soft MIMO Detector on GPU

Michael Wu · Yang Sun · Siddharth Gupta · Joseph R. Cavallaro

Received: 1 February 2010 / Revised: 13 August 2010 / Accepted: 18 August 2010 / Published online: 4 September 2010

Abstract Multiple-input multiple-output (MIMO) significantly increases the throughput of a communication system by employing multiple antennas at the transmitter and the receiver. To extract maximum performance from a MIMO system, a computationally intensive search based detector is needed. To meet the challenge of MIMO detection, typical suboptimal MIMO detectors are ASIC or FPGA designs. We aim to show that a MIMO detector on Graphic processor unit (GPU), a low-cost parallel programmable co-processor, can achieve high throughput and can serve as an alternative to ASIC/FPGA designs. However, careful architecture aware software design is needed to leverage the performance offered by GPU. We propose a novel soft MIMO detection algorithm, multi-pass trellis traversal (MTT), and show that we can achieve ASIC/FPGA-like performance and handle different configurations in software on GPU. The proposed design can be used to accelerate wireless physical layer simulations and to offload MIMO detection processing in wireless testbed platforms.

Keywords GPU · Soft output detection · MIMO · Wireless baseband architecture

M. Wu (✉) · Y. Sun · S. Gupta · J. R. Cavallaro
Electrical and Computer Engineering, Rice University, Houston, TX 77005, USA
e-mail: mbw2@rice.edu

Y. Sun
e-mail: ysun@rice.edu

S. Gupta
e-mail: sgupta@rice.edu

J. R. Cavallaro
e-mail: cavallar@rice.edu

1 Introduction

Wireless communication systems enable higher data rate services by providing higher spectral efficiency. Wireless physical layers for high data rate wireless standards such as WiMAX and 3GPP LTE downlink often use multiple-input multiple-output combined with orthogonal frequency division multiplexing (MIMO-OFDM). MIMO increases spectral efficiency by employing multiple antennas at the transmitter and at the receiver. OFDM divides the available bandwidth into a set of orthogonal subchannels or subcarriers. To combat errors due to channel noise and fading, a channel decoder such as low density parity code (LDPC) is combined with a soft output MIMO detector at the receiver to maximize performance gain. As the signal received at each antenna for each subcarrier consists of a combination of multiple data streams from multiple transmit antennas, a higher complexity detector is required to recover the transmitted vector compared to single antenna systems. Although an exhaustive search-based MIMO detector would be optimal, complexity would be prohibitive. Fortunately, a suboptimal MIMO detector can provide close to optimal performance with significantly lower complexity.

There are two main approaches to MIMO detection, depth-first tree-search algorithms such as sphere detection algorithm and breadth-first tree-search algorithms such as K-Best. The soft sphere detector suffers from non-deterministic complexity and variable throughput, and the signal-to-noise ratio (SNR) must be accurately estimated so that an initial radius can be determined. The inherent sequential nature of the depth-first search process significantly limits the throughput of the detector, especially when SNR is low. The K-Best algorithm is a fixed-complexity algorithm with fixed throughput. But this algorithm has a high sorting complexity which significantly limits the throughput of the detector, especially when K is large.

To meet the challenge of MIMO detection, typical suboptimal MIMO detectors are ASIC designs. For example, Burg et al. [3] implemented a depth-first 4×4 16-QAM detector in ASIC with an average throughput of 73 Mbps at an SNR of 20 dB. Wong et al. [19] first introduced a K-best (with $K = 10$) 4×4 16-QAM detector in ASIC achieving 10 Mbps. Later on Guo and Nilsson [9] developed a K-best Schnorr–Euchner (KSE) 4×4 16-QAM detector (with $K = 5$) in ASIC with a higher throughput of 53.3 Mbps. FPGA is another popular platform that can meet high data rate requirements. Huang et al. [11] prototyped a 4×4 16-QAM detector on a Xilinx FPGA device with a throughput of 81.5 Mbps and 36.1 Mbps based on Schnorr–Euchner (SE) and Viterbo-Boutros (VB) algorithm respectively. Software based designs are another viable alternative but typically do not meet performance requirements. Qi and Chakrabarti [15] mapped a depth-first detector on a multi-core software radio architecture (SDR), but did not include the throughput of the implementation. Antikainen et al. [2] presented a software defined implementation of a 4×4 16-QAM K-best detector (with $K = 7$) on a parallel transport triggered architecture (TTA) processor with a throughput of 5.3 Mbps. Janhunen et al. [12] implemented a 2×2 64-QAM K-best detector (with $K = 16$) on a programmable Sandbridge SB3500 processor which achieved a throughput of 32.0 Mbps, but requires an additional hardware sorter not in the original processor.

Programmable graphics processing units (GPU) deliver extremely high computation throughput by employing many cores working on a large set of data in parallel. As GPUs become increasingly more flexible, they can accelerate other tasks beyond the realm of graphics. For example, researchers have found that GPUs can perform very fast LDPC decoding [6]. Although the power consumption of GPU is higher than ASICs and FPGAs, GPU as an alternative to the traditional ASIC and FPGA solutions for wireless applications, especically in the realm of simulation and software defined wireless test-beds, remains attractive for several reasons. Since Communication algorithms typically are very parallel and can take advantage of the inherently parallel structure of GPUs, they can create a platform that is capable very high throughput. Unlike ASICs, GPUs can be reconfigured dynamically to handle different workloads. These types of processors are extremely cost-effective and ubiquitous in mobile and desktop devices. Combined with a short learning curve, GPUs reduce barriers to entry for academic and industrial research groups and enable very fast physical layer simulations as well as replacing custom ASICs or FPGAs on wireless testbed platforms such as WARPLab [1].

However, the underlying hardware of a GPU is fixed. Careful architecture-aware algorithm design and mapping are required to achieve good performance. Much of the mapping and optimization are left to the programmer. For example, the programmer needs to specify how to use the limited resources on the GPU, such as on-chip shared memory. Furthermore, the programmer needs to specify how computation is partitioned on GPU by partitioning threads among the cores to handle the workload. Designing a detection algorithm that scales well, while keeping the cores fully utilized to achieve peak throughput across different combinations of number of antennas and different modulations, is a difficult task. To the best of our knowledge, there are no existing implementations of a soft MIMO detector on a GPU besides our recent work [20, 21]. In this paper, we propose a MIMO soft detection algorithm, multi-pass trellis traversal (MTT), which is well suited for this architecture. We show that this MIMO detector implementation can achieve good performance while maintaining flexibility offered by programmable hardware. We also compare the performance of MTT against K-best and our previous GPU implementation [21]. We measure the efficiency of our implementation by measuring how well the hardware executes our code as well as the quality of the compiled code.

We give an overview of the CUDA architecture in Section 2 followed by an overview of MIMO detection in Section 3. Section 4 gives a description of the proposed detection algorithm. In Section 5, we present the major software blocks of our implementation. We present the performance results and provide an analysis of the performance results in Section 6. Finally, we conclude our investigation in Section 7.

2 Compute Unified Device Architecture (CUDA)

Compute Unified Device Architecture [13] is an NVIDIA GPU software programming model that allows the programmer to take advantage of the massive computation ability of a NVIDIA GPU. The programming model, which is shown in Fig. 1, is explicitly parallel. A programmer defines a kernel which specifies the series of computation steps on a data set. At runtime, the kernel spawns a large number of thread blocks, each

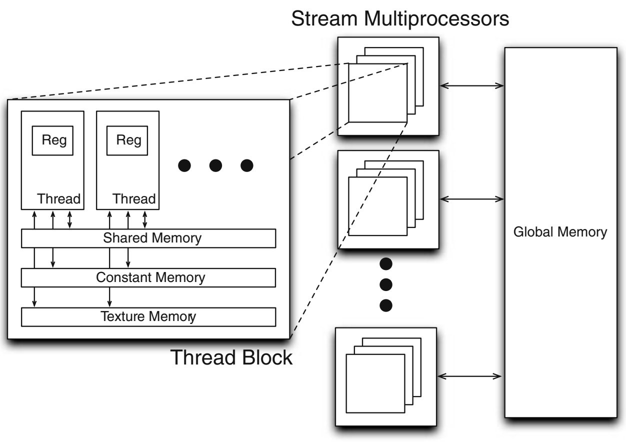


Figure 1 CUDA thread.

of which consists of up to 512 threads. Each thread can select a set of data using its own unique ID and executes the set of operations defined by the kernel on the data set. Threads within a block execute independently in this model but can synchronize through a low overhead barrier and can share data through shared memory. By contrast thread blocks are completely independent and can be synchronized by terminating the kernel and writing to global memory, which is expensive.

The massive computation ability of a NVIDIA GPU is due to the presence of multiple stream multiprocessors (SM). The overall architecture of an SM is eight ALU-wide single instruction multiple data (SIMD). At runtime, the architecture spawns multiple threads as defined by the kernel across multiple processors. Specifically, each thread block is mapped onto an SM. During execution, the kernel divides the threads into groups of 32. As the overall architecture is single instruction multiple data, if all 32 threads perform the same instruction, they share the same *warp* instruction which is executed over four cycles. Otherwise, threads execute serially. However, stalls can occur which lead to low core utilization. For example, global memory accesses lead to data stalls and register to register dependencies cause pipeline stalls.

To keep core utilization high, a SM can switch and issue an independent *warp* instruction from the same block or another thread block with zero-overhead. By executing multiple thread blocks on a SM concurrently, stalls can be kept to a minimum as the SM has multiple independent *warps* to choose from. Beside zero-overhead thread switching, local resources such as registers, shared memory and constant memory are local and provide faster memory access times. The characteristics of each memory type are shown in Table 1. On-chip memory is especially useful since reducing memory access time is crucial for efficient implementation. For example, shared memory, which can be as fast as a register, can reduce memory access time by keeping data on-chip and reducing redundant calculations by allowing data sharing among independent threads. However, shared memory on each SM is banked 16 ways. If 16 threads, half of a *warp*, are scheduled to access shared memory at the same time, they must meet certain conditions to allow the instruction to execute in one cycle. It takes one cycle if all threads within half of a *warp* access the same memory location (broadcast) or if none of them accesses the same bank. However, random layout with some broadcast and some one-to-one accesses will be serialized and cause a stall.

Table 1 Available resources for each memory.

Type	Speed	Access	Size
Register	Fast	RW	8,192 per SM
Shared memory	Fast	RW	16 KB per SM
Constant memory	Fast	RO	8 KB per SM
Texture memory	Fast	RO	8 KB per SM
Global memory	Slow	RW	>512 MB per SM

There are several other limitations to shared memory. First, only threads within a block can share data among themselves and threads between blocks can not share data through shared memory. Second, there are only 16 KB of shared memory on each stream multiprocessor and shared memory is divided among the thread blocks on a SM. Using too much shared memory can reduce the number of concurrent thread blocks mapped onto a SM. As a result, to keep the multiprocessor from idling, designing an algorithm that effectively partitions shared memory, has an efficient memory access pattern, and does not require synchronization between blocks and needs few global memory accesses is a non-trivial task.

3 System Model

For an $M \times N$ MIMO configuration, the transmitter sends different signals on M antennas and the receiver receives N different signals, one per receiver antenna. An $M \times N$ MIMO system can be modeled as:

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

where $\mathbf{y} = [y_0, y_1, ..., y_{M-1}]^T$ is the received vector. $\mathbf{H}$ is the $M \times N$ channel matrix, where each element, $h_{i,j}$, is an independent zero mean circularly symmetric complex Gaussian random variable with unit variance. Noise at the receiver is $\mathbf{w} = [w_0, w_1, ...w_{N-1}]^T$, where w_i is an independent zero mean circularly symmetric complex Gaussian random variable with σ^2 variance per dimension. The transmit vector is $\mathbf{s} = [s_0, s_1, ..., s_{M-1}]$, where s_i is drawn from a finite complex constellation alphabet, $\mathbf{\Omega}$, of cardinality Q. For example, the constellation alphabet for QPSK is $\{-1-j, -1+j, 1-j, 1+j\}$ and $Q = 4$ for this particular case.

After complex QR decomposition of the channel matrix, $\mathbf{H}$, we can model the $M \times N$ MIMO system with an equivalent model:

$$\mathbf{y} = \mathbf{QRs} + \mathbf{w} \tag{2}$$

$$\hat{\mathbf{y}} = \mathbf{Rs} + \hat{\mathbf{w}} \tag{3}$$

where $\mathbf{R}$ is an $M \times N$ complex upper triangular matrix. The vector $\hat{\mathbf{y}} = [\hat{y}_0, \hat{y}_1, ..., \hat{y}_{N-1}]$ is the effective complex receive vector.

Each symbol s_m is obtained using the mapping function $\mathbf{s} = \text{map}(\mathbf{x})$, where $\mathbf{x} = \{x_0, x_1, ..., x_{M_c-1}\}$, an $M_c \times 1$ vector (block) of transmitted binary bits. $M_c = \log_2 Q$ is the number of bits per constellation symbol.

The soft MIMO detector calculates the *a posteriori* probability (APP) in terms of log likelihood ratio (LLR) for each transmitted bit, x_k. Assuming no extrinsic probability, using the max-log approximation, the LLR can be expressed as [10]:

$$L(x_k|\hat{\mathbf{y}}) \approx \frac{1}{2\sigma^2}\left(\min_{\mathbf{x}\in\mathbb{X}_{k,-1}} \Lambda(\mathbf{s},\mathbf{y}) - \min_{\mathbf{x}\in\mathbb{X}_{k,+1}} \Lambda(\mathbf{s},\mathbf{y})\right), \tag{4}$$

where the set $\mathbb{X}_{k,+1} = \{\mathbf{x}|x_k = +1\}$ and set $\mathbb{X}_{k,-1} = \{\mathbf{x}|x_k = -1\}$ and

$$\Lambda(\mathbf{s}, \hat{\mathbf{y}}) = \|\hat{\mathbf{y}} - \mathbf{Rs}\|_2^2 \tag{5}$$

4 Proposed MIMO Detection

One way we can solve the detection problem is through exhaustive search. However, searching through all possible transmit vectors is a time intensive process. For a 4×4 MIMO point to point link, if the transmitter utilizes 16 QAM, the total number of possible transmit vectors is $16^4 = 66{,}534$. If the transmitter utilizes 64 QAM, the total number of possible transmit vectors is $64^4 = 16{,}777{,}216$. To reduce complexity, there are two classes of search-based MIMO detection algorithms to find the candidate lists, the set $\mathbb{X}_{k,+1}$ and $\mathbb{X}_{k,-1}$, for the soft decision MIMO detector, K-best MIMO detection [7] and depth-first sphere detection [19]. In both cases, the algorithms view the search space, the set of all possible transmit vectors, as a tree. Sphere-detection is a depth-first tree search algorithm. In this case, we traverse the tree depth first. Each time we reach the last level of the tree of a transmit vector, we use the Euclidean distance to prune all nodes with partial distance bigger than the current Euclidean distance. The drawback of this algorithm is that it is essentially sequential, we search for candidates depth first one at a time. The runtime is not deterministic and in the worse case the detector needs to traverse through the entire tree. K-best detection algorithm is a breadth-first tree search algorithm, a greedy MIMO detection algorithm. It reduces the number of candidates we search through in the detection process by detecting input symbols antenna by antenna, keeping at most K candidates per level. There are a few drawbacks to this algorithm. First, to find the K-best candidates per level, the algorithm requires expansion and sorting of KM candidates. Authors in [12] explored an implementation of the K-best MIMO detector in software and proposed a hybrid scheme, where a software processor is combined with a hardware sorter to meet the performance requirements. Furthermore, sorting across a tree level requires storing KM candidates, which can outstrip the amount of shared memory on a GPU. As such, we look for an alternative search algorithm, a sort-free algorithm that is very data parallel and efficiently uses shared memory.

Without loss of generality, we use a simple 3×3 QPSK system to explain our proposed algorithm in this section. In Sections 5 and 6, we will use common 2×2 and 4×4 configurations with various modulation orders up to 64 QAM.

4.1 MIMO Trellis

To generate LLR values for each transmitted bit x_k based on (4), the soft MIMO detector needs to compute the minimum Euclidean distance

$$\Lambda = \left\| \begin{bmatrix} \hat{y}_0 \\ \hat{y}_1 \\ \hat{y}_2 \end{bmatrix} - \begin{bmatrix} R_{00} & R_{01} & R_{02} \\ 0 & R_{11} & R_{12} \\ 0 & 0 & R_{22} \end{bmatrix} \begin{bmatrix} s_0 \\ s_1 \\ s_2 \end{bmatrix} \right\|^2, \tag{6}$$

over sets $\mathbb{X}_{k,+1}$ and $\mathbb{X}_{k,-1}$. The calculation of Λ can be decomposed as: $\Lambda = w^{<0>} + w^{<1>} + w^{<2>}$, where $w^{<t>}$ is the 1-D Euclidean distance and is calculated as

$$\begin{aligned} w^{<0>} &= \|\hat{y}_2 - R_{22}s_2\|^2, \\ w^{<1>} &= \|\hat{y}_1 - (R_{11}s_1 + R_{12}s_2)\|^2, \\ w^{<2>} &= \|\hat{y}_0 - (R_{00}s_0 + R_{01}s_1 + R_{02}s_2)\|^2. \end{aligned} \tag{7}$$

This process can be illustrated using a MIMO flow graph as shown in Fig. 2. There are 3 trellis stages, one stage per antenna. In each stage, there are Q vertices, one per constellation point. The edge between $v(t-1, i)$ and $v(t, j)$ has a weight of $w_{i,j}^{<t>}$. The weight function depends on its current stage and all its

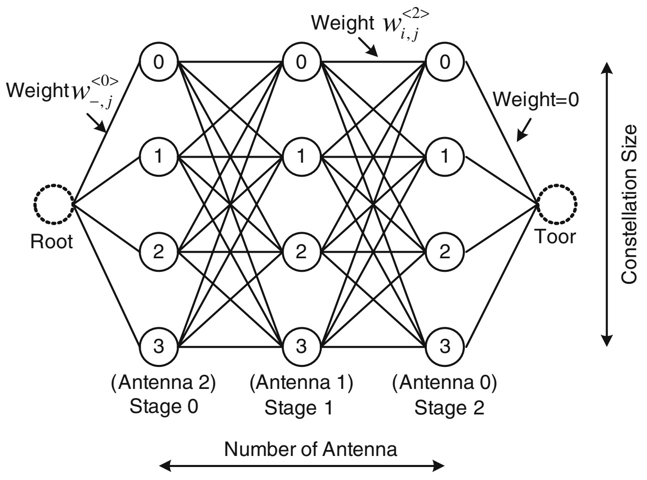


Figure 2 MIMO detection flow graph.

predecessors. For example, $w_{i,j}^{<2>}$ depends on the vertices in stages 2, 1, and 0.

4.2 Soft MIMO Detection

To compute the LLR value for each transmitted bit x_k, we first generate a candidate list for each trellis stage. For each vertex i $(0 \le i \le Q-1)$ in the stage t $(0 \le t \le M-1)$, the detector finds the shortest path, which must contain this vertex, from the root to the toor. The Q conditioning shortest paths found at every stage t make a candidate list $\mathcal{L}_t$.

We then use the lists to compute the LLR for each bit in a straight forward manner

$$L(x_i^{<t>}|\mathbf{y}) = \frac{1}{2\sigma^2}\left(\min_{\mathbf{x}\in\mathcal{L}_{t,-1}} \Lambda - \min_{\mathbf{x}\in\mathcal{L}_{t,+1}} \Lambda\right). \tag{8}$$

4.3 Candidate List Generation

In this section, we introduce a trellis based shortest path algorithm to approximately solve the soft detection problem. There are two ways of reducing the number of paths in the trellis. We can either prune the incoming paths or outgoing paths at each vertex.

4.3.1 Edge Reduction

Edge reduction reduces the number of paths by pruning incoming paths. Figure 3a shows that each vertex i at each stage t has Q incoming subpaths $h_0, ..., h_{Q-1}$.

Let the partial distance be d_k, which is the cumulative weight of the subpath h_k from the root to this

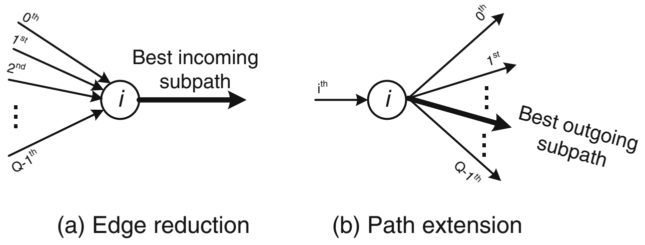


Figure 3 Data flow at vertex $v(t, i)$.

vertex i. Among the Q incoming subpaths, we select the best subpath h_m with the the smallest partial distance,

$$m = \operatorname*{argmin}_{m\in\{0,...,Q-1\}} d_m, \tag{9}$$

and discard the other $Q-1$ subpaths.

4.3.2 Path Extension

Given one incoming path and multiple outgoing paths, path extension reduces the number of paths by pruning outgoing paths. Figure 3b shows that each node i at each stage t has Q outgoing subpaths. The outgoing path weight from node $v(t, i)$ to node $v(t+1, k)$ is updated as

$$d'_k = d_m + w_{i,k}^{<t+1>},\ 0 \le k \le Q-1. \tag{10}$$

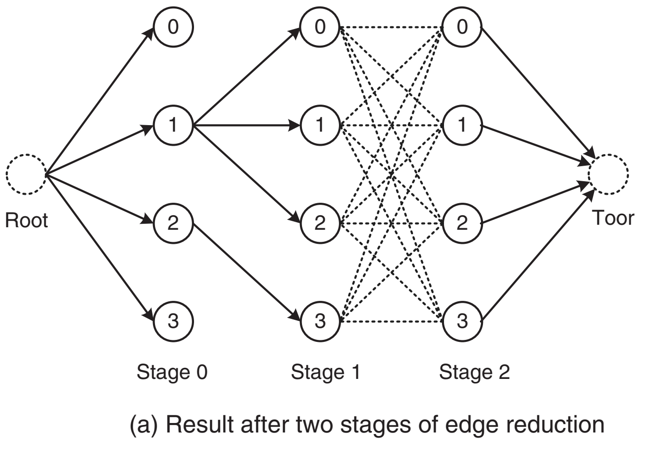
Among the Q outgoing subpaths we find the shortest outgoing subpath h'_n where

$$n = \operatorname*{argmin}_{n\in\{0,...,Q-1\}} d'_n. \tag{11}$$

4.3.3 Shortest Path Algorithm

The goal is to find the shortest path through the trellis for each node i. The search process can be expressed as a series of edge reductions followed by a series of path extensions. To generate $\mathcal{L}_t$, the candidate list for antenna $N-t-1$, we first perform edge reductions stage by stage at each vertex until there is one path per vertex at stage t. If we perform edge reductions after this stage, we can not guarantee that the candidate list has a path from the root to each vertex in stage t. Therefore, after stage t, we perform path extensions stage by stage at each vertex until we have completely traversed the trellis.

Figure 4 illustrates an example of the search process for $\mathcal{L}_1$. We start at the root node and perform edge reduction at each vertex in stage 0. To prune the edges between stage 0 and stage 1, we perform edge reduction at each vertex in stage 1. At this point, there is a path from the root node to each vertex in stage 1 as shown in



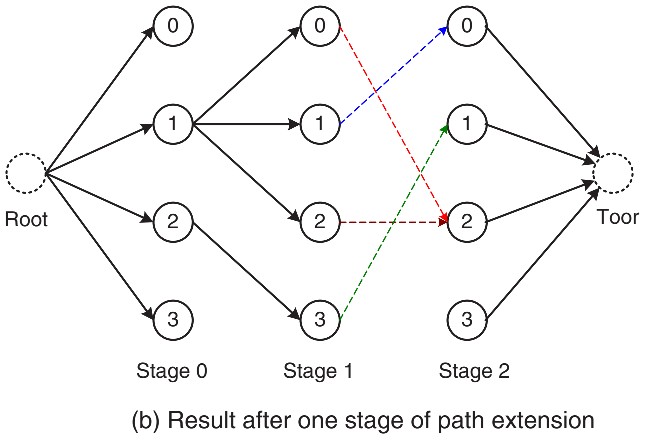


Figure 4 Search process for generating $\mathcal{L}_1$.

Fig. 4a. We then perform edge extension at each vertex in stage 1 to prune the outgoing paths between stage 1 and stage 2. We then have a complete path from root to toor for each vertex in stage 1 as shown in Fig. 4b.

There are common steps when generating candidate lists for each stage. For example, all search processes start with a path reduction at stage 0. The search processes can be represented with a data flow diagram, shown by Fig. 5.

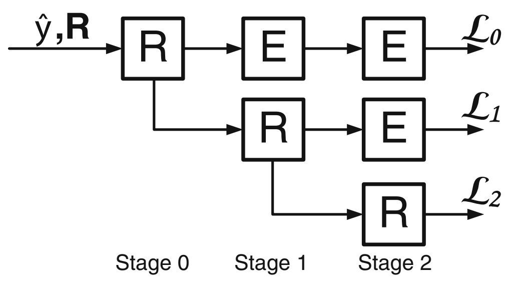


Figure 5 Data-flow diagram for generating candidate lists.

5 Implementing Soft MIMO Detector on CUDA

We implemented the algorithm described in the previous section on GPU. In our implementation, a single kernel generates the candidate lists and computes LLRs for a large number of symbols at a time. At runtime, the kernel spawns a large number of independent soft MIMO detector thread blocks, one thread block for each channel matrix and the corresponding receive vector. Each thread block generates a candidate list for each trellis level and calculates the LLR for each bit using the candidate lists. Effectively the kernel creates a large array of soft MIMO detectors that operate on an array of data in parallel. This reduces overhead since synchronization across different stream multiprocessor is not needed.

Given a receive vector, the corresponding channel matrix and the complex constellation alphabet, a soft MIMO detector block generates the candidate lists through a combination of edge reductions and path extension steps. Given the incoming subpaths and the associated partial distances, each step prunes the number of possible subpaths and outputs the updated subpaths and path partial distances. Both reduction and extension are extremely regular and can be efficiently implemented on the GPU. At each stage, the detector does either Q path reductions or Q edge extensions. Therefore, we can handle the computation by spawning Q threads per thread block, one per each vertex. We use $\log_2(Q)$ threads out of Q threads to perform LLR computation. This section is less parallel than path reduction and path extension. This method does not require terminating a kernel or reading and writing from slower global memory.

We attempt to keep our detector operating at peak utilization by minimizing stall time. This algorithm has a regular memory access pattern which reduces the number of stalls. Furthermore, by using an efficient traversal, we reduce the amount of memory required and allow more concurrent thread blocks to mask stalls. We also improve the performance of the detector by reducing the number of instructions required to perform MIMO detection by sharing computation across threads within a thread block. We unroll loops when possible to reduce instruction count.

We took several additional steps to reduce the overall complexity of the algorithm. Since a reduction step and the edge reduction directly above prune the same set of edges between stage i and stage $i+1$ and have the same set of incoming subpaths, both steps compute the same Q^2 weights. Computation can be reduced by allowing these two steps to share computation.

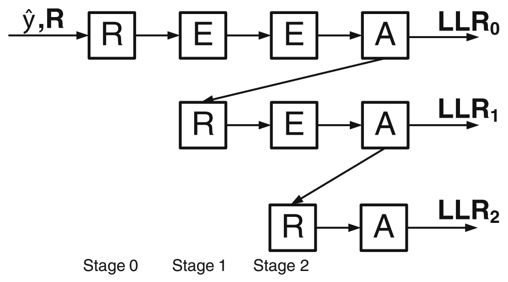


Figure 6 CUDA MIMO detector data flow.

Figure 6 illustrates the steps for a 3×3 MIMO detector. The algorithm generates $\mathcal{L}_0$, $\mathcal{L}_1$, $\mathcal{L}_2$ and $\mathcal{L}_3$ using a series of path reduction and path extension steps followed by LLR computation steps. We will now describe the implementation of each step of the detection algorithm, path extension, path reduction and LLR computation.

5.1 Extension

The inputs to the extension step are the outputs from the previous step. There are Q incoming subpath and Q incoming path partial distances, one subpath and a partial distance per vertex. Since we have Q threads, each thread handles one incoming path by searching for the best path among Q outgoing paths. Particularly, thread k, assigned to vertex k, evaluates all Q outgoing paths for path k. For the path extension corresponding to stage t, the computation for the path weight between vertex k (in stage $t-1$) and vertex q (in stage t) is:

$$w_{k,q}^{<t>} = \left\| \hat{y}_{N-t-1} - \sum_{j=N-t-1}^{N-1} R_{(N-k-1,j)} s_j \right\|_2^2 \tag{12}$$

where h'_k is kth subpath and s_j is the jth element of $\{h'_k, q\}$.

The calculation above is done in two steps to reduce required computation. Thread k first calculates δ_k, the kth intermediate partial distance vector:

$$\delta_k = \sum_{j=N-1-k}^{N-2} R_{(N-1-k,j)} s_j \tag{13}$$

where s_j is the jth element of a kth subpath h'_k.

Thread k now evaluates Q outgoing paths by evaluating each q_k in our complex constellation alphabet Ω.

$$w_{k,q}^{<t>} = \left\| \hat{y}_{N-t-1} - \delta_k - R_{(N-1,N-1)} q_k \right\|_2^2 \tag{14}$$

Thread k picks the smallest outgoing path by evaluating the outgoing paths one by one. The path selected is the new kth path. We update the partial distances as well.

Algorithm 1 summarizes steps taken to find the path with the smallest partial distances. Line 2 calculates δ_k using Eq. 13. Lines 4–17 evaluate Q outgoing paths by evaluating all constellation points in our complex constellation alphabet Ω. Line 12 computes edge weight $w_{k,q}^{<t>}$ and line 13 computes the partial distance, d_k. Lines 14–17 search outgoing paths for the smallest partial distance serially. The path selected is the new kth path.

Algorithm 1 The kth thread searches for the best outgoing path

```
1: //Calculate intermediate PD vectors
2: Calculate $\delta_k$
3: //Search for the path with minimum partial distance serially
4: w = 0
5: Fetch $d'_k$ from shared memory
6: Fetch $\Omega_0$ from shared memory
7: Calculate $w_{k,0}^{<t>}$ using $\delta_k$ and $\Omega_0$
8: Update $d_k$
9: $d_w = d_k$
10: for $q = 1$ to $Q-1$ do
11:     Fetch $\Omega_q$ from constant memory
12:     Calculate $w_{k,q}^{<t>}$ using $\delta_k$ and $\Omega_q$
13:     Update $d_k$
14:     if $(d_k) < (d_w)$ then
15:         $d_w = d_k$
16:     end if
17: end for
18: Store $w$th path into $k$th path history in shared memory
19: Store $w$th path's partial distance in shared memory
20: Sync Barrier
```

For the extension step right above a reduction step, thread k also saves δ_k into shared memory to speed up the next reduction step.

5.2 Reduction

For each iteration of the edge reduction, thread q needs to pick the best path out of Q paths connected to vertex q. For the iteration corresponding to stage t, the path weight between vertex k in stage $t-1$ and vertex q in stage t also can be computed using Eq. 12.

Similar to path extension, each weight calculation can be done in two steps to reduce complexity. However, the extension step above each reduction step already computed all δ_k. The values can be reused which reduce complexity significantly. The search process is similar to path extension except each thread evaluates the incoming path, not each outgoing path. Each thread computes Q partial distances serially and finds the best incoming path with the minimum partial distance. At the end of the iteration, there are Q paths, one path per thread. The paths are written to the shared memory for the next iteration.

The steps in the algorithm are summarized in Algorithm 2. The algorithm works as follows. Each thread calculates Q partial distances serially and finds the path with the minimum partial distance. At the end of the iteration, there are Q paths, one path per thread. The paths and the partial distances are written to the shared memory for the next iteration.

Algorithm 2 The qth thread searches for the best incoming path

```
1: //Search for the path with minimum partial distance serially
2: w = 0
3: Fetch $\delta_0$ from shared memory
4: Fetch $d'_0$ from shared memory
5: Fetch $\Omega_q$ from constant memory
6: Calculate $w_{0,q}^{<t>}$ using $\delta_0$ and $\Omega_q$
7: Update $d_0$
8: $d_w = d_k$
9: for $k = 1$ to $Q - 1$ do
10:    Fetch $d'_k$ from shared memory
11:    Fetch $\delta_k$ from shared memory
12:    Calculate $w_{k,q}^{<t>}$ using $\delta_k$ and $\Omega_q$
13:    Update $d_k$
14:    if $(d_k) < (d_w)$ then
15:       $d_w = d_k$
16:    end if
17: end for
18: Sync Barrier
19: Store $w$th path into $q$th path history in shared memory
20: Store $w$th path's partial distance in shared memory
21: Sync Barrier
```

5.3 LLR Computation

The algorithm generates an LLR for each bit. There are $\log_2(Q)$ parallel LLR computations for each candidate list. The thread block spawns Q threads for the reduction steps and extension steps. The complexity of LLR computation is smaller than the reduction and the extension step. Therefore, we propose a simple linear search. Thread k computes LLR for bit k, where $k < \log_2(Q)$. This method is less efficient than path extension or path reduction as only $\log_2(Q)$ threads are doing useful work.

The Q partial distances from previous steps is the candidate list. We will call these values cumulative distances. To compute the LLR for the kth bit, the kth thread looks at the kth bit, searches for the two smallest cumulative distances, one minimal cumulative distance where the kth bit is 0 and one minimal cumulative distance where the kth bit is 1. The difference between the two cumulative distances is the LLR. The steps in LLR computation are summarized in Algorithm 3.

Algorithm 3 The kth thread compute the kth LLR

```
1: $m_0 = 999$
2: $m_1 = 999$
3: if $k < \log_2(Q)$ then
4:    for $k = 0$ to $Q - 1$ do
5:       if $k$th bit is 0 and $m_0 > d_k$ then
6:          $m_0 = d_k$
7:       else if $k$th bit is 1 and $m_1 > d_k$ then
8:          $m_1 = d_k$
9:       end if
10:   end for
11:   $LLR_{t,k} = \frac{(m_0 - m_1)}{\sigma}$
12: end if
13: Sync Barriers
```

5.4 Memory Transport Scheduling and Code Optimization

Since the GPU is connected to the host through the PCI-express bus, transport time results in a measurable penalty. If synchronous memory copy is used to copy data in and out of the GPU, reading from host memory to global memory, writing from global memory to host memory, and kernel execution can not happen concurrently. However, GPUs support asynchronous memory copy which allows global memory access to overlap with kernel execution. This is accomplished by breaking data into sections and creating a stream per data chuck. By using asynchronous memory copy, while the kernel is performing computation for one stream, memory operations, both reading from host memory to global memory and writing from global memory to host memory can happen in parallel. This minimizes the performance penalty due to transport overhead.

The algorithm described maps efficiently onto a single multiprocessor. The data parallelism of this algorithm is Q. There are Q edge reductions or Q extensions we can do at each stage. Therefore, computations can be balanced evenly across Q threads. However, the minimum number of threads within a *warp* is 32. When $Q = 4$ or $Q = 16$, a *warp* is not completely occupied, reducing the effective throughput. To increase efficiency of the detector, we allow each thread block to consist of more than one MIMO detector, which in turn allows each thread block to have 32 threads and fully occupy one *warp*.

6 Performance Results

Throughout rest of the paper we will refer to our configurable multi-pass trellis traversal real-time MIMO detector on a GPU simply as the "MTT". To evaluate the performance of MTT, we tested our detector on a Linux platform with 8GB DDR2 memory running at 800 MHz and an Intel Core 2 Quad Q6600 running at 2.4 GHz. The GPU used in our experiment is a NVIDIA Telsa C1060 graphic card, which has 240 stream processors running at 1.3 GHz and 4 GB of GDDR3 memory running at 1600 MHz. The host computer first generates the random input symbols and a random channel. After passing the input symbols through the random channel, the host performs sorted QR-decomposition on the channel matrix H to generate $\mathbf{R}$ and $\hat{\mathbf{y}}$, which are fed into the detection kernel running on the GPU.

6.1 MTT Detector Performance

We first evaluate the performance of this detector by comparing the bit error rate (BER) performance against other detectors. We compare MTT against the optimal exhaustive solution which is an exhaustive search. We also compare MTT against the performance of K-Best, a well-known breadth-first algorithm. Finally, to measure how much improvement our MMT detector gains from the additional extension steps, we compared MTT against our first GPU MIMO detector, a one-pass trellis detector (OT), which does only reduction steps through the trellis once to perform detection. To mitigate inaccuracies in LLR computation due to the small list, we apply the LLR clipping technique to the K-Best detector [5] and OT. It should be noted that in the K-Best and one-pass trellis detector algorithm the cumulative distance for a particular bit can be missing due to the small list, so the LLR clipping is necessary in the K-Best algorithm. The LLR clipping is not needed in MTT because each node in the trellis has an associated full Euclidean path. Thus, we can always find a cumulative distance for any bit required in the bit LLR computation.

We run BER simulations using 2×2 and 4×4 4-QAM/16-QAM/64-QAM MIMO systems. The soft output of the detector is fed to a length 2304, rate 1/2 WiMAX layered LDPC decoder [17], which performs up to 15 LDPC iterations. Figure 7(a), (c), and (e) compare the BER performance of the 2×2 MTT with the K-Best detectors. Figure 7(b), (d), and (f) compare the BER performance of the proposed 4×4 detector with the K-Best detectors. As can be seen, MTT performs better than the K-Best detector with $K = Q$ and OT for 2×2 MIMO receiver. This is expected as MTT is the optimal detector for 2×2 as MTT enumerates all possible paths through each trellis vertex. For 4×4 MIMO receiver, MTT performs close to the K-Best detector with $K = Q$. Compared to BER performance of the simple one-pass trellis detector where the trellis is only visited once from left to right, MTT performs better since it evaluates more paths per trellis vertice and is able to compute more accurate LLRs for the decoder.

6.2 MTT Detector Throughput

We now look at the throughput of MTT detector on the GPU. To keep utilization high, a thread block detects multiple symbols in parallel. Each thread block detects eight symbols for 4-QAM, two symbols for 16-QAM, and one symbol for 64-QAM. In our benchmark, both 2×2 and 4×4 MIMO configurations are tested. The detector kernel detects 8 streams of 16,384 symbols for 2×2 and 8 streams of 8,192 symbols for 4×4. Execution time of the detector is averaged over 1,000 runs. We compared both asynchronous memory copy and synchronous memory copy implementations of this MIMO detector.

Table 2 shows the execution time and the throughput performance for the 2×2 MTT MIMO detector. Table 3 shows the execution time and the throughput performance for the 4×4 MTT MIMO detector. The table includes performance of our synchronous implementation, our asynchronous implementation, as well as performance of the kernel of the MIMO detector.

For both 2×2 and 4×4 MIMO configurations, asynchronous memory transfer is an effective way of hiding data transfer latency. By breaking incoming data into eight streams and overlapping transfer and computation, our MIMO detector performs very close to kernel running time.

Figures 8 and 9 compare the throughput of the proposed MTT detector with asynchronous memory transfer to the performance requirement of a 5 MHz LTE downlink MIMO configuration.

To meet the LTE requirement, our detector needs to meet both throughput and latency requirements of

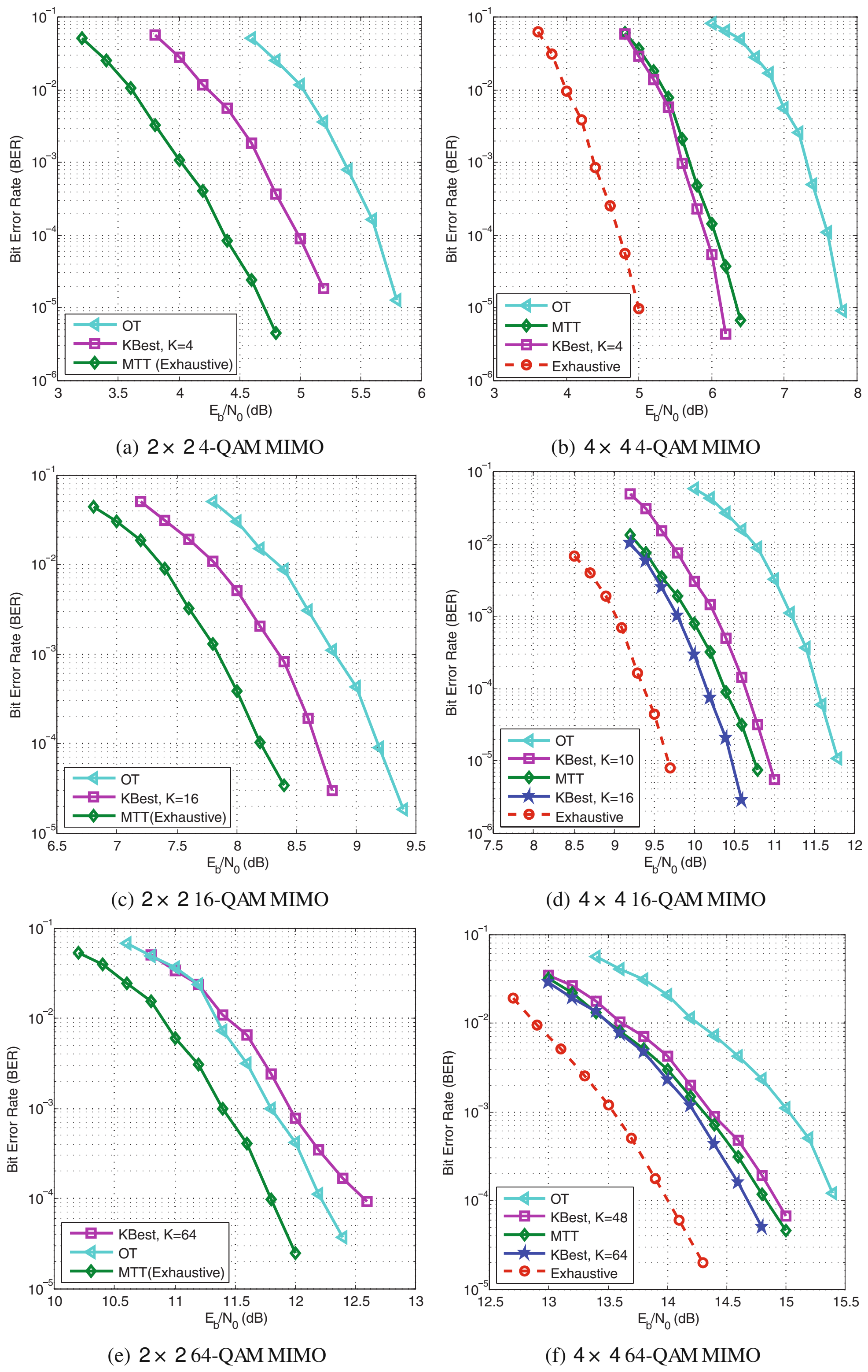


Figure 7 Simulation results for an LDPC-coded MIMO system.

(a) 2×2 4-QAM MIMO

(b) 4×4 4-QAM MIMO

(c) 2×2 16-QAM MIMO

(d) 4×4 16-QAM MIMO

(e) 2×2 64-QAM MIMO

(f) 4×4 64-QAM MIMO

this standard. Although the current LTE specification does not define the exact end to end latency requirement, the specification aims to enable <5 ms latency for small packets. Using 5 ms as the maximum latency required between a mobile device and the base-station, our detector can handle 4 and 16 QAM for 2×2 and 4×4 5 MHz LTE MIMO systems. Since the detector can achieve more than four times the performance

Table 2 Average runtime for 2 × 2 MIMO detection.

	Runtime (ms)/throughput (Mbps)		
Q	Synchronous	Asynchronous	Kernel
4	5.05/99.10	0.75/663.65	0.61/822.59
16	9.49/105.27	3.70/269.89	3.57/280.08
64	46.85/37.35	39.97/43.91	39.80/43.86

requirement of 5 MHz LTE MIMO configuration for 4 and 16-QAM for 2 × 2 and for 4 QAM 4 × 4 LTE MIMO system, our detector can also handle the larger 20 MHz LTE MIMO configuration for these cases. Furthermore, for these cases, our detector can meet the latency requirement in asynchronous mode as well.

6.3 Detector Instruction Throughput Ratio

The current implementation attempts to maximize efficiency by ensuring each thread block is a multiple of 32 threads. By employing a regular algorithm that allows regular memory access, stall time can be reduced. CUDA Visual Profiler provides the instruction throughput ratio in the summary table. This metric measures efficiency of the mapping as it is the ratio of achieved instruction rate to peak single issue instruction rate [14]. Accordingly, the achieved instruction rate is I/T, where I is the number of executed *warp* instructions and T is the actual time in *ms* it takes to run the algorithm. The peak single instruction rate is F_c/CPI, where F_c is clock frequency and CPI is the average number of cycles per instruction, Therefore, the instruction throughput ratio can be calculated as:

$$R = \frac{I/T}{F_c/CPI} = \frac{I \times CPI \times F_c^{-1}}{T} \tag{15}$$

In CUDA, the average CPI is 4 cycles per instruction and each SM is clocked at 1.3 GHz. The estimated runtime is shown in Table 4.

The ratio, R, is smaller than 1 since an instruction throughput ratio of 1 corresponds to the maximum instruction throughput. Instruction throughput ratio is lowest for 4 QAM since the detector does a smaller number of computations per global memory fetch. Conversely, instruction throughput ratio is close to 1 for 16 and 64 QAM as the numbers of stalls due to long device memory access for the computation intensive cases decreases as the detector does more computations per each global memory fetch.

Table 3 Average runtime for 4 × 4 MIMO detection.

	Runtime (ms)/throughput (Mbps)		
Q	Synchronous	Asynchronous	Kernel
4	12.52/39.90	1.76/284.75	1.62/308.40
16	19.85/50.35	8.31/120.25	8.19/122.03
64	138.17/10.85	124.62/12.04	124.52/12.05

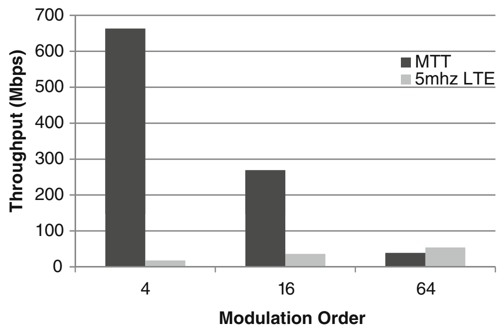


Figure 8 Performance compared to 5 MHz LTE 2 × 2 MIMO.

6.4 Detector Instruction Mix

Instruction throughput ratio measures how well instructions for our MIMO detector execute on the hardware. However, it does not measure how well these instructions solve our problem. We use *decuda* [18], a disassembler, to study the quality of detector code generated by the CUDA compiler. The main steps of the algorithm are edge reductions, path extensions and LLR computations. We measure the quality of the instructions that make up our detector by looking at the loop body within these three functions. Using the disassembler, we see that path extensions, path reductions, and LLR computations are completely unrolled for all cases except for 4 × 4 64-QAM. Particularly, path extension and path reduction are essentially the

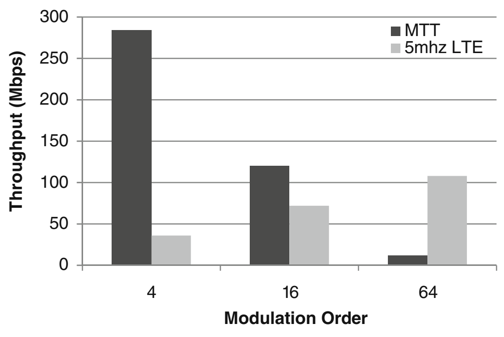


Figure 9 Performance compared to 5 MHz LTE 4 × 4 MIMO.

Table 4 Instruction throughput ratio for 2×2, 16,800 subcarriers.

Modulation	I	T	R
4 QAM	13,894	0.08	0.549
16 QAM	137,712	0.45	0.940
64 QAM	1,601,220	4.98	0.996

same. Each loop iteration consists of two *add*, two *ABS* and two additional *add* instructions, which is the minimum number of instructions needed to compute the partial distance of each incoming path. The if statement within these loops consists of three instructions, one compare instruction that sets the predication register and two predicated *min* for computing the minimum cumulative weights thus far for the next iteration. For both path extension and reduction, there are a total of eight instruction per loop iteration. For 4×4, there is an additional store to save the index of the best path. For the LLR computation, each loop iteration consists of one compare, one shared memory load and two stores.

For the 2×2 configuration, there is one reduction step, one extension step and two LLR computation steps. After counting the number of instructions outside of the loop, the number of instructions (I) required for our MIMO detector is modeled as the following:

$$I = 113 + 16Q + 8Q \tag{16}$$

For the 4×4 configuration, there are three path reductions, six path extensions and four LLR computations. After counting the number of instructions outside of the loop, the number of instructions required for our MIMO detector is modeled as the following:

$$I = 600 + 81Q + 16Q \tag{17}$$

The constants for the 2×2 and 4×4 MIMO cases are different since there are more computations outside of the loops for 4×4 MIMO case. For example, there are more intermediate partial distance vectors (Eq. 13) for 4×4 MTT to compute. Table 5 compares our model against the number of instruction reported by the NVIDIA profiler for one thread block. Note that the result reported by the profiler is approximate as it varies by a few instructions from run to run.

When Q is large, most instructions are loop iterations. For example, for 2×2 MIMO configuration, 74 percent of the instructions are in loops for 16-QAM. Similarly 93 percent of the instructions are loop iterations for 64-QAM. This is also true for 4×4 16-QAM. The model does not accurately predict the number of instructions executed for 4×4 64-QAM since loops are not completely unrolled. This is due to the tradeoff between code unrolling and code expansion. Nevertheless, mapping is efficient since the code structure matches the computation required and the instruction throughput ratio shows few stalls, which implies efficient memory accesses.

Table 5 Number of instructions per threadblock.

	2×2		4×4	
	Model	Profiler	Model	Profiler
4	209	209	988	1,014
16	497	496	2,152	2,137
64	1,649	1,780	6,808	9,749

6.5 ASIC/FPGA/ASIP Comparisons

Although a conventional MIMO ASIC detector could achieve higher throughput with fewer silicon resources, it lacks the necessary flexibility to support different modulation orders and different number of antennas. Moreover, the fixed-point arithmetic employed by the ASIC has to be designed very carefully to avoid large performance degradation. For example, the internal bit width could be large due to the correlation of the channel matrices and the “colored noise". This is not a concern for GPU since GPU executes all computations in floating-point.

Table 6 compares our GPU design with state-of-the-art ASIC/FPGA/ASIP designs in terms of throughput. Compared to our previous work [20], this work is a more complete comparison since it is also a soft detector. In [8], a depth-first search detector with 256 searches per level is implemented. In [9], a K-best detector with $K = 5$ and real decomposition is implemented. In [4], a relaxed K-best detector with $K = 48$ is implemented. In [2], a K-best with $K = 7$ detector is implemented. We also list our early ASIC design [16] based on the same trellis detection algorithm described above. As can be seen, the proposed detection algorithm is not only suitable for parallel ASIC implementation but also suitable for GPU-based parallel software implementation. Compared to ASIC/FPGA/ASIP solutions from [2, 4, 8, 9] for 4×4 MIMO systems, our GPU design can achieve comparable or even higher throughput. In summary, the GPU design has more flexibility to support different MIMO system configurations and has the capability to support floating-point signal processing which can eliminate the need for fixed-point design analysis.

Table 6 Throughput comparison with ASIC/FPGA/ASIP solutions for 4 × 4 system.

	4 × 4 QPSK	4 × 4 16-QAM	4 × 4 64-QAM	Output type
GPU	284.7 Mbps	120.0 Mbps	12.0 Mbps	Soft decision
FPGA [4]	N/A	N/A	8.57 Mbps	Soft decision
ASIP [2]	N/A	5.3 Mbps	N/A	Hard decision
ASIC [8]	19.2 Mbps	38.4 Mbps	N/A	Soft decision
ASIC [9]	N/A	53.3 Mbps	N/A	Soft decision
ASIC [16]	300 Mbps	600 Mbps	N/A	Soft decision

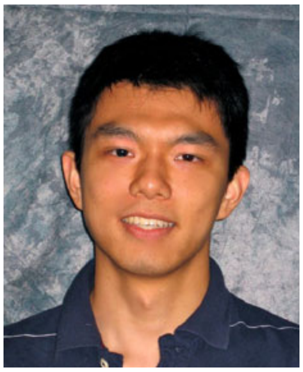
7 Conclusion

This paper presented a soft MIMO detector implementation on GPU. We show our proposed multi-pass trellis traversal performs similarly to soft K-best MIMO detector with clipping and out-performs one-pass trellis traversal with LLR clipping. By using the NVIDIA profiler to measure how well the compiled code runs on GPU and the disassembler to study the quality of detector code generated by the CUDA compiler, we show that this algorithm is well-suited to the GPU. Our detector performs at least as well as the conventional fixed-point VLSI and FPGA implementations while maintaining support for different MIMO system configurations, allowing the fast yet flexible wireless physical layer simulation as well as high throughput MIMO enabled software defined radio.

Acknowledgements This work was supported in part by Nokia, NSN, Texas Instruments, Xilinx, and by NSF under grants CCF-0541363, CNS-0551692, CNS-0619767, EECS-0925942 and CNS-0923479.

References

1. Amiri, K., Sun. Y., Murphy, P., Hunter, C., Cavallaro, J. R., et al. (2007). Warp, a unified wireless network testbed for education and research. In *MSE '07: Proceedings of the 2007 IEEE international conference on microelectronic systems education*.
2. Antikainen, J., Salmela, P., Silven, O., Juntti, M., Takala, J., & Myllyla, M. (2007). *Application-specific instruction set processor implementation of list sphere detector*. EURASIP Journal on Embedded Systems.
3. Burg, A., Borgmann, M., Wenk, M., Zellweger, M., Fichtner, W., & Bolcskei, H. (2005). VLSI implementation of MIMO detection using the sphere decoding algorithm. *IEEE Journal Solid-State Circuit, 40*, 1566–1577.
4. Chen, S., Zhang, T., & Xin, Y. (2007). Relaxed K-best MIMO signal detector design and VLSI implementation. *IEEE Transactions on Very Large Scale Integration (VLSI) System, 15*, 328–337.
5. de Jong, Y. L. C. , & Willink, T. J. (2002). Iterative tree search detection for MIMO wireless systems. *IEEE Transactions on Communications, 53*(6), 930–935.
6. Falcão, G., Silva, V., & Sousa, L. (2009). How GPUs can outperform ASICs for fast LDPC decoding. In *ICS '09: Proceedings of the 23rd international conference on supercomputing*.
7. Fincke, U., & Pohst, M. (1985). Improved methods for calculating vectors of short length in a lattice, including a complexity analysis. *Mathematics of Computation, 44*(170),463–471.
8. Garrett, D., Davis, L., ten Brink, S., Hochwald, B., & Knagge, G. (2004). Silicon complexity for maximum likelihood MIMO detection using spherical decoding. *IEEE Journal of Solid-State Circuit, 39*, 1544–1552.
9. Guo, Z., & Nilsson, P. (2006). Algorithm and implementation of the K-best sphere decoding for MIMO detection. *IEEE Journal on Selected Areas in Communication, 24*, 491–503.
10. Hochwald, B., & Brink, S. (2003). Achieving near-capacity on a multiple-antenna channel. *IEEE Transactions on Communications, 51*, 389–399.
11. Huang, X., Liang, C., & Ma, J. (2008). System architecture and implementation of MIMO sphere decoders on FPGA. *IEEE Transactions on Very Large Scale Integration (VLSI) System, 2*, 188–197.
12. Janhunen, J., Silvn, O., & Juntti, M. (2010). Programmable processor implementations of K-best list sphere detector for MIMO receiver. *Signal Processing, 90*(1), 313–323.
13. NVIDIA Corporation (2008). CUDA compute unified device architecture programming guide. http://www.nvidia.com/object/cuda_develop.html.
14. NVIDIA Corporation (2009). NVIDIA CUDA visual profiler version 2.2 readme. http://developer.download.nvidia.com/compute/cuda/2_2/toolkit/docs/cudaprof_1.2_readme.html.
15. Qi, Q., & Chakrabarti, C. (2007). Sphere decoding for multiprocessor architectures. In *IEEE workshop on signal processing systems* (pp. 17–19).
16. Sun, Y., & Cavallaro, J. R. (2009). High throughput vlsi architecture for soft-output mimo detection based on a greedy graph algorithm. In *GLSVLSI '09: Proceedings of the 19th ACM great lakes symposium on VLSI*. ACM.
17. Sun, Y., & Cavallaro, J. R. (2008). A low-power 1-Gbps reconfigurable LDPC decoder design for multiple 4G wireless standards. In *IEEE international SOC conference* (pp. 367–370).
18. van der Laan, W. J. (2009). Decuda. http://wiki.github.com/laanwj/decuda.
19. Wong, K., Tsui, C., Cheng, R., & Mow, W. (2002). A VLSI architecture of a K-best lattice decoding algorithm for MIMO channels. In *IEEE int. symp. on circuits and syst.* (Vol. 3, pp. 273–276).
20. Wu, M., Sun, Y., & Cavallaro, J. R. (2009). Reconfigurable real-time MIMO detector on GPU. In *IEEE 43rd asilomar conference on signals, systems and computers (ASILOMAR'09)*.
21. Wu, M., Gupta, S., Sun, Y., & Cavallaro, J. R. (2009). A GPU implementation of A real-time MIMO detector. In *IEEE workshop on signal processing systems (SiPS'09)*.



Michael Wu received his B.S. degree from Franklin W. Olin College of Engineering in May of 2007 and his M.S. degree from Rice University in May of 2010, both in Electrical and Computer Engineering. He is currently a Ph.D candidate in the E.C.E department at Rice University. His research interests are wireless algorithms, software defined radio on GPGPU and other parallel architectures, and high performance wireless receiver designs.

Yang Sun received the B.S. degree in Testing Technology & Instrumentation in 2000, and the M.S. degree in Instrument Science & Technology in 2003, both from Zhejiang University, Hangzhou, China. From 2003 to 2004, he worked at S3 Graphics Co. Ltd. as an ASIC design engineer, developing 3D Graphics Processors (GPU) for computers. From 2004 to 2005, he worked at Conexant Systems Inc. as an ASIC design engineer, developing Video Decoders for digital satellite-television set-top boxes (STBs). He is currently a Ph.D student in the Department of Electrical and Computer Engineering at Rice University, Houston, Texas. His research interests include parallel algorithms and VLSI architectures for wireless communication systems, especially forward-error correction (FEC) systems. He received the 2008 IEEE SoC Conference Best Paper Award, the 2008 IEEE Workshop on Signal Processing Systems Best Paper Award (Bob Owens Memory Paper Award), and the 2009 ACM GLSVLSI Best Student Paper Award.

Siddharth Gupta received his B.S. degree in Electrical Engineering in 2007 and his M.S. degree in Electrical Engineering in 2010, both from Rice University, Houston, TX. Since 2009, he has been a Research Engineer at the Center for Multimedia Communications at Rice University, Houston, TX. His research interests are in platforms for wireless communications, low power systems and sensor processing.

Joseph R. Cavallaro received the B.S. degree from the University of Pennsylvania, Philadelphia, Pa, in 1981, the M.S. degree from Princeton University, Princeton, NJ, in 1982, and the Ph.D. degree from Cornell University, Ithaca, NY, in 1988, all in electrical engineering. From 1981 to 1983, he was with AT&T Bell Laboratories, Holmdel, NJ. In 1988, he joined the faculty of Rice University, Houston, TX, where he is currently a Professor of electrical and computer engineering. His research interests include computer arithmetic, VLSI design and microlithography, and DSP and VLSI architectures for applications in wireless communications. During the 1996–1997 academic year, he served at the National Science Foundation as Director of the Prototyping Tools and Methodology Program. He was a Nokia Foundation Fellow and a Visiting Professor at the University of Oulu, Finland in 2005 and continues his affiliation there as an Adjunct Professor. He is currently the Associate Director of the Center for Multimedia Communication at Rice University. He is a Senior Member of the IEEE. He was Co-chair of the 2004 Signal Processing for Communications Symposium at the IEEE Global Communications Conference and General Co-chair of the 2004 IEEE 15th International Conference on Application-Specific Systems, Architectures and Processors (ASAP).