A GPU Implementation of a Real-Time MIMO Detector

Michael Wu, Siddharth Gupta, Yang Sun, and Joseph R. Cavallaro
Electrical and Computer Engineering
Rice University
Houston, Texas 77005
{mbw2, sgupta, ysun, cavallar}@rice.edu

Abstract—Multiple-input multiple-output (MIMO) is an existing technique that can significantly increase throughput of the system by employing multiple antennas at the transmitter and the receiver. Realizing maximum benefit from this technique requires computationally intensive detectors which poses significant challenges to receiver design. Furthermore, a flexible detector or multiple detectors are needed to handle different configurations. Graphical Processor Unit (GPU), a highly parallel commodity programmable co-processor, can deliver extremely high computation throughput and is well suited for signal processing applications. However, careful architecture aware design is needed to leverage performance offered by GPU. We show we can achieve good performance while maintaining flexibility by employing an optimized trellis-based MIMO detector on GPU.

I. INTRODUCTION

Wireless communication systems enable higher data rate services by providing higher spectral efficiency. Multiple-input multiple-output combined with orthogonal frequency division multiplexing (MIMO-OFDM) is used in current and upcoming standards such as WiMAX and 3GPP LTE. 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. As the received signal at each antenna for each subcarrier is a signal that 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.

The typical suboptimal MIMO detectors are fixed ASIC designs operating with a specific configuration and a predetermined workload in mind. For example, Burg *et al.* [1] implemented a depth-first 4×4 16-QAM detector in ASIC with an average throughput of 73 Mbps at a signal-to-noise ratio (SNR) of 20 dB. Wong *et al.* [2] first introduced a K-best (with $K=10$) 4×4 16-QAM detector in ASIC achieving 10 Mbps. Later on Guo and Nilsson [3] developed a KSE (with $K=5$) 4×4 16-QAM detector in ASIC with a higher throughput of 53.3 Mbps. In addition, researchers have investigated many other ways of hardware implementations. Huang *et al.* [4] 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 SE and VB algorithm

respectively. Qi and Chakrabarti [5] mapped a depth-first detector on a multi-core software radio architecture (SDR). Antikainen *et al.* [6] presented an application-specific instruction set processor (ASIP) implementation of K-best detector.

Beside these traditional solutions, graphic processor unit (GPU) has become a viable alternative to high performance accelerators for several reasons. First, GPU delivers extremely high computation throughput by employing many cores to execute a common set of operations on a large set of data in parallel. Many communication algorithms are inherently data parallel and computationally intensive, and can take advantage of highly parallel computation offered by GPU. Second, although custom ASIC and FPGA are capable of delivering higher throughput than GPU, GPU can deliver real-time throughput and continues to grow exponentially in term of performance. Combined with the fact that these types of processors are extremely cost-effective and ubiquitous in mobile and desktop devices, communication algorithms in the future can be offloaded onto this type of processor in place of custom ASIC or FPGA. Finally, unlike ASIC, GPU is extremely flexible and can be reconfigured on the fly to handle different workloads.

However, careful architecture aware algorithm design is needed to achieve high performance on the GPU. For example, due to the limited amount of resource on GPU, such as on-chip memory and/or long latency due to software sorting [7], many existing algorithms such as depth first sphere detector and K-best detector do not map very efficiently onto this architecture. Furthermore, designing a detection algorithm that scales well, keeping the cores fully utilized to achieve peak throughput across different combinations of number of antennas and different modulation, is a difficult task.

The most intensive baseband processing blocks in a MIMO receiver are the detector and the channel decoder. Falcão *et al.* [8] presented an LDPC decoder capable of real time throughput on GPU. In this paper, we will explore a new design paradigm by also applying the GPU for real time MIMO detection. We aim to show that an optimized trellis-based MIMO detector can achieve good performance while maintaining flexibility offered by programmable hardware. In section II we will cover the system model and in section III give an overview of the detection algorithm. This is followed by a discussion on the CUDA architecture in section IV and an overview of the implementation in section V. Finally, we will give performance results in section VI

and conclude in section VII.

II. SYSTEM MODEL

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

$$\mathbf{y} = \mathbf{H}\mathbf{s} + \mathbf{w} \quad (1)$$

where $\mathbf{y} = [y_0, y_1, \dots, y_{N-1}]^T$ is the received vector. $\mathbf{H}$ is the $N \times N$ channel matrix, where each element, $H_{i,j}$, represents the complex gain between i th transmit antenna and j th receive antenna and is an independent zero mean circularly symmetric complex Gaussian random variable with unit variance. Noise at the receiver is $\mathbf{w} = [w_0, w_1, \dots, w_{N-1}]^T$, where w_i is also an independent zero mean circularly symmetric complex Gaussian random variables with σ^2 variance. The transmit vector is $\mathbf{s} = [s_0, s_1, \dots, s_{N-1}]$, where s_i is drawn from a finite complex constellation alphabet, Ω , of cardinality Q . The constellation alphabets we consider in this paper are QPSK, 16QAM, 64QAM and 256QAM. 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 $N \times N$ MIMO system with an equivalent model:

$$\mathbf{y} = \mathbf{Q}\mathbf{R}\mathbf{s} + \mathbf{w} \quad (2)$$

$$\hat{\mathbf{y}} = \mathbf{R}\mathbf{s} + \hat{\mathbf{w}} \quad (3)$$

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

III. TRELLIS-BASED MIMO DETECTION

The hard decision maximum likelihood (ML) detector finds the transmit vector, $\mathbf{s}$, that minimizes the euclidean distance between the received vector, $\hat{\mathbf{y}}$ and the expected received vector $\mathbf{R}\mathbf{s}$. It can be expressed as:

$$\underset{\mathbf{s} \in \Omega^N}{\operatorname{argmin}} \|\hat{\mathbf{y}} - \mathbf{R}\mathbf{s}\|_2^2 \quad (4)$$

The hard decision maximum likelihood (ML) detector can also be represented using a trellis graph.

A. Graph construction

Without loss of generality, we use a 4×4 QPSK system in this section to construct our graph. To find the transmit vector with the smallest euclidean distance, we need to calculate the euclidean distance of each transmit vector. Given a transmit vector, $\mathbf{s}$, the euclidean distance is defined as:

$$\Lambda = \|\hat{\mathbf{y}} - \mathbf{R}\mathbf{s}\|_2^2$$

$$\Lambda = \left\| \begin{bmatrix} \hat{y}_0 \\ \hat{y}_1 \\ \hat{y}_2 \\ \hat{y}_3 \end{bmatrix} - \begin{bmatrix} R_{00} & R_{01} & R_{02} & R_{03} \\ 0 & R_{11} & R_{12} & R_{13} \\ 0 & 0 & R_{22} & R_{23} \\ 0 & 0 & 0 & R_{33} \end{bmatrix} \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix} \right\|_2^2$$

The calculation of Λ can be decomposed as: $\Lambda = w^{<0>} + w^{<1>} + w^{<2>} + w^{<3>}$, where $w^{<t>}$ is the 1-D Euclidean distance for t th antenna and is calculated as

$$\begin{aligned} w^{<0>} &= \|\hat{y}_3 - R_{33}s_3\|^2, \\ w^{<1>} &= \|\hat{y}_2 - (R_{22}s_2 + R_{23}s_3)\|^2, \\ w^{<2>} &= \|\hat{y}_1 - (R_{11}s_1 + R_{12}s_2 + R_{13}s_3)\|^2, \\ w^{<3>} &= \|\hat{y}_0 - (R_{00}s_0 + R_{01}s_1 + R_{02}s_2 + R_{03}s_3)\|^2. \end{aligned} \quad (5)$$

The search space of all transmit vectors is shown in Figure 1, where each transmit vector is a path through the trellis. There are 4 trellis stages, one stage per antenna. Each stage has 4 vertices, corresponding to constellation points. A path through the trellis from root to toor is a transmit vector. The edge between the k th vertex at stage $t-1$, $v(t-1, k)$, and the i th vertex at stage t , $v(t, i)$, has a weight of $w_{k,i}^{<t>}$. The weight function does not depend on the future stages, but only depends on its current stage and all its predecessors. For example, $w_{k,i}^{<2>}$ depends on the vertices in stages 2, 1, and 0.

Fig. 1: Flow graph for MIMO detection

B. Greedy Shortest Path Algorithm

One way of finding the most likely transmit vector is through exhaustive search. Searching through all possible transmit vectors is a time intensive process. For example, if the transmitter is utilizing 64-QAM, then the total number of possible transmit vectors is $64^4 = 16,777,216$. Therefore, instead of finding the best transmit candidate by evaluating all complete paths through the trellis, a greedy shortest path algorithm [9, 10] can approximately solve the hard detection problem in this section. In this greedy algorithm, we prune the incoming paths at each vertex.

Edge reduction reduces the number of paths by pruning incoming paths. First we connect the Q surviving path, $h'_0, \dots, h'_{Q-1}$, to a vertex i .

$$h_k = \{h'_k, i\}, 0 \leq k \leq Q-1 \quad (6)$$

The new cumulative weight for path k , d_k , is generated by adding the path weight to vertex i , $w_{k,i}^{<t>}$, to the old cumulative weight for path k , d'_k .

$$d_k = d'_k + w_{k,i}^{<t>} \quad (7)$$

Figure 2 shows that each vertex i at each stage t has Q incoming subpaths $h_0, \dots, h_{Q-1}$ and the newly updated partial distance, d_k ,

Fig. 2: Data flow at vertex $v(t, i)$

which is the cumulative weight of the subpath h_k from the root to this vertex i . Among the Q incoming subpaths, we select the best subpath h_m with the minimum weight

$$m = \underset{m \in \{0, \dots, Q-1\}}{\operatorname{argmin}} d_m, \quad (8)$$

and discard the other $Q - 1$ subpaths.

The goal is finding the shortest path through the trellis. The search process can be expressed as a series of edge reductions. We perform edge reductions until we have completely traversed the trellis. Figure 3 shows an example of the result graph after applying edge reductions at each vertex. At stage 3, we have four surviving paths, one path per vertex at stage 3. The best path is chosen among the four remaining paths at the toor.

Fig. 3: Search process using edge reduction

IV. COMPUTE UNIFIED DEVICE ARCHITECTURE (CUDA)

Compute Unified Device Architecture [11] is a software programming model that exposes the massive computation potential offered by the programmable GPU. The raw computation power is enabled by many cores on the programmable GPU. A GPU can have multiple stream multiprocessors (SM), where each stream multiprocessor consists of eight pipelined cores. During execution, all cores in a SM execute a single 32-bit integer or float operation on multiple data in parallel. A CUDA device has a large amount (up to 1GB) of off-chip device memory (global memory). In addition, fast on-chip resources, such as registers, shared memory and constant memory can be used in place of off-chip global memory to keep the computation throughput high.

In this model, the programmer defines the kernel function, a set of common operations. At runtime, the kernel spawns a large number of threads blocks. Each thread block contains multiple threads, up to a 512 threads per thread block. Each

thread can select a set of data using its own unique ID and executes the kernel function on the set of data. Threads executes independently in this model. However, threads within a block can synchronize through a barrier and writing to shared memory. In contrast, thread blocks are completely independent and can be synchronized by terminating the kernel and writing to global memory.

During kernel execution, multiple thread blocks can be assigned to a SM and is executed concurrently. CUDA divides threads within a thread block into blocks of 32 threads. These 32 threads are executed as a group using the same common instruction, a *warp* instruction, at each step. A SM issues a *warp* instruction whose operands are ready. To mask the pipeline latency and memory stalls, a SM can switch and issue an independent *warp* instruction from the the same thread block or another concurrent thread block with zero-overhead. Therefore, to achieve peak performance, one would like to map many concurrent threads onto a SM. The five types of memory that a

TABLE I: Available resources for each memory

Type	Speed	Access	Size
Register	fast	RW	8192 per multiprocessor
Shared Memory	fast	RW	16 KB per multiprocessor
Constant Memory	fast	RO	8 KB per multiprocessor
Texture Memory	fast	RO	8 KB per multiprocessor
Global Memory	slow	RW	> 512 MB per device

thread can access are register, shared memory, constant memory, texture memory, and global memory (resource for each are shown in Table I). Before the start of the program, the host copies data from system memory into global memory and constant memory which all threads on the device can access. This is slow as the GPU is often connected to the host through PCI-express bus.

Fetching data from global memory, the external DRAM, also results in a latency penalty. Registers, shared memory, and constant memory can reduce memory access time by reducing global memory access. Registers and share memory are on-chip resources. Shared memory is slower than registers, but can be accessed by threads within a thread block. However, shared memory on each SM has 16 access ports. If 16 threads, half of a *warp*, are scheduled to access shared memory at the same time, they must have certain conditions to allow the instruction to execute with one load or store. It takes one load or store if all threads access the same port (broadcast) or none of the threads accesses the same port. However, random layout with some broadcast and some one-to-one accesses will be serialized and cause a stall. Although constant memory resides in global memory, memory access to this memory space is cached. Like shared memory, it takes one cached read if all threads access the same location. Unlike shared memory, all other constant memory access patterns will be serialized and need global memory access.

Both shared memory and registers are divided among concurrent threads on a SM. Since there are only 16 KB of shared memory and 8196 registers per stream multiprocessor, designing an algorithm that effectively partitions registers and shared memory such that at least a few blocks can be mapped onto the same processor, each thread has an efficient memory access

pattern is a non-trivial task. Trellis based detection algorithm works particularly well with this architecture.

V. MAPPING MIMO DETECTION ON CUDA

Typically there are hundreds of subcarriers in many high data rate standards such as LTE. In our implementation, each stream multiprocessor handles a set of subcarriers independently. There is no need to synchronize across stream multiprocessors.

The algorithm proposed in section III maps efficiently onto a single multiprocessor. The data parallelism of this algorithm is Q —there are Q edge reductions we can do at each stage. Therefore, computation can be balanced evenly across Q threads. However, the minimum number of threads within a *warp* is 32. When $Q = 4$ and $Q = 16$, 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*.

Therefore, the implemented MIMO detection kernel creates many thread blocks that executes in parallel. Depending on Q , each thread block consists of one or more MIMO detectors. Each MIMO detector, has Q threads, performs edge reduction iteratively to estimate the input vector given a unique channel matrix and a receive vector. To keep the detectors operating at peak utilization, on-chip memory is used to reduce access penalty. In addition, computation is shared among threads to reduce complexity.

A. Memory Utilization

A *warp* will not execute until all operands are ready, therefore it is desirable to reduce memory access time. Threads in our detector need to fetch $\mathbf{R}$, $\hat{\mathbf{y}}$, $\mathbf{s}$, and fetch/write path history, partial euclidean vector, and cumulative euclidean distance. The trellis-based detection algorithm memory access is regular enough such that memory access often can be parallelized, leading to low latency memory access.

In our detector, shared memory stores data that needs be shared across threads: path history, partial euclidean vector, and cumulative euclidean distance. The addressing pattern for vectors in shared memory is either linear or broadcast, causing no conflicts. Inputs to the detector, channel matrix $\mathbf{R}$, the received vector $\hat{\mathbf{y}}$, and $\mathbf{\Omega}$, are stored in the constant memory. The memory access pattern for $\mathbf{R}$ and $\hat{\mathbf{y}}$ is broadcast in our detector for $Q = 16$, $Q = 64$ and $Q = 256$. For these cases, constant memory access is cached since each thread within a half of *warp* always access the same element in $\mathbf{R}$ and $\hat{\mathbf{y}}$. However, a thread block for $Q = 4$ consists of multiple Viterbi MIMO detectors. In this case, each half *warp* needs simultaneous access to four different elements in $\mathbf{R}$ and in $\hat{\mathbf{y}}$. This is less efficient as the first constant memory read is cached and the subsequent memory accesses are serialized.

B. Algorithm Implementation

The detection process can be divided into several steps. In the first step, each thread calculates the initial partial distance for a edge from root to a vertex in trellis stage 0. The results are stored

into shared memory. In the second step, the detector does edge reduction iteratively until there is one path per trellis vertex at stage $N - 1$. Each successive iterations of edge reduction can be divided into three stages. In the first stage, each thread calculates the partial distance vector of an incoming path. This immediate result is stored into the shared memory, which is shared across all threads for the next stage. In the second stage, each thread evaluates Q possible paths by calculating Q partial distances by using vectors calculated in previous iteration. The best path, one with the lowest partial distance, is picked. In the third stage, Q best paths, one per state, are stored into shared memory for the next iteration. In the last step, our detector picks the path through the trellis that has the lowest partial distance. The algorithm is summarized in Figure 4.

Fig. 4: Summary of Algorithm Implementation

1) *Calculate the Initial Partial Distance (Stage 0)*: There are Q edges between the root node and vertices at stage 0. For the initial calculation, thread q calculates the partial distance for a edge between the root node and q^{th} vertex.

$$w_{-,q}^{<0>} = \|\hat{\mathbf{y}}_{N-1} - R_{(N-1,N-1)}q\|_2^2$$

Since all threads fetch the same element from $\mathbf{R}$ and $\hat{\mathbf{Y}}$, this results in a broadcast which is handled efficiently by constant memory. Thread q then stores vertex q into q^{th} path history and $w_{-,q}^{<0>}$ into the cumulative partial distance for trellis q . Addressing is linear in this case. After synchronization, when all threads finish executing this step, the algorithm proceeds to the iterative stage.

2) *Edge Reduction Iterations, (Stage 1 to Stage $N - 1$)*: For each iteration of the edge reduction, Q surviving paths connect to each vertex. 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 and vertex q can be expressed as:

$$w_{k,q}^{<t>} = \left\| \hat{\mathbf{y}}_{N-t-1} - \sum_{j=N-t-1}^{N-1} R_{(N-k-1,j)} s_j \right\|_2^2 \quad (9)$$

where s_j is the j^{th} element of k^{th} subpath from the previous level. And can be expressed as $\{h'_k, q\}$.

To reduce complexity, the calculation can be done in two steps.

$$\delta_k = \sum_{j=N-1-k}^{N-2} R_{(N-1-k,j)} s_j \quad (10)$$

$$w_{k,q}^{<t>} = \|\hat{\mathbf{y}}_{N-t-1} - \delta_k - R_{(N-1,N-1)}q\|_2^2 \quad (11)$$

where s_j is the j^{th} element of a k^{th} subpath h'_k .

Thread q first calculates δ_q and stores the result (an intermediate partial distance vector) into shared memory. We synchronize the threads for the next step. The partial distance vector calculated is used to speed up the Q partial distance calculation for each vertex. This reduces complexity, the original algorithm requires $N \times Q^2$ complex multiplies, while the reduced complexity algorithms only require $N \times Q$ complex multiplies.

Each thread finds the the path with the minimum partial distance using edge reduction. Each thread calculates Q partial distances serially and finds the path with the minimum weight. 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 1.

Algorithm 1 The q^{th} thread searches for the path with lowest cumulative weight

---

```

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

```

VI. PERFORMANCE ANALYSIS

The GPU used is an Nvidia 9600GT graphic card, which has 64 stream processors running at 1900MHz and 512MB of DDR3 memory running at 2000 MHz. The test code first generates the random input symbols and a random channel. After passing the input symbols through this channel, it performs QR-decomposition on the channel matrix $\mathbf{H}$ to generate $\mathbf{R}$ and $\hat{\mathbf{y}}$. Both $\mathbf{R}$ and $\hat{\mathbf{y}}$ are fed into the detection kernel running on GPU.

Various modulation schemes are compared: 4-QAM, 16-QAM, 64-QAM and 256-QAM. The detector kernel detects 300 symbols for each modulation. To keep utilization high, each thread block detects 16 symbols for 4-QAM, 4 symbols for 16-QAM, and 1 symbol for 64-QAM and 256-QAM. The execution time is averaged over 1000 runs. As the GPU is often connected

to the host through the PCI-express bus, transfer data via this bus results in a measurable and non-negligible latency penalty. Table II shows the results for 2×2 and 4×4 MIMO system with and without transport time.

TABLE II: Average Runtime for 2×2 and 4×4 300 subcarriers

Q	Runtime(ms)/Throughput(Mbps)			
	2×2		4×4	
	w. transport	w/o. transport	w. transport	w/o. transport
4	0.089/12.88	0.0195/58.56	0.096/23.83	0.050/46.16
16	0.106/21.54	0.0314/72.84	0.102/44.84	0.061/74.30
64	0.373/9.20	0.201/17.10	0.541/12.68	0.473/14.50
256	3.937/1.16	3.303/1.38	7.95/1.17	7.116/1.29

MIMO-OFDM is used to achieve high data rate in a real time system such as 3GPP LTE and WiMAX. The number of data subcarriers per symbol is 300 for a 5 MHz LTE MIMO system. Since each slot is 0.5ms and consists of 7 MIMO symbols, our detector needs to detect a 300 subcarrier MIMO OFDM symbol in 0.0714 ms to handle maximum throughput for a particular configuration. If we consider peak throughput without the transfer overhead, our detector can handle 4-QAM and 16-QAM for 2×2 and 4×4 5MHz LTE MIMO system. The detector can support other standards such as WiMAX by changing the number of symbols fed into the detector.

The current implementation attempts to maximize efficiency by ensuring each thread block has an integer multiple of 32 threads and employing a regular algorithm with very regular memory access. To measure the efficiency of the mapping, we can use the *warp* instruction count to generate an upper bound limit on execution time (without transport). The number of *warp* instructions executed on a SM to detect this set of MIMO symbols is recorded for each modulation using the CUDA Visual Profiler. The estimated execution time, T_{est} , can be calculated as:

$$T_{est} = C \times CPI \times F_c^{-1} \quad (12)$$

where F_c is clock frequency, C is the number of executed *warp* instructions, and CPI is the average number of cycles per instruction.

In CUDA, the average CPI is 4 cycles per instruction and each SM is clocked at 1900MHz. The estimated runtime is shown in Table III. The estimated detection time is shorter

TABLE III: Estimated Runtime for 4×4 , 300 subcarriers

Modulation	C	T_{est}	Estimated/Actual
4-QAM	5043	0.05	0.21
16-QAM	10808	0.06	0.37
64-QAM	99409	0.47	0.44
256-QAM	1430730	7.12	0.42

than the actual execution time. This is expected since CPI of 1 corresponds to the maximum instruction throughput. For example, a MAD instruction with one shared memory operand achieves only 67% of the maximum instruction throughput [12]. Furthermore, occupancy for all four cases is below 50% due to the high number of registers each thread uses. This affects 4-QAM the most, as the additional stalls due to inefficient constant

memory access pattern can not always be masked through fast thread switching.

A. Compared to ASIC/FPGA/ASIP

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”. The GPU, on the other hand, will never encounter performance loss due to its floating point computation capability.

Table IV compares our GPU design with state-of-the-art ASIC/FPGA/ASIP designs in terms of throughput. In [13], a depth-first search detector with 256 searches per level is implemented. In [3], a K-best detector with $K = 5$ and real decomposition is implemented. In [14], a relaxed K-best detector with $K = 48$ is implemented. In [6], a K-best with $K = 7$ detector is implemented. We also list our early ASIC design [9] 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 solutions from [3, 13, 14], our GPU design can achieve comparable throughput. Compared to ASIP solution from [6], our GPU design has a 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 greatly simplify the fixed-point design issues.

TABLE IV: Throughput comparison with ASIC/FPGA/ASIP

	4x4 QPSK	4x4 16QAM	4x4 64QAM	4x4 256QAM
ASIC [13]	19.2 Mbps	38.4 Mbps	NA	NA
ASIC [3]	NA	53.3 Mbps	NA	NA
FPGA [14]	NA	NA	8.57 Mbps	NA
ASIP [6]	NA	5.3 Mbps	NA	NA
ASIC [9]	300 Mbps	600 Mbps	NA	NA
GPU	46.16 Mbps	74.30Mbps	14.50Mbps	1.29Mbps

VII. CONCLUSION

This paper presents a Trellis MIMO detector implementation using a floating-point GPU. The algorithm was designed to fully utilize the multiple stream processors in GPU. Compared to the conventional fixed-point VLSI implementations, the GPU based MIMO detector has more flexibility in supporting different MIMO system configurations while still maintaining good performance and achieving high throughput. The GPU based MIMO detector implementation proposed in this paper open up a new opportunity for MIMO software defined radio.

VIII. ACKNOWLEDGMENT

The authors would like to thank Nokia, Nokia Siemens Networks (NSN), Xilinx, and US National Science Foundation (under grants CCF-0541363, CNS-0551692, CNS-0619767, and EECS-0925942) for their support of the research.

REFERENCES

- [1] A. Burg, M. Borgmann, M. Wenk, M. Zellweger, W. Fichtner, and H. Bolcskei, “VLSI Implementation of MIMO Detection Using the Sphere Decoding Algorithm,” *IEEE J. Solid-State Circuit*, vol. 40, pp. 1566–1577, July 2005.
- [2] K. Wong, C. Tsui, R. Cheng, and W. Mow, “A VLSI architecture of a K-best lattice decoding algorithm for MIMO channels,” in *IEEE Int. Symp. on Circuits and Syst.*, vol. 3, May 2002, pp. 273–276.
- [3] Z. Guo and P. Nilsson, “Algorithm and implementation of the K-best sphere decoding for MIMO detection,” *IEEE J. Selected Areas in Commun.*, vol. 24, pp. 491–503, Mar 2006.
- [4] X. Huang, C. Liang, and J. Ma, “System Architecture and Implementation of MIMO Sphere Decoders on FPGA,” *IEEE Tran. VLSI*, vol. 2, pp. 188–197, Feb 2008.
- [5] Q. Qi and C. Chakrabarti, “Sphere Decoding for Multiprocessor Architectures,” in *IEEE Workshop on Signal Processing Systems*, Oct. 2007, pp. 17–19.
- [6] J. Antikainen, P. Salmela, O. Silven, M. Juntti, J. Takala, and M. Myllyla, “Application-Specific Instruction Set Processor Implementation of List Sphere Detector,” *EURASIP Journal on Embedded Systems*, 2007.
- [7] J. Janhunen, O. Silven, M. Juntti, and M. Myllyla, “Software defined radio implementation of k-best list sphere detector algorithm,” in *Embedded Computer Systems: Architectures, Modeling, and Simulation, 2008. SAMOS 2008. International Conference on*, July 2008, pp. 100–107.
- [8] G. Falcão, V. Silva, and L. Sousa, “How gpus can outperform asics for fast ldpc decoding,” in *ICS '09: Proceedings of the 23rd international conference on Supercomputing*. New York, NY, USA: ACM, 2009, pp. 390–399.
- [9] Y. Sun and J. R. Cavallaro, “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*. New York, NY, USA: ACM, 2009, pp. 445–450.
- [10] —, “A new mimo detector architecture based on a forward-backward trellis algorithm,” in *IEEE 42nd Asilomar Conference on Signals, Systems and Computers (ASILOMAR'08)*, Oct. 2008.
- [11] NVIDIA Corporation, *CUDA Compute Unified Device Architecture Programming Guide*, 2008. [Online]. Available: http://www.nvidia.com/object/cuda_develop.html
- [12] V. Volkov and J. W. Demmel, “Benchmarking gpus to tune dense linear algebra,” in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–11.
- [13] D. Garrett, L. Davis, S. ten Brink, B. Hochwald, and G. Knagge, “Silicon Complexity for Maximum Likelihood MIMO Detection Using Spherical Decoding,” *IEEE J. Solid-State Circuit*, vol. 39, pp. 1544–1552, Sep 2004.
- [14] S. Chen, T. Zhang, and Y. Xin, “Relaxed K-Best MIMO Signal Detector Design and VLSI Implementation,” *IEEE Tran. VLSI*, vol. 15, pp. 328–337, Mar. 2007.