Reconfigurable Real-time MIMO Detector on GPU

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

***Abstract*—In a high performance multiple-input multiple-output (MIMO) system, a soft output MIMO detector combined with a channel decoder is often used at the receiver to maximize performance gain. Graphic processor unit (GPU) is a low-cost parallel programmable co-processor that can deliver extremely high computation throughput and is well suited for signal processing applications. We propose and implement a novel soft MIMO detection algorithm and show we meet real-time performance while maintaining flexibility using GPU.**

I. Introduction

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 to deliver real-time throughput. For example, researchers have found that GPU, like ASIC, can perform low density parity code (LDPC) decoding as well as ASIC [1]. Combined with the fact that these types of processors are extremely cost-effective and ubiquitous and can be reconfigured on the fly to handle different workloads, communication algorithms in the future can be offloaded onto this type of processor in place of custom ASIC or FPGA.

In many wireless systems, a channel decoder such as LDPC is combined with a soft output MIMO detector at the receiver to maximize performance gain. Besides the channel decoder, the MIMO detector is another computation intensive block. Although an exhaustive search based MIMO detector would be optimal, its complexity would be prohibitive. Fortunately, suboptimal MIMO detectors can provide close to optimal performance with significantly lower complexity. The typical suboptimal MIMO detectors are ASIC designs [2–4]. In addition, researchers have investigated many other ways of hardware implementation such as FPGA [5, 6] and application-specific instruction set processor (ASIP) [7]. To the best of our knowledge, there are no existing implementations of soft MIMO detector on GPU. In this paper, we aim to show that besides these traditional solutions, graphic processor unit (GPU) has become a viable alternative to high performance accelerators for soft MIMO detection.

However, careful architecture-aware algorithm design is needed to achieve high performance on the GPU. For example, due to the limited amount of resources on GPU, such as on-chip memory, many existing algorithms, such as depth first sphere detector and K-best detector, do not map very efficiently onto this architecture. In this paper, we propose a MIMO soft detection algorithm specifically designed for this type of architecture based on multi-pass forward trellis traversal. We also show that this soft MIMO detector implementation can achieve good performance while maintaining flexibility offered by programmable hardware.

II. System Model

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

$$\mathbf{y} = \mathbf{Hs} + \mathbf{w} \tag{1}$$

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 variables with σ^2 variance per dimension. The transmit vector is $\mathbf{s} = [s_0, s_1, ..., s_{N-1}]$, 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 as:

$$\mathbf{y} = \mathbf{QRs} + \mathbf{w} \tag{2}$$
$$\hat{\mathbf{y}} = \mathbf{Rs} + \hat{\mathbf{w}} \tag{3}$$

where $\mathbf{R}$ is a $M \times N$ 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 $s_m = \text{map}(\mathbf{x})$, where $\mathbf{x} = \{x_0, x_1, ..., x_{M_c-1}\}$, a $M_c \times 1$ 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 max-log approximation, LLR can be expressed as [8]:

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

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

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

III. Compute Unified Device Architecture (CUDA)

The raw computation power offered by the programmable GPU is enabled by many cores. Eight cores form a stream multiprocessor (SM). During execution, all cores in a multiprocessor execute the same 32-bit integer or float operation on different sets of data. However, computation throughput can become I/O limited if memory bandwidth is low. Fortunately, 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.

Compute Unified Device Architecture [9] is a software programming model that exposes the massive computation potential offered by the programmable GPU. A kernel, a series of operations applied to a set of data can be defined using this programming model. At runtime, multiple threads are spawned, where each thread runs the operations defined by the kernel on a data set. Threads are independent in this model. However, threads within a block can share computation through barrier synchronization and shared memory. Thread blocks are completely independent and only can be synchronized through writing to the global memory and terminating the kernel. Figure 1 shows the thread hierarchy.

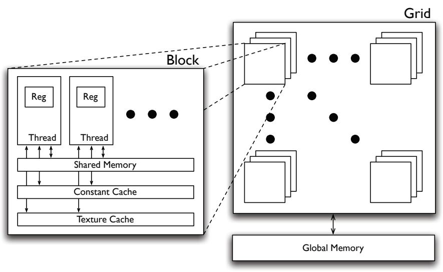


Fig. 1: CUDA thread model.

During execution, each thread block is assigned to an SM. CUDA divides threads within a thread block into blocks of 32 threads. These 32 threads, a WARP, are executed as a group on an SM over 4 cycles. As data is not cached, SM can stall waiting for data. To keep cores utilized, multiple thread blocks (concurrent thread blocks) are mapped onto an SM and executed on an SM at the same time. Since the GPU can switch between WARP instructions with zero-overhead, GPU can minimize stalls by switching over to another independent WARP instruction on a stall.

Besides fast thread switching, shared memory, which can be as fast as a register, can reduce memory access time by keeping data on-chip and reduce redundant calculations by allowing data sharing among independent threads. 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 meet certain conditions to allow the instruction to execute in one cycle. It takes one cycle if all threads access the same port (broadcast) or none of the threads access the same port. However, random layout with some broadcast and some one-to-one accesses will be serialized and cause a stall.

There are several other limitations with shared memory. First, only threads within a block can share data among themselves and threads between blocks can not share data through shared memory. Although a fast block synchronization method is described in [10], the overhead is still large on the order of microseconds. Second, there are only (16KB) on each stream multiprocessor and shared memory is divided among the concurrent thread blocks on a SM. Therefore, the number of concurrent thread blocks on a SM can be small if each thread block uses a large amount of sharde memory. As a result, designing an algorithm that maps efficiently onto GPU is a non-trivial task.

IV. Proposed Soft MIMO Detector

In this section, we propose using a greedy shortest path algorithm developed in our previous work [11] to approximately solve the soft detection problem. Only one kernel is required for candidate list generation. At runtime, the kernel spawns a large number of soft MIMO detector thread blocks, one thread block for each channel matrix and the corresponding receive vector. Each thread block uses Q threads to generate a candidate list for each trellis level and calculate 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. We choose this algorithm because it maps efficiently onto GPU. First, there are a fair amount of common computations across threads in a thread-block. Second, memory access is fast since this algorithm has a regular memory access pattern.

We use a 4×4 QPSK system to explain our proposed algorithm in this section. The search space becomes larger for larger systems with more antennas and higher modulation. Therefore, we can extend Figure 2 by adding one trellis stage per antenna and one trellis level per constellation point. However, we can still apply the same greedy shortest path algorithm to solve the soft detection problem.

A. Graph construction

The goal of the soft MIMO detector is to generate the LLR value for each transmitted bit x_k based on (4), which requires the calculation of the minimum Euclidean distance

$$\Lambda = \left\| \begin{bmatrix} \hat{y}_0 \\ \hat{y}_1 \\ \hat{y}_2 \\ \hat{y}_3 \end{bmatrix} - \begin{bmatrix} R_{00} & R_{01} & R_{02} & R_{03} \\ 0 & R_{11} & R_{12} & R_{13} \\ 0 & 0 & R_{22} & R_{23} \\ 0 & 0 & 0 & R_{33} \end{bmatrix} \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix} \right\|^2, \quad (6)$$

over sets $\mathbb{X}_{k,+1}$ and $\mathbb{X}_{k,-1}$. The calculation of Λ can be decomposed as: $\Lambda = w^{<0>} + w^{<1>} + w^{<2>} + w^{<3>}$, where $w^{<t>}$ is the 1-D Euclidean distance and is calculated as

$$\begin{aligned} w^{<0>} &= \|\hat{y}_3 - R_{33}s_3\|^2, \\ w^{<1>} &= \|\hat{y}_2 - (R_{22}s_2 + R_{23}s_3)\|^2, \\ w^{<2>} &= \|\hat{y}_1 - (R_{11}s_1 + R_{12}s_2 + R_{13}s_3)\|^2, \\ w^{<3>} &= \|\hat{y}_0 - (R_{00}s_0 + R_{01}s_1 + R_{02}s_2 + R_{03}s_3)\|^2. \quad (7) \end{aligned}$$

This process can be viewed using a flow graph which is shown in Figure 2. There are 4 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 does not depend on the future stages, but only depends on its current stage and all its predecessors. For example, $w_{i,j}^{<2>}$ depends on the vertices in stages 2, 1, and 0.

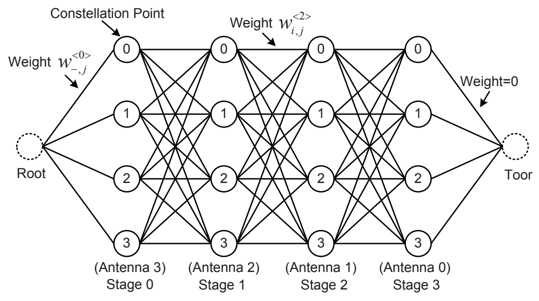


Fig. 2: Flow graph for MIMO detection.

B. Problem Statement

To generate 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 as:

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

C. Candidate List Generation

To generate the candidate lists, the algorithm searches for a shortest path through the trellis for each vertex i in our trellis graph. The algorithm does this by pruning unlikely paths through the trellis. There are two ways of reducing the number of paths. We can either prune the incoming paths or outgoing paths at each vertex. Edge reduction reduces the number of paths by pruning the number of incoming paths for a vertex to one. Similarly, path extension reduces the number of paths by pruning the number of outgoing paths for vertex to one. Figure 3 shows an edge reduction and a path extension.

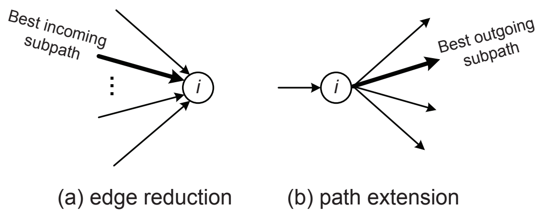
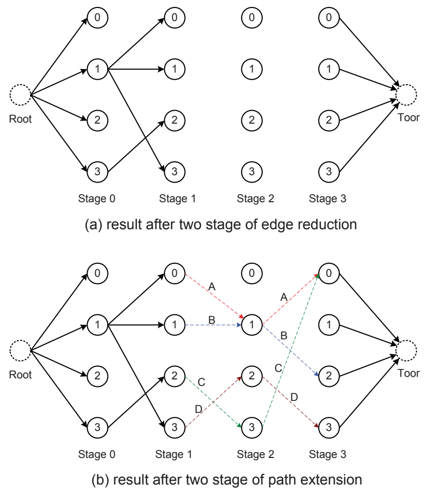


Fig. 3: Data flow at vertex $v(t,i)$

The candidate list search process can be expressed as edge reductions followed by path extensions. To generate the candidate list for $\mathcal{L}_t$, we perform edge reduction until there is one path per trellis stage at level t. If we perform edge reduction after this level, we can not guarantee each path in candidate list has a vertex from trellis level t. Therefore, after this trellis level t, we perform path extension until we have completely traversed the trellis. Figure 4 shows each stage of the search process for $\mathcal{L}_1$. The complete search process can be represented with a data flow diagram, shown by Figure 5. There are common steps when generating candidate lists for each trellis level. For



Root
Toor
Stage 0
Stage 1
Stage 2
Stage 3
(a) result after two stage of edge reduction
A
B
C
D
(b) result after two stage of path extension

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

example, all search processes starts with a path reduction at stage 0. Furthermore, since each reduction step and the edge reduction directly above both prune the edges between stage i and stage $i{+}1$ and have the same set of incoming subpaths, both steps need the same Q^2 weights. Computation can be reduced by allowing these two steps to share computations.

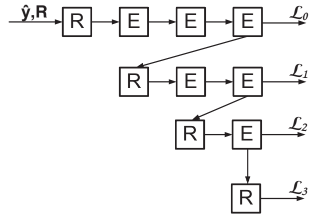


Fig. 5: Data-flow diagram for generating candidate lists.

We will now describe the algorithm and the software implementation of extension and reduction steps.

1) Path Extension: Figure 3 shows that each vertex i at stage t has Q outgoing subpaths. Since we have Q vertices to extend, we use all Q threads, one thread per vertex. Specifically, thread k evaluates all Q outgoing paths and picks the path with the smallest path weight for vertex k. An outgoing path's edge weight between vertex k (in stage $t-1$) and vertex q (in stage t) can be expressed as

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

where h'_k is the k^{th} subpath and s_j is the j^{th} element of $\{h'_k, q\}$. The k^{th} outgoing path weight is then updated as

$$d_k = d'_k + w_{k,q}^{<t>}. \qquad (10)$$

Algorithm 1 summarizes steps taken to find the path with the smallest path weight. Line 2 calculates δ_k,

$$\delta_k = \sum_{j=N-1-k}^{N-2} R_{(N-1-k,j)} s_j, \qquad (11)$$

where s_j is the j^{th} element of a k^{th} subpath h'_k. Lines 4-17 evaluate Q outgoing paths by evaluating all constellation points in our complex constellation alphabet Ω. Line 12 first computes edge weight $w_{k,q}^{<t>}$ and line 13 computes the updated path weight, d_k. Lines 14-17 search outgoing paths with the smallest cumulative weight serially. The path selected is the new k^{th} path.

Algorithm 1 The k^{th} 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
```

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.

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

2) Edge Reduction: Figure 3 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. 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 can also be computed using equation (9).

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

$$\delta_k = \sum_{j=N-1-k}^{N-2} R_{(N-1-k,j)} s_j, \qquad (12)$$

$$w_{k,q}^{<t>} = \left\| \hat{y}_{N-t-1} - \delta_k - R_{(N-1,N-1)} q \right\|_2^2, \qquad (13)$$

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

Notice that the extension step above each reduction step already computed all δ_k, which reduces complexity significantly. 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 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.

Algorithm 2 The q^{th} 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
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
```

D. 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. Although we can terminate our thread blocks and spawn $log_2(Q)$ threads to perform LLR computation, the overhead to terminate a kernel is large. Furthermore, the complexity of LLR computation is smaller than the reduction and the extension step. Therefore, we propose a simple linear search. In this search, thread k is responsible for bit k, where $k < log_2(Q)$. This method is inefficient as only $log_2(Q)$ threads are making useful computations. However, each thread does computation independently and does not require any synchronization.

The steps are summarized in Algorithm 3: The input to the LLR computation, the candidate lists, are the Q path weights. The code block in lines 4-10 searches for two smallest weights in a linear fashion. Lines 5-6 search for the minimal weight where k^{th} bit is 0 and lines 7-8 search for the minimal weight where k^{th} bit is 1. Line 11 computes the difference between the two minimums, which is equal to the LLR.

Algorithm 3 The kth thread compute the k^{th} LLR

```
1: $m_0 = 999$
2: $m_1 = 999$
3: if $k < log_2(Q)$ then
4:     for $j = 0$ to $Q-1$ do
5:         if $k^{th}$ bit of $j$ is 0 and $m_0 > d_j$ then
6:             $m_0 = d_j$
7:         else if $k^{th}$ bit of $j$ is 1 and $m_1 > d_j$ then
8:             $m_1 = d_j$
9:         end if
10:     end for
11:     $LLR_k = \frac{(m_0 - m_1)}{\sigma^2}$
12: end if
13: SYNC
```

V. Simulation Results

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. Figure 6 compares the BER performance of the proposed greedy algorithm with the traditional K-best algorithm. In this simulation, the soft-output of the GPU detector is fed to a length 2304, rate 1/2 WiMax LDPC decoder [12] running on CPU, which performs up to 15 iterations. The simulation shows that this detector performs as well as K-best detector with large K.

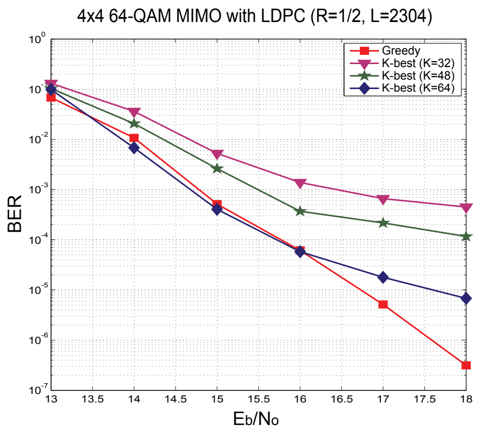


Fig. 6: BER performance comparison

We configured our detector for 3GPP LTE. 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 OFDM symbols, our detector needs to detect 2100 subcarriers in 0.5 ms to handle maximum throughput for this particular configuration. Various modulation schemes are compared: 4-QAM, 16-QAM, 64-QAM. 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 I shows the results for 2×2 and 4×4 MIMO systems with and without transport time.

TABLE I: Average Runtime for 2×2 and 4×4 2100 subcarriers

	Runtime(ms)/Throughput(Mbps)			
	2×2		4×4	
Q	w. transport	w/o. transport	w. transport	w/o. transport
4	0.089/62.67	0.047/180.10	0.81/20.54	0.31/54.19
16	0.40/41.52	0.27/63.05	2.19/15.35	1.19/28.17
64	3.05/8.268074	2.86/8.80407	13.09/3.85	11.91/4.23

Once factoring in the transfer overhead, the proposed detector can handle 4-QAM and 16-QAM for a 2×2 MIMO 5 MHz LTE system. Without factoring in the transfer overhead, the proposed detector can also handle 4×4 MIMO 5 MHz LTE system. Larger configurations can be achieved using larger devices. The detector can support other standards such as WiMAX by changing the number of symbols fed into the detector.

VI. Conclusion

This paper presents a soft 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 achieving high throughput that can meet LTE performance requirements. The GPU based MIMO detector proposed in this paper opens up a new opportunity for MIMO software defined radio.

VII. Acknowledgments

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] 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*, pp. 390–399.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] K. Amiri, C. Dick, R. Rao and J. R. Cavallaro, "A High Throughput Configurable SDR Detector for Multi-user MIMO Wireless Systems," *Springer Journal of Signal Processing Systems*, 2009.

[7] 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.

[8] B. Hochwald and S. Brink, "Achieving Near-Capacity on a Multiple-Antenna Channel," *IEEE Tran. Commun.*, vol. 51, pp. 389–399, Mar 2003.

[9] *NVIDIA Corporation, CUDA Compute Unified Device Architecture Programming Guide*, 2008. [Online]. Available: http://www.nvidia.com/object/cuda_develop.html

[10] 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*, 2008, pp. 1–11.

[11] 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*, 2009, pp. 445–450.

[12] ——, "A Low-power 1-Gbps Reconfigurable LDPC Decoder Design for Multiple 4G Wireless Standards," in *IEEE International SOC Conference*, Spet. 2008, pp. 367–370.