RICE UNIVERSITY

On the Application of Graphics Processor to Wireless Receiver Design

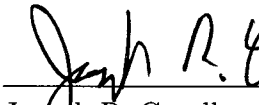
by

Michael Wu

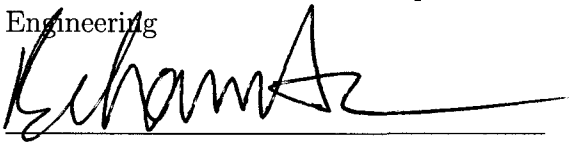
A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

MASTER OF SCIENCE

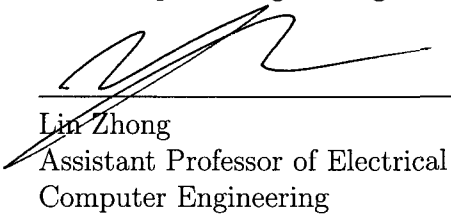
Approved, Thesis Committee:



Joseph R. Cavallaro, Chair
Professor of Electrical and Computer Engineering



Behnaam Aazhang
J.S. Abercrombie Professor of Electrical and Computer Engineering



Lin Zhong
Assistant Professor of Electrical and Computer Engineering

Houston, Texas

April, 2010

ABSTRACT

On the Application of Graphics Processor to Wireless Receiver Design

by

Michael Wu

In many wireless systems, a Turbo decoder is often combined with a soft-output multiple-input and multiple-output (MIMO) detector at the receiver to maximize performance in many 4G and beyond wireless standards. Although custom application specific designs are usually used to meet this challenge, programmable graphics processing units (GPU) has become an alternative to the traditional ASIC and FPGA solution for wireless applications. However, careful architecture-aware algorithm design and mapping are required to maximize performance of a communication block on GPU. For MIMO soft detection, we implemented a new MIMO soft detection algorithm, multi-pass trellis traversal (MTT). For Turbo decoding, we used a parallel window algorithm. We showed that our implementations can achieve high throughput while maintaining good performance. This work will allow us to implement a complete iterative MIMO receiver in software on GPU in the future.

Acknowledgments

I would like to thank my advisor, Professor Joseph R. Cavallaro, for his thoughtful comments and support for the last three years. I would also like to thank other members of my committee, Professor Behnaam Aazhang and Professor Lin Zhong for their constructive comments.

I would also like to thank my family. First, to my parents, Youfeng and Jane, I could not have accomplished this without your support. Second, to my sisters, Susan and Catherine, for being supportive and helpful as always.

Last but not least, I would like to thank Sid Gupta, Kia Amiri, Guohui Wang, and Bei Yin for their useful feedback and comments.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

In many wireless systems, a channel decoder such as Turbo codes is combined with a soft output multiple-input and multiple-output (MIMO) detector at the receiver to maximize performance gain. The combination of MIMO and Turbo decoder is used in many 4G and beyond wireless standards such as IEEE 802.16e WiMax, and 3GPP LTE (long term evolution). Although the combination of MIMO detector and Turbo decoder improves performance of a MIMO system dramatically, both MIMO detector and Turbo decoder are very computation intensive blocks. In the case of MIMO detector, as an exhaustive search based MIMO detector's complexity would be prohibitive, implementations are suboptimal MIMO detectors which can provide close to optimal performance with significantly lower complexity. Nevertheless, typical suboptimal MIMO detectors are ASIC designs [1, 2, 3], other implementations include FPGAs [4, 5] and application-specific instruction set processors (ASIPs) [6]. In the case of Turbo decoder, the inherently large decoding latency and a complex iterative decoding algorithm have made it very difficult to achieve high throughput in general purpose processors or digital signal processors. As a result, Turbo decoders are often implemented in ASIC or FPGA [7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18].

To accelerate 3D visual effects that are moving toward photorealism, programmable

graphics processing units (GPU) delivers extremely high computation throughput by employing many cores working a large set of data in parallel. The GPU as an alternative to the traditional ASIC and FPGA solution for wireless applications is attractive for several reasons. As GPUs becomes increasingly more flexible, it can handle general purpose computations and can accelerate other tasks beyond the realm of graphics. Communication algorithms typically are very parallel and can take advantage of the inherently parallel structure of the GPU. For example, researchers have found GPUs can perform low density parity code (LDPC) decoding as well as ASICs [19]. Furthermore, an implementation on GPU can be reconfigured easily to handle different workloads as it is done completely in software. Finally these type 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 ASICs or FPGAs.

However, the underlaying hardware of GPU is fixed. Careful architecture-aware algorithm design and mapping are required to achieve good performance. Much of the mapping and optimizations are left to the programmer. For example, the programmer needs to specify how to use the limited amount of resources on GPU, such as on-chip shared memory. In addition, the programmer needs to specify how computation is partitioned on GPU by partitioning threads among the cores to handle the workload. An implementation that scales well, while keeping the cores fully utilized to achieve

peak throughput across different configurations, is a difficult to achieve.

1.1 Contribution

Our goal is to design and implement MIMO detection and Turbo detector *efficiently* on GPU to maximize performance. To the best of our knowledge, there are no existing implementations of soft MIMO detector or Turbo decoder on GPU. For MIMO soft detection, we implement a new 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 MMT against K-best and a suboptimal one-pass trellis traversal implementation. Furthermore, 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. For Turbo decoder, our implementation partitions the decoding workload across cores and prefetch data to reduce memory stalls. As parallelization of the decoding algorithm can improve throughput of a decoder at the expense of decoder performance, we provide both throughput and performance of the decoder and show that we can parallelize the workload on GPU while maintaining reasonable performance. This work will allow us to implement a complete iterative MIMO receiver in software, which includes both MIMO detector and Turbo decoder on GPU in the future.

1.2 Thesis Overview

The organization of the thesis is as follows. Chapter 2 give an overview of the CUDA architecture. Chapter 3 gives an overview of the MIMO detection and Turbo decoding algorithms. Chapter 4 discusses the implementation of MIMO detector on GPU, as well as the performance results and analyses. Chapter 5 discusses the implementation of Turbo detector on GPU, as well as the performance results and analysis. Finally, we will conclude in chapter 6.

Chapter 2
Overview of CUDA

2.1 Compute Unified Device Architecture

Compute Unified Device Architecture [20] is a software programming model that allows the programmer to harness the massive computation potential offered by the programmable GPU. The programming model is explicitly parallel. The programmer explicitly specifies the parallelism, i.e. how operations are applied to a set of data, in a kernel. At runtime, multiple threads are spawned, where each thread can select a set of data using its own unique ID and runs the operations defined by the kernel on the data set. Each thread block contains multiple threads, up to a 512 threads per thread block. In this programming model, threads are completely independent. 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.

Compared to traditional general purpose processors, programmable GPU has much higher peak computation throughput. The overall architecture is shown in figure 2.1. The computation power is enabled by many stream multiprocessors (SM) on the GPU, where each SM is 8 ALU single instruction multiple data (SIMD) core. During runtime, a kernel is mapped onto the device by mapping each thread block to an SM. Threads within a thread block are divided into blocks of 32 threads. If

all 32 threads are doing the same set of operation, these 32 threads, also known as a WARP, are executed as a group on an SM over 4 cycles. Otherwise, threads are executed serially. There are a number of reasons for stalls to occur. As data is not cached, SM can stall waiting for data. Furthermore, floating point pipeline is long and register to register dependency can cause a stall in the pipeline. To keep cores utilized, multiple thread blocks, or 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.

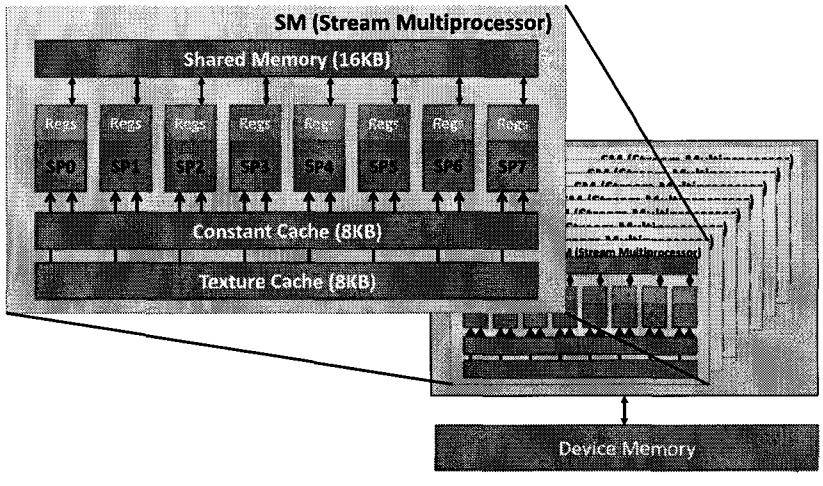


Figure 2.1 CUDA architecture model

Computation throughput can still 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 device memory to keep the computation

throughput high. Shared memory is especially useful. It 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. It takes one cycle if 16 consecutive threads access the same port (broadcast) or none of the threads access the same port (one to one). 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 block can share data among themselves and threads between blocks can not share data through shared memory. Second, there are only (16KB) of shared memory on each stream multiprocessor and share memory is divided among the concurrent thread blocks on a SM. Using too much shared memory can reduce the number of concurrent thread blocks mapped onto a SM.

Table 2.1 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

There are several other limitations with shared memory. First, only threads within

block can share data among themselves and threads between blocks can not share data through share memory. Second, there are only (16 KB) of shared memory on each stream multiprocessor and share 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 partition shared memory, has an efficient memory access pattern, does not require synchronization between blocks and needs few global memory access is non-trivial task.

2.2 Alternative Platforms

There are other many-core architectures other than CUDA. We investigate CUDA as a software wireless receiver platform for several reasons. As GPU is a high volume consumer device, an efficient software wireless receiver will lead to extremely cost-effective accelerator, which can accelerate software defined radio platforms and simulations. In addition, compared to alternative platforms, Nvidia provides a mature development platform, which includes a robust set of tools and good documentation. For example, Nvidia provides documentation for the user to interface the software written in CUDA to Matlab. Furthermore, the software blocks described in this thesis could be ported to other many-core architectures as they are similar to CUDA. We will now describe several other many-core architectures.

Imagine stream processor [21, 22], in particular, shares many similarities to a stream multiprocessors on CUDA. Like CUDA, Imagine stream processor exploits data level parallelism (DLP) by providing several ALU lanes that share the same instruction stream. Unlike CUDA, Imagine stream processor exploits instruction level parallelism (ILP) by using VLIW instructions. Each ALU lane, in this case, has multiple parallel ALUs controlled by an VLIW instruction. Imagine stream processor also has a multi-level memory subsystem that tries to reduce memory I/O bandwidth to external device memory. There is a set of registers, Local Registers File (LSR), per ALU lane. There is also a SRAM block, Stream Register File (SRF), which serves as the intermediate buffer between external device memory and LSR. Furthermore, there is a communication network which allows the lanes to exchange data. In CUDA, registers function both as LSR and SRF, serve as the intermediate buffer between external device memory and ALUs. However, registers on CUDA are still local to each ALU. Therefore, there is a dedicated shared memory which allows the ALUs to shared data by reading and writing to shared memory. Finally, the Imagine Stream Processor has more memory on-die than a CUDA stream multiprocessor. However, CUDA exploits task level parallelism (TLP) by supporting fast thread-switching between different instruction streams. By interleaving computation from different tasks, CUDA can hide potential stalls. Finally, CUDA supports branch, which make it more flexible than Imagine stream processor.

AMD [23] also provides graphics processor units for general purpose computing. CUDA and AMD graphics processors are very similar–both are essentially many SIMD cores on the same die. The overall architecture is very similar with different terminology and slight different hardware configuration. For example, a wavefront is the same as WARP. However, a wavefront consists of 64 threads. A group is the same as a thread-block. Global data share has the same functionality as shared memory. However, shared registers can be used to shared data between wavefronts on an AMD graphics processor. The biggest difference between these two processors how ALUs are arranged. First, there are more ALU lanes per core on AMD graphics processor. Like Imagine stream processor, each ALU lane on AMD graphics processor is VLIW–up to 5 ALU instructions can be issued per VLIW instruction. Although the instruction issue width is wider for AMD graphics processors, this adds another level of complexity. The compiler will try to extract ILP by vectorizing a thread's instructions. However, the programmer need to vectorize code explicitly to maximize performance [23]. Nevertheless, receiver algorithms described can be implemented on this platform, although code need to be optimized for this hardware's specific parameters to maximize performance.

A more general many-core model is multiple instruction multiple data (MIMD). In this model, each core is completely independent and does not need to share the same set of instructions. There are several examples of upcoming MIMD proces-

sors, for example, Tilera Tile-Gx, Intel 48-core single-core cloud computer, Larrabee, and Siliconhive HiveFlex Processors. The overall architecture of these processors is similar–many cores with local data cache connected by network. The specific architecture for each core is different for each architecture. For example, Larrabee is arranged into SIMD clusters while HiveFlex is arranged into VLIW clusters. Similarly, the specific arrangement of local data cache and interconnect differs between these processors. However, unlike Graphic processors, where each thread-block is independent and can be scheduled on any SIMD core by hardware, the programmer may need to distribute the work and balance the workload across on MIMD processors as cores can be executing different set of instructions. Nevertheless, a MIMD model can be used to pipeline and/or implement different communication blocks on the same die. Hence, these upcoming platforms that may lead to even more flexible software receivers in the future.

Chapter 3
MIMO Detection and Turbo Decoder

In section 3.1 of this chapter, we will first describe the soft-decision MIMO detection problem. We will give a brief summary of different MIMO detection algorithms and describe a trellis MIMO detection algorithm which we implemented on GPU. In section 3.3 of this chapter, we will describe the Turbo decoding algorithm and then describe parallel Turbo decoding techniques.

3.1 MIMO Detection

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:

$$\mathbf{y} = \mathbf{Hs} + \mathbf{w} \tag{3.1}$$

where $\mathbf{y} = [\mathbf{y_0}, \mathbf{y_1}, ..., \mathbf{y_{M-1}}]^\mathbf{T}$ 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 variables with unit variance. Noise at the receiver is $\mathbf{w} = [\mathbf{w_0}, \mathbf{w_1}, ...\mathbf{w_{N-1}}]^\mathbf{T}$, where w_i is an independent zero mean circularly symmetric complex Gaussian random variables with σ^2 variance per dimension. The transmit vector is $\mathbf{s} = [\mathbf{s_0}, \mathbf{s_1}, ..., \mathbf{s_{M-1}}]$, 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:

$$\mathbf{y} = \mathbf{QRs} + \mathbf{w} \tag{3.2}$$

$$\hat{\mathbf{y}} = \mathbf{Rs} + \hat{\mathbf{w}} \tag{3.3}$$

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 $\mathbf{x} = \{\mathbf{x_0}, \mathbf{x_1}, ..., \mathbf{x_{M_c-1}}\}$, 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 decision MIMO detector calculates the *a posteriori* probability (APP) in term of log likelihood ratio (LLR) for each transmitted bit, x_k. Assuming no extrinsic probability, using max-log approximation, LLR can be expressed as [24]:

$$L(x_k|\hat{\mathbf{y}}) \approx \frac{\mathbf{1}}{\mathbf{2}\sigma^{\mathbf{2}}} \left(\min_{\mathbf{x} \in \mathbf{X_{k,-1}}} \mathbf{\Lambda}(\mathbf{s}, \mathbf{y}) - \min_{\mathbf{x} \in \mathbf{X_{k,+1}}} \mathbf{\Lambda}(\mathbf{s}, \mathbf{y}) \right), \tag{3.4}$$

where the set $X_{k,+1} = \{x|x_k = +1\}$ and set $X_{k,-1} = \{x|x_k = -1\}$ and

$$\Lambda(\mathbf{s}, \hat{\mathbf{y}}) = \|\hat{\mathbf{y}} - \mathbf{Rs}\|_{\mathbf{2}}^{\mathbf{2}} \tag{3.5}$$

One way we can solve this problem is through exhaustive search. For a 4×4 MIMO point to point link, if the transmitter is utilizing 16 QAM,the total number

of possible transmit vectors is $16^4 = 66534$.If the transmitter is utilizing 64QAM,the total number of possible transmit vectors is $64^4 = 16777216$. In either case,searching through all possible transmit vectors is an time intensive process. To reduce complexity, there are two algorithms of searching through find list of candidates, the set of $X_{k,+1}$ and $X_{k,-1}$, for the soft decision MIMO detector, K-best MIMO detection[25] and depth-first sphere detection[2]. In both cases, the algorithms view the search space, the set of all possible transmit vectors, as a tree. K-best detection algorithm is a breath-first tree search algorithm, a greedy MIMO detection algorithm. It reduces the number of candidate we search through in the detection process by detecting input symbols antenna by antenna, keeping at most k-vectors per level. There are quite few drawback for this algorithm. An initial implementation we used K-best detection algorithm, we found sorting takes up to 70% of the run time and requires many reads from and writes to memory. 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. We can parallelize the workload by splitting up the tree. However, the runtime of the tree paritions will be not deterministic and performance of the detector will be bound by the parition that takes longest to complete. As such, we search for an alternative

search algorithm–an sort-free algorithm that is very data parallel.

3.2 Multi-pass trellis traversal MIMO Detection

Multi-pass trellis traversal MIMO detector is first proposed by Yang Sun for custom ASIC design [26, 27]. The algorithm is well suited for GPU architecture for several reasons. The algorithm is very regular, data parallel and completely sort-free. In addition, the multi-pass trellis traversal MIMO detector improves the reliability of the LLR values to improve the performance of the detector.

Without loss of generality, we will now use a 3×3 QPSK system to explain our proposed algorithm in this section.

MIMO Trellis

To generate LLR value for each transmitted bit x_k based on (3.4), the soft MIMO detector needs to compute the minimum Euclidean distance

$$\Lambda = \left\| \begin{bmatrix} \hat{y}_0 \\ \hat{y}_1 \\ \hat{y}_2 \end{bmatrix} - \begin{bmatrix} R_{00} & R_{01} & R_{02} \\ 0 & R_{11} & R_{12} \\ 0 & 0 & R_{22} \end{bmatrix} \begin{bmatrix} s_0 \\ s_1 \\ s_2 \end{bmatrix} \right\|^2, \tag{3.6}$$

over sets $\{X_{k,+1}$ and $\{X_{k,-1}$. 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

$$\begin{aligned} w^{<0>} &= \|\hat{y}_2 - R_{22}s_2\|^2, \\ w^{<1>} &= \|\hat{y}_1 - (R_{11}s_1 + R_{12}s_2)\|^2, \\ w^{<2>} &= \|\hat{y}_0 - (R_{00}s_0 + R_{01}s_1 + R_{02}s_2)\|^2. \end{aligned} \tag{3.7}$$

This process can be illustrated using a MIMO flow graph as shown in Figure 3.1. 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 predecessors. For example, $w_{i,j}^{<2>}$ depends on the vertices in stages 2, 1, and 0.

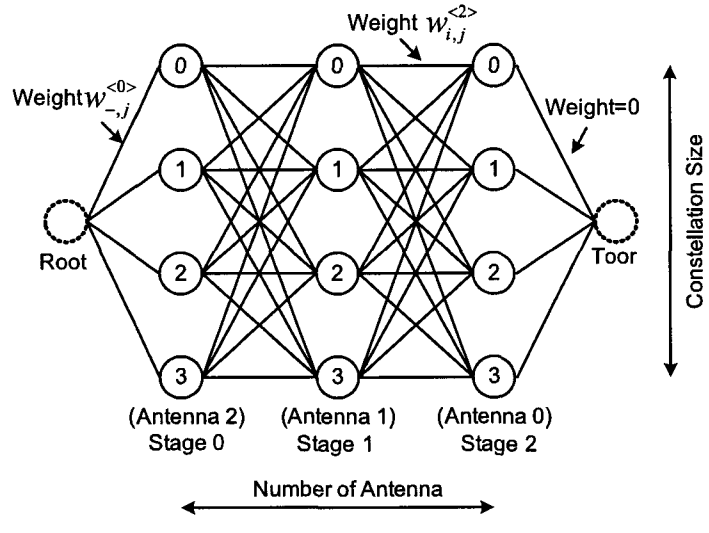


Figure 3.1 MIMO detection flow graph

3.2.1 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 $(0 \leq i \leq Q-1)$ in the stage t $(0 \leq t \leq M-1)$, 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 $\mathcal{L}_{\sqcup}$. We then use the lists to compute the LLR for each bit in a straight forward manner

$$L(x_i^{<t>}|\mathbf{y}) = \frac{1}{2\sigma^2}\left(\min_{\mathbf{x}\in\mathcal{L}_{\sqcup,-\infty}} \mathbf{\Lambda} - \min_{\mathbf{x}\in\mathcal{L}_{\sqcup,+\infty}} \mathbf{\Lambda}\right). \tag{3.8}$$

3.2.2 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.

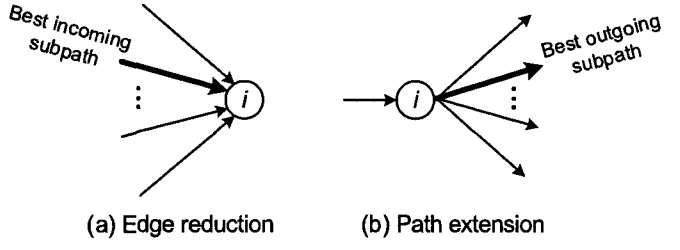


Figure 3.2 Data flow at vertex $v(t,i)$

Edge Reduction

Edge reduction reduces the number of paths by pruning incoming paths. Figure 3.2(a) 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 vertex i. Among the Q incoming subpaths, we select the best subpath h_m with the the smallest partial distance.

$$m = \underset{m \in \{0,...,Q-1\}}{\operatorname{argmin}} d_m, \tag{3.9}$$

and discard the other $Q - 1$ subpaths.

Path Extension

Given one incoming path and multiple outgoing paths, path extension reduces the number of path by pruning outgoing paths. Figure 3.2(b) 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

$$d'_k = d_m + w_{i,k}^{<t+1>}, \ 0 \leq k \leq Q - 1, \tag{3.10}$$

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

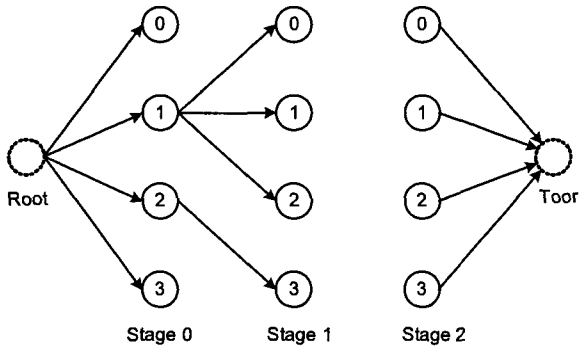
$$n = \underset{n \in \{0,...,Q-1\}}{\operatorname{argmin}} d'_n. \tag{3.11}$$

Shortest Path Algorithm

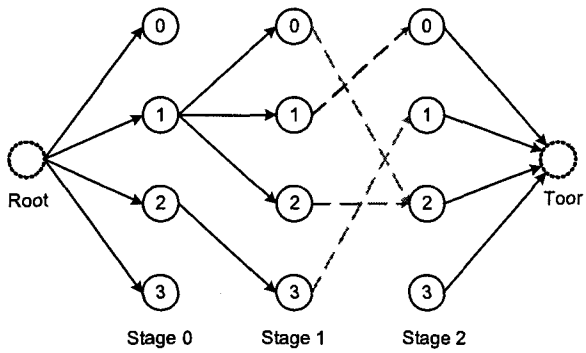
The goal is finding the shortest path through the trellis for each node i. The search process can be expressed as edge reductions followed by path extensions. To generate the candidate list for $\mathcal{L}_{\sqcup}$, we perform edge reductions until there is one path per trellis stage at level t. If we perform edge reductions 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 extensions until we have completely traversed the trellis. Figure 3.3 shows each stage of the search process for $\mathcal{L}_{\infty}$. We do two rounds of edge reduction followed by one round of path extension. There are common steps when generating candidate lists for each trellis level. For example, all search processes starts with a path reduction at stage 0. The search processes can be represent with a data flow diagram, shown by figure 3.4.

3.3 Overview of Turbo Decoder

The principle of Turbo decoding is based on the BCJR or MAP (maximum *a posteriori*) algorithms [28]. The structure of a MAP decoder is shown in Figure 3.5. One iteration of the decoding process consists of one pass through both decoders. Although both decoders perform the same set of computations, the two decoders have different inputs. The inputs of the first decoder are the deinterleaved extrinsic log-likelihood ratios (LLRs) from the second decoder and the input LLRs from the



(a) Result after two stages of edge reduction



(b) Result after one stage of path extension

Figure 3.3 Search process for generating $\mathcal{L}_\infty$.

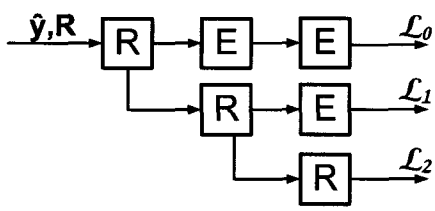


Figure 3.4 Data-flow diagram for generating candidate lists

channel. The inputs of the second decoder are the interleaved extrinsic LLRs from the first decoder and the input LLRs from the channel.

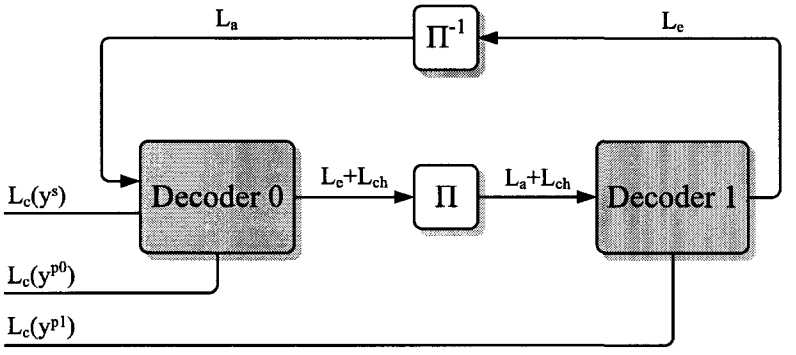


Figure 3.5 Overview of Turbo decoding

To decode a codeword with N information bits, each decoder performs a forward traversal followed by a backward traversal through an N-stage trellis to compute an extrinsic LLR for each bit. The trellis structure, or the connections between two stages of the trellis, is defined by the encoder. Figure 3.6 shows the trellis structure for the 3GPP LTE Turbo code, where each state has two incoming paths, one path for $u_b = 0$ and one path for $u_b = 1$. Let s_k be a state at stage k, the branch metric (or transition probability) is defined as:

$$\gamma_k(s_{k-1}, s_k) = (L_c(y_k^s) + L_a(y_k^s))u_k + L_c(y_k^p)p_k \tag{3.12}$$

where u_k, the information bit, and p_k, the parity bit, are dependent on the path taken (s_{k+1}, s_k). $L_c(y_k^s)$ is the systematic channel LLR, $L_a(y_k^s)$ is the a-priori LLR, and $L_c(y_k^p)$ is the parity bit channel LLR at stage k.

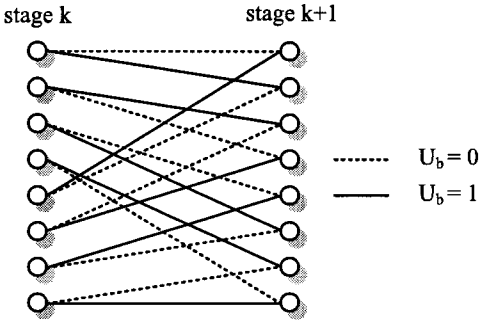


Figure 3.6 3GPP LTE Turbo code trellis with 8 states

The decoder first performs a forward traversal to compute α_k, the forward state metrics for the trellis state in stage k. The state metrics α_k are computed recursively as the computation depends on α_{k-1}. The forward state metric for a state s_k at stage k, $\alpha_k(s_k)$, is defined as:

$$\alpha_k(s_k) = \max^*_{s_{k-1} \in K} (\alpha_{k-1}(s_{k-1}) + \gamma(s_{k-1}, s_k)) \tag{3.13}$$

where K is the set of paths that connect a state in stage $k-1$ to state s_k in stage k.

After the decoder performs a forward traversal, the decoder performs a backward traversal to compute β_k, the backward state metrics for the trellis state in stage k. The backward state metric for state s_k at stage k, $\beta_k(s_k)$, is defined as:

$$\beta_k(s_k) = \max^*_{s_{k+1} \in K} (\beta_{k+1}(s_{k+1}) + \gamma(s_{k+1}, s_k)) \tag{3.14}$$

Although the computation is the same as the computation for α_k, the state transitions are different. In this case, K is the set of paths that connect a state in stage $k+1$ to state s_k in stage k.

After computing β_k, the state metrics for all states in stage k, we compute two LLRs per trellis state. We compute one state LLR per state s_k, $\Lambda(s_k|u_k = 0)$, for the incoming path that is connected to state s_k which corresponds to $u_k = 0$. In addition, we also compute one state LLR per state s_k, $\Lambda(s_k|u_b = 1)$, for the incoming path that is connected to state s_k which corresponds to $u_k = 1$. The state LLR, $\Lambda(s_k|u_b = 0)$, is defined as:

$$\Lambda(s_k|u_b = 0) = a_{k-1}(s_{k-1}) + \gamma(s_{k-1}, s_k) + \beta_k(s_k) \tag{3.15}$$

where the path from s_{k-1} to s_k with $u_b = 0$ is used in the computation. Similarly, the state LLR, $\Lambda(s_k|u_b = 1)$, is defined as:

$$\Lambda(s_k|u_b = 1) = a_{k-1}(s_{k-1}) + \gamma(s_{k-1}, s_k) + \beta_k(s_k) \tag{3.16}$$

where the path from s_{k-1} to s_k with $u_b = 1$ is used in the computation.

To compute the extrinsic LLR for u_k, we perform the following computation:

$$\begin{aligned} L_e(k) = \quad & \max^*_{s_k \in K}(\Lambda(s_k|u_b = 0) - \Lambda(s_k|u_b = 1)) \\ & - L_a(y_k^s) - L_c(y_k^s) \end{aligned} \tag{3.17}$$

where K is the set of all possible states and max*() is defined as $\max^*(S) = ln(\sum_{s \in S} e^s)$.

Chapter 4
MIMO Detector on GPU

4.1 Proposed Implementation on GPU

A single kernel generates the candidate lists and computes LLRs for a large number of problems 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 operates 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 com-

putation. This section is less parallel than path reduction and path extension. This method does not require terminating a kernel, and reading and writing from slower global memory.

We attempt to keep our detector operating at peak utilization by minimizing stall time. Penalty to due to memory access is low since this algorithm has a regular memory access pattern. 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 the detector by reducing the number of instructions required to perform MIMO detection through sharing computation across threads within a thread block. We unroll loops when possible to reduce instruction count.

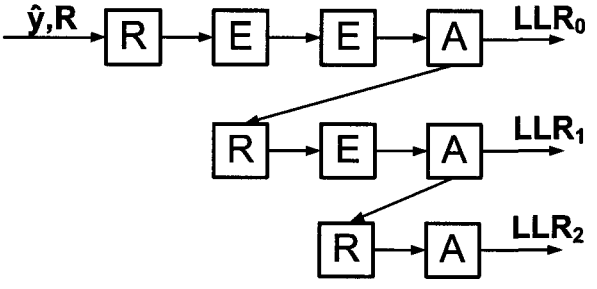


Figure 4.1 CUDA MIMO detector data flow

We took several additional steps to reduce the overall complexity of algorithm. Since both a reduction step and the edge reduction directly above prune the 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 share computation.

Figure 4.1 illustrates the steps for a 4×4 MIMO detector. The algorithm generates $\mathcal{L}_{\prime}$, $\mathcal{L}_{\infty}$, $\mathcal{L}_{\in}$ and $\mathcal{L}_{\ni}$ using a series of path reduction and path extension steps followed by APP computation. We will now describe implementation of each step of the detection algorithm, path extension, path reduction and LLR computation.

4.1.1 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 subpaths and path 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 path 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:

$$w_{k,q}^{<t>} = \left\| \hat{y}_{N-t-1} - \sum_{j=N-t-1}^{N-1} R_{(N-k-1,j)} s_j \right\|_2^2 \tag{4.1}$$

where h'_k is kth subpath and s_j is the j^{th} 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:

$$\delta_k = \sum_{j=N-1-k}^{N-2} R_{(N-1-k,j)} s_j \tag{4.2}$$

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

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

$$w_{k,q}^{<t>} = \left\| \hat{y}_{N-t-1} - \delta_k - R_{(N-1,N-1)} q_k \right\|_2^2 \tag{4.3}$$

Thread k picks the smallest outgoing path by evaluating the outgoing paths one by one. The path selected is the new k^{th} path. And 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 equation(4.2). 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 partial distance, d_k. Lines 14-17 search outgoing paths with the smallest partial distance serially. The path selected is the new k^{th} path.

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.

4.1.2 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 equation(4.1).

Algorithm 1 The k^{th} thread searches for the best outgoing path

1: //Calculate intermediate PD vectors

2: Calculate δ_k

3: //Search for the path with minimum partial distance serially

4: w = 0

5: Fetch d'_k from shared memory

6: Fetch Ω_0 from shared memory

7: Calculate $w_{k,0}^{<t>}$ using δ_k and Ω_0

8: Update d_k

9: $d_w = d_k$

10: **for** $q = 1$ to $Q - 1$ **do**

11: Fetch Ω_q from constant memory

12: Calculate $w_{k,q}^{<t>}$ using δ_k and Ω_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 wth path into k^{th} path history in shared memory

19: Store wth path's partial distance in shared memory

20: SYNC

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, which reduces complexity significantly. The search process is similar to path extension except each thread evaluates 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 are written to the shared memory for the next iteration.

4.1.3 LLR Computation

The algorithm generates a 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

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 δ_0 from shared memory

4: Fetch d'_0 from shared memory

5: Fetch Ω_q from constant memory

6: Calculate $w_{0,q}^{<t>}$ using δ_0 and Ω_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 δ_k from shared memory

12: Calculate $w_{k,q}^{<t>}$ using δ_k and Ω_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 wth path into q^{th} path history in shared memory

20: Store wth path's partial distance in shared memory

21: SYNC

useful work. However, each thread does computation independently and does not require any synchronization.

The candidate lists are the Q path partial distances. To compute LLR for k^{th} bit, the k^{th} thread looks at k^{th} bit, search for two smallest partial distances, one minimal partial distances where k^{th} bit is 0 and one minimal partial distances where k^{th} bit is 1. The difference between the two partial distances is the LLR. The steps in LLR computation are summarized in algorithm 3.

4.1.4 Additional Optimizations

Since GPU is connected to the host through the PCI-express bus, transport time results in measurable penalty. GPU supports asynchronous memory copy which allows global memory access to overlap with kernel execution. This is accomplished by breaking data into chucks and creating a stream per data chuck. While the kernel is preforming computation for one stream, memory operations, both reading from host memory to global memory as well as writing from global memory to host memory can happen in parallel. This minimizes the performance penalty due to transport overhead.

4.2 Performance Results

In the rest of the paper we will refer to our configurable multi-pass trellis traversal real-time MIMO detector on GPU simply the "MTT". To evaluate the performance

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** $k = 0$ 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: $LLR_k = \frac{(m_0 - m_1)}{\sigma}$

12: **end if**

13: SYNC

of "MTT detector", we tested our detector on a Linux platform with 8GB DDR2 memory running 800 MHz and an Intel Core 2 Quad Q6600 running at 2.4Ghz. The GPU used in our experiment is a Nvidia Telsa C1060 graphic card, which has 240 stream processors running at 1.3GHz and 4GB 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 QR-decomposition on the channel matrix H to generate $\mathbf{R}$ and $\hat{\mathbf{y}}$, which are fed into the detection kernel running on GPU.

4.2.1 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 solution which is exhaustive search. In addition, we compare MTT against the performance of K-Best, a well-known breadth-first algorithm. Finally, to measure how multiple passes through the trellis improves performance, we compared MTT against our first GPU MIMO detector, one-pass trellis detector (OT), which does only one-pass through the trellis. To mitigate inaccuracies in LLR computation due to the small list, we apply the LLR clipping technique to the K-Best detector [29] and OT. It should be noted that in the K-Best and one-pass trellis detector algorithm the Euclidean distance for hypothesis-0 or hypothesis-1 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 partial distance for hypothesis-0 and hypothesis-1 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 [30], which performs up to 15 LDPC iterations. Figures 4.2, 4.3, and 4.4 compare the BER performance of the MMT with the K-Best detectors. Figures 4.5, 4.6, and 4.7 compare the BER performance of the proposed 4×4 detector with the K-Best detectors. As can be seen, MTT performs better than K-Best detector with $K = M$ 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 K-Best detector with $K = M$. 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 path per trellis vertice and hence able to compute more accurate LLR for the decoder.

4.2.2 MTT Detector Throughput

We now look at the throughput of this detector on the GPU. To keep utilization high, a thread block detects multiple symbols in parallel – each thread block detects 8 symbols for 4-QAM, 2 symbols for 16-QAM, and 1 symbol for 64-QAM. In our

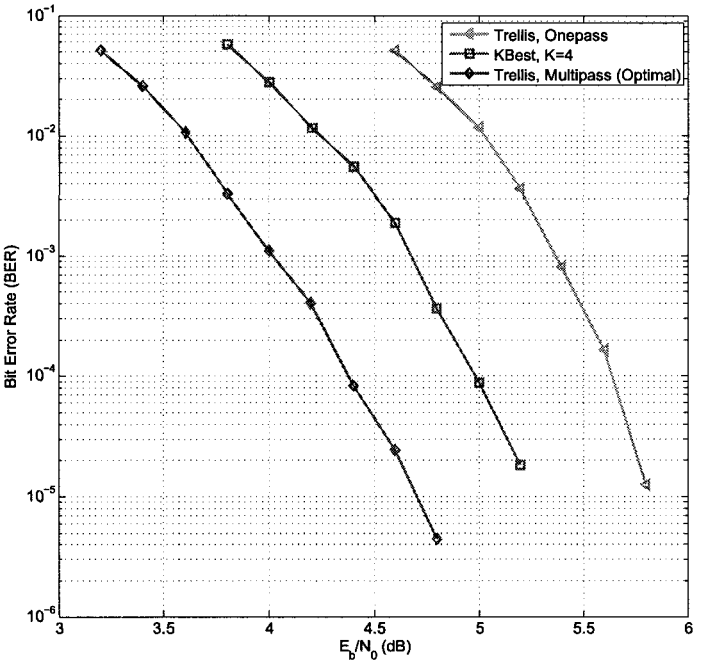


Figure 4.2 Simulation results for a LDPC-coded 2 × 2 4-QAM MIMO system.

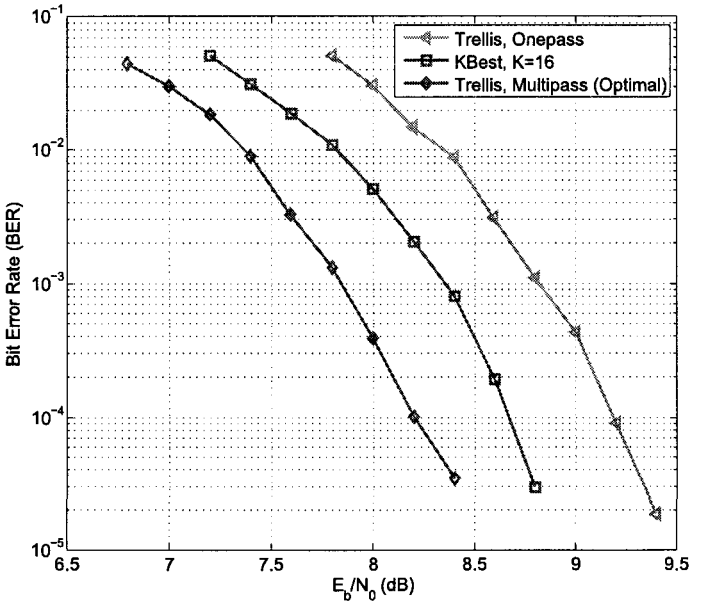


Figure 4.3 Simulation results for a LDPC-coded 2 × 2 16-QAM MIMO system.

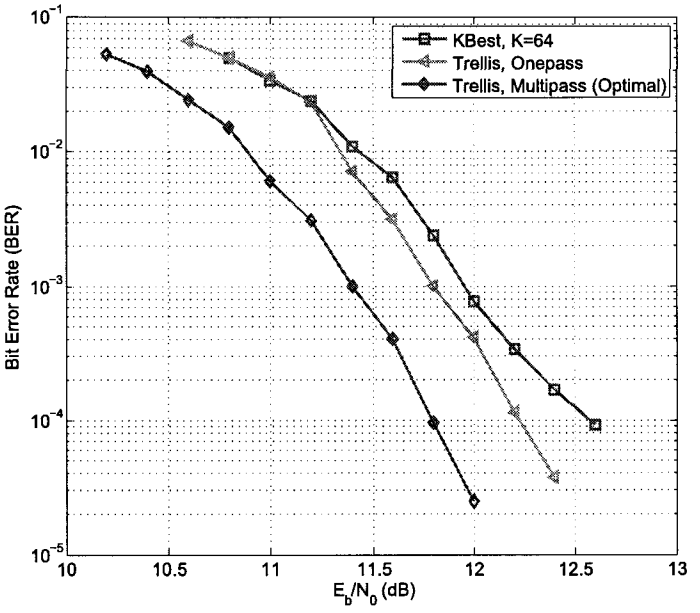


Figure 4.4 Simulation results for a LDPC-coded 2 × 2 64-QAM MIMO system.

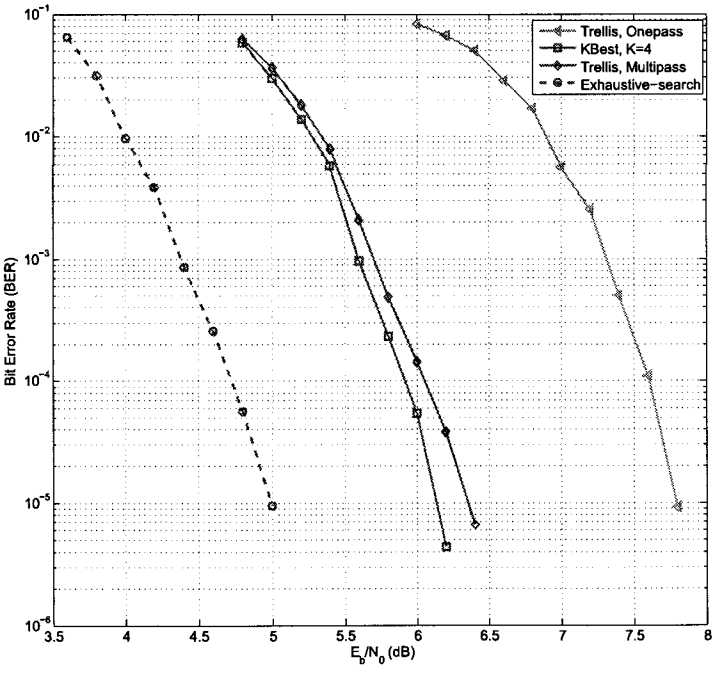


Figure 4.5 Simulation results for a LDPC-coded 4 × 4 4-QAM MIMO system.

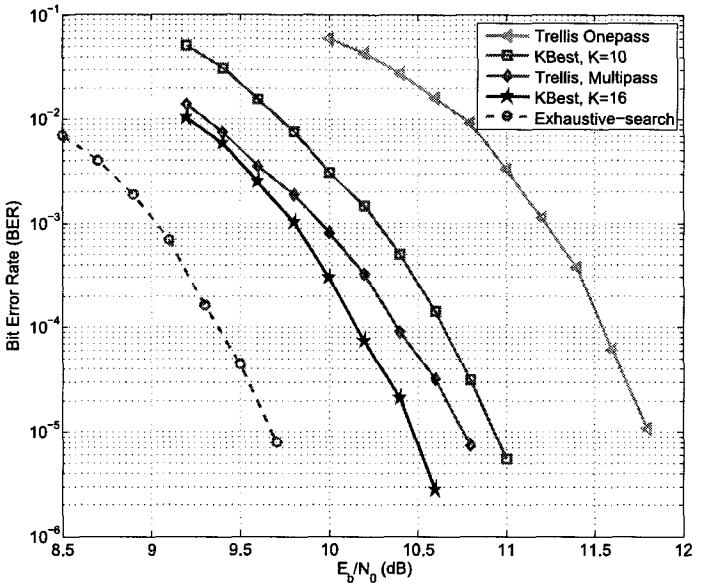


Figure 4.6 Simulation results for a LDPC-coded 4 × 4 16-QAM MIMO system.

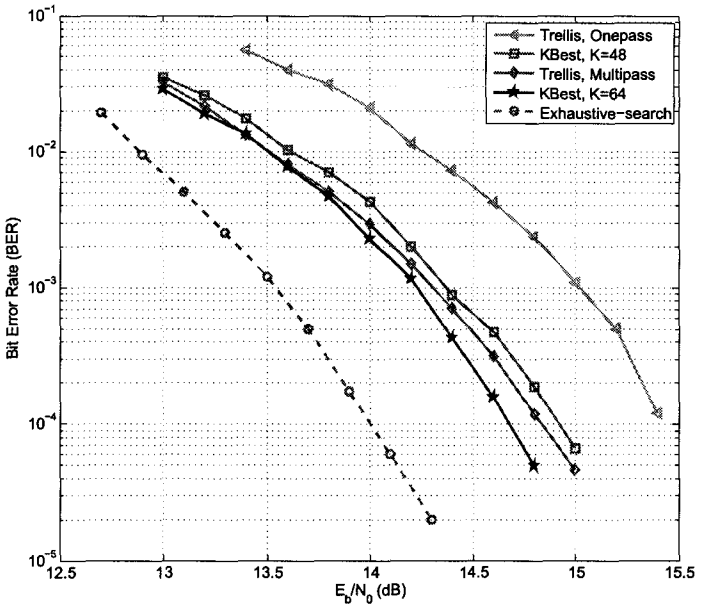


Figure 4.7 Simulation results for a LDPC-coded 4 × 4 64-QAM MIMO system.

benchmark, both 2 × 2 and 4 × 4 MIMO configurations are tested. The detector kernel detects 8 streams of 16384 symbols for 2 × 2 and 8 streams of 8192 symbols for 4 × 4. Execution time of the detector is averaged over 1000 runs. We compared both asynchronous and synchronous implementations of this MIMO detector.

Table 4.1 shows the execution time and the throughput performance for 2 × 2 MIMO detector. Table 4.2 shows the execution time and the throughput performance for 4 × 4 MIMO detector. The table includes performance of our synchronous implementation, our asynchronous implementation, as well as performance of the kernel of the MIMO detector.

Table 4.1 Average Runtime for 2 × 2

	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

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 eights stream and overlapping transfer and computation, our MIMO detector performs very close to kernel running time.

Table 4.2 Average runtime for 4×4

	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

MIMO-OFDM is used to achieve high data rate in a real time system such as 3GPP LTE and WiMAX. Figure 4.8 and figure 4.9 compares the throughput of the proposed detector with asynchronous memory transfer to the performance requirement of a 5 MHz LTE MIMO configuration.

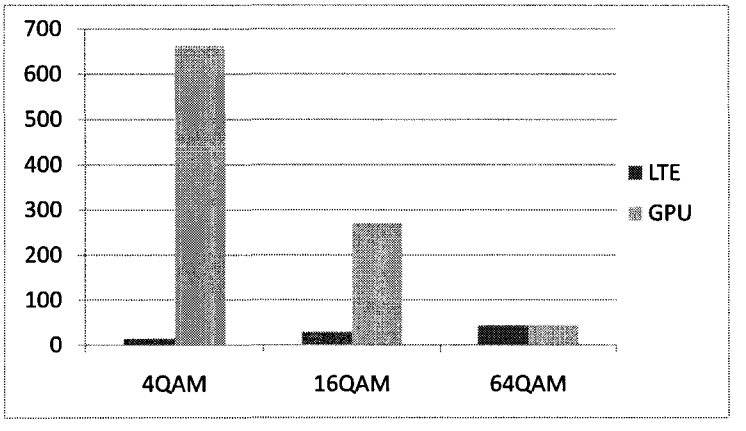


Figure 4.8 Performance compared to 5 MHz LTE 2×2 MIMO

Our detector can handle 4-QAM, 16-QAM, 64-QAM for 2×2 and 4×4 5 MHz LTE MIMO system. Since our detector can achieve more than four times the performance

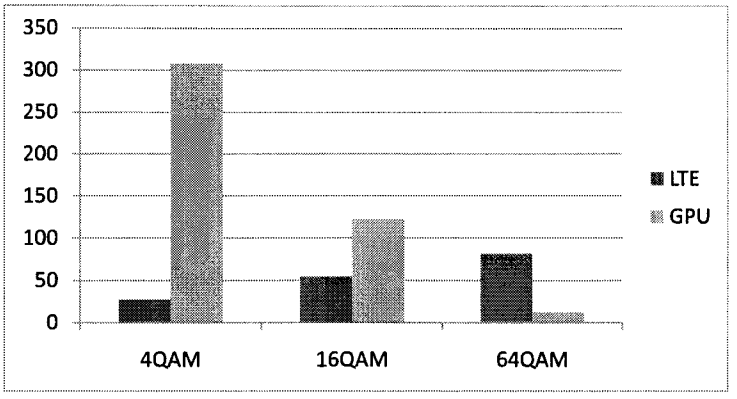


Figure 4.9 Performance compared to 5 MHz LTE 4 × 4 MIMO

requirement of 5 MHz LTE MIMO configuration for 4-QAM, 16-QAM for 2 × 2 and for 4-QAM 4 × 4 LTE MIMO system, our detector can also handle larger 20 MHz LTE MIMO configuration for these cases. Category 1 to category 4 devices are 2 × 2 devices, while category 5 devices are 4 × 4 devices [31]. To support 4 × 4, we need 7 times number of cores to support the workload. However, the number of cores in GPU is growing rapidly [31]. Assuming the number of cores continue to double after Fermi, we expect to meet the performance requirement by using two next generation graphic cards.

4.2.3 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 with allows for regular memory access, stall time can be reduced. CUDA Visual Profiler provides 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. Accordingly, the achieved instruction rate is I/T, where I is the number of executed *warp* instructions and T is the actual time 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:

$$R = \frac{I/T}{F_c/CPI} = \frac{I \times CPI \times F_c^{-1}}{T} \tag{4.4}$$

In CUDA, the average CPI is 4 cycles per instruction and each SM is clocked at 1.3GHz. The estimated runtime is shown in Table 4.3.

Table 4.3 Instruction Throughput Ratio for 2×2, 16800 subcarriers

Modulation	I	T	R
4-QAM	13894	0.08	0.549
16-QAM	137712	0.45	0.940
64-QAM	1601220	4.98	0.996

The ratio is smaller than 1 since instruction throughput ratio of 1 corresponds to the maximum instruction throughput. Instruction throughput ratio is lowest for 4-QAM since the detector does smaller number of computations per global memory fetch. Conversely, instruction throughput ratio is close to 1 for 16-QAM and 64-QAM

as stall due to long device memory access for the computation intensive cases as the detector does more computations for each global memory fetch.

4.2.4 Detector Instruction Mix

Instruction throughput ratio measures how well instructions for our MIMO detector executes on the hardware. However, it does not measure how well these instructions solve our problem. We use decuda, 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 APP computations. We measure 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 APP computations are completely unrolled. Particularly, path extension and path reduction are essentially the same. Each loop iteration consists of two add, two abs, two add, 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, another instruction stores the minimum partial distance for the next iteration. For both path extension and reduction, there are a total of 8 instruction per loop iteration. For 4×4, there is an additional store to save the index of the best path. For the APP computation, each loop iteration consists of one compare, one shared memory load and two stores.

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

$$N = 113 + 16Q + 8Q \tag{4.5}$$

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

$$N = 600 + 81Q + 16Q \tag{4.6}$$

Table 4.4 compares our model against the number of instructions reported by Nvidia Profiler for one thread block. Note that compiled code are strip-mined into 32 wide WARP instructions during execution. Therefore we divide the number of instruction reported by the profiler by 2 for 64QAM. Furthermore, the result reported by 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 time are loop iterations for 64-QAM. Since each iteration of the loop consists of reasonable number of instructions, performance of this MIMO detector

Table 4.4 Number of Instructions Per Threadblock

	2×2		4×4	
	Model	Profiler	Model	Profiler
4	209	209	988	1014
16	497	496	2152	2137
64	1649	1780	6808	9749

for 16-QAM and 64-QAM will not improve significantly more without changing the underlying algorithm.

4.2.5 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 4.5 compares our GPU design with state-of-the-art ASIC/FPGA/ASIP designs in terms of throughput. Compared to our previous work [32], this work is a better comparison since it is also a soft detector. In [33], 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 [34], 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 [26] 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 [33, 3, 34, 6] 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 4.5 Throughput comparison with ASIC/FPGA/ASIP solutions for 4×4 system.

	4x4 QPSK	4x4 16-QAM	4x4 64-QAM
GPU	284.7 Mbps	120.0Mbps	12.0Mbps
FPGA [34]	N/A	N/A	8.57 Mbps
ASIP [6]	N/A	5.3 Mbps	N/A
ASIC [33]	19.2 Mbps	38.4 Mbps	N/A
ASIC [3]	N/A	53.3 Mbps	N/A
ASIC [26]	300 Mbps	600 Mbps	N/A

Chapter 5
Turbo Decoder on GPU

5.1 Proposed Implementation on GPU

A straight-forward implementation of the decoding algorithm requires the completion of N stages of α_k computation before the start of β_k computation. Throughput of such a decoder would be low on GPU. First, the parallelism of this decoder would be low; since we would spawn only one thread block with 8 threads to traverse the trellis in parallel. Second, memory required to save N stages of α_k is significantly larger than the shared memory size. Finally, a traversal from stage 0 to stage $N-1$ takes many cycles to complete and leads to very long decoding delay.

Figure 5.1 provides an overview of our implementation. At the beginning of the decoding process, the inputs of the decoder, LLRs from the channel, are copied from the host memory to device memory. Instead of spawning only one thread-block per codeword to perform decoding, a codeword is split into P sub-blocks and uses P independent thread blocks in parallel. We still assign 8 threads per each thread block as there are only 8 trellis states. However, both the amount of shared memory required and the decoding latency are reduced as a thread-block only needs to traverse through $\frac{N}{P}$ stages. After each half decoding iteration, thread blocks are synchronized by writing extrinsic LLRs to device memory and terminating the kernel. In the device memory, we allocate memory for both extrinsic LLRs from the first half iteration and

extrinsic LLRs from the second half iteration. During the first half iteration, the P thread blocks read from extrinsic LLRs from the second half iteration. During the second half of the iteration, the direction is reversed. The α and β values between neighboring thread-blocks are exchanged to improve performance.

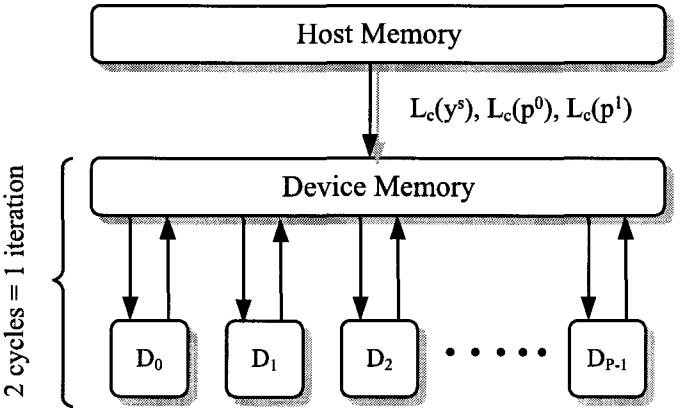


Figure 5.1 Overview of our MAP decoder implementation

Only one MAP kernel is needed as each half iteration of the MAP decoding algorithm performs the same sequence of computations. However, since the input changes and the output changes between each half iteration, the kernel needs to be reconfigurable. Specifically, the first half iteration reads a-priori LLRs and writes extrinsic LLRs without any interleaving or deinterleaving. The second half iteration reads a-priori LLRs interleaved and writes extrinsic LLRs deinterleaved. The kernel handles reconfiguration easily with a couple of simple conditional reads and writes at the beginning and the end of the kernel. Therefore, this kernel executes twice per iteration. The implementation details of the reconfigurable MAP kernel are described in the

following subsections.

5.1.1 Shared Memory Allocation

To increase locality of the data, our implementation attempts to prefetch data from device memory into shared memory and keep intermediate results on die. Since the backward traversal depends on the results from the forward traversal, we save $\frac{N}{P}$ stages of α_k values in shared memory from the forward traversal. Since there are 8 threads, one per trellis state, each thread block requires $\frac{8N}{P}$ floats for α. Similarly, we need to save β_k to compute β_{k-1}, which requires 8 floats. In order to increase thread utilization during extrinsic LLR computation, we save up to 8 stages of $\Lambda_k(s_k|u_b = 0)$ and $\Lambda_k(s_k|u_b = 1)$, which requires 128 floats. In addition, at the start of the kernel, we prefetch $\frac{N}{P}$ LLRs from the channel and $\frac{N}{P}$ a-priori LLRs into shared memory for more efficient access. A total of $\frac{10N}{P} + 196$ floats is allocated per thread-block. Since we only have $16KB$ of shared memory which is divided among concurrent executing thread blocks, small P increases the amount of shared memory required per thread block which reduces the number of concurrent executing thread blocks significantly.

5.1.2 Forward Traversal

During the forward traversal, each thread block first traverses through the trellis to compute α. We assign one thread to each trellis level; each thread evaluates two incoming paths and updates $\alpha_k(s_j)$ for the current trellis stage using α_{k-1}, the for-

ward metrics from the previous trellis stage $k-1$. The decoder use Equation (3.13) to compute α_k. The computation, however, depends on the path taken (s_{k-1}, s_k). The two incoming paths are known a-priori since the connections are defined by the trellis structure as shown in Figure 3.6. Table 5.1 summarizes operands needed for α computation. The indices of the α_k are stored in constant memory. Each thread

Table 5.1 Operands for α_k computation

	$u_b = 0$		$u_b = 1$	
Thread id (i)	s_{k-1}	p_k	s_{k-1}	p_k
0	0	0	1	1
1	3	1	2	0
2	4	1	5	0
3	5	0	6	1
4	1	0	0	0
5	2	1	3	1
6	5	1	4	1
7	6	0	7	0

loads the indices and the values $p_k|u_b = 0$ and $p_k|u_b = 1$ at the start of the kernel. The pseudo-code for one iteration of α_k computation is shown in Algorithm 4:

The memory access pattern is very regular for the forward traversal. Threads access

Algorithm 4 thread i computes $\alpha_k(i)$

1: $a_0 \leftarrow \alpha_{k-1}(s_{k-1}|u_b = 0) + L_c(y_k^s) * (p_k|u_b = 0)$

2: $a_1 \leftarrow \alpha_{k-1}(s_{k-1}|u_b = 1) + (L_c(y_k^s) + L_a(k))$

3: $\quad +L_c(p_k^s)(p_k|u_b = 1)$

4: $\alpha_k(i) = \max^*(a_0, a_1)$

5: SYNC

values of α_{k-1} in different memory banks. Since all threads access the same a-priori LLR and parity LLR in each iteration, memory accesses are broadcast reads. Therefore, there are no shared memory conflicts in either case, that is memory reads and writes are handled efficiently by shared memory.

5.1.3 Backward Traversal and LLR Computation

After the forward traversal, each thread block traverses through the trellis backward to compute β. We assign one thread to each trellis level to compute β, followed by computing Λ_0 and Λ_1 shown in Algorithm 5. The indices of β_{k+1} and value of p_k are summarized in Table 5.2. Similar to the forward traversal, there are no shared memory bank conflicts since each thread accesses an element of α or β in a different bank.

After computing Λ_0 and Λ_1 for stage k, we can compute the extrinsic LLR for stage k. However, there are 8 threads available to compute the single LLR, which introduces parallelism overhead. Instead of computing one extrinsic LLR for stage

Table 5.2 Operands for β_k computation

	$u_b = 0$		$u_b = 1$	
Thread id (i)	s_{k+1}	p_k	s_{k+1}	p_k
0	0	0	4	0
1	4	1	0	0
2	5	1	1	1
3	1	0	5	1
4	2	0	6	1
5	6	1	2	1
6	7	1	3	0
7	3	0	7	0

Algorithm 5 thread i computes $\beta_k(i)$ and $\Lambda_0(i)$ and $\Lambda_1(i)$

1: $b_0 \leftarrow \alpha_{k+1}(s_{k+1}|u_b = 0) + L_c(y_k^s) * (p_k|u_b = 0)$

2: $b_1 \leftarrow \alpha_{k+1}(s_{k+1}|u_b = 1) + (L_c(y_k^s) + L_a(k))$

3: $\quad + L_c(p_k^s)(p_k|u_b = 1)$

4: $\beta_k(i) = \max^*(b_0, b_1)$

5: SYNC

6: $\Lambda_0(i) = \alpha_k(i) + L_p(i)p_k + \beta_{k+1}(i)$

7: $\Lambda_1(i) = \alpha_k(i) + (L_c(k) + L_a(k)) + L_p(s_k)p_k + \beta_k(i)$

k as soon as the decoder computes β_k, we allow the threads to traverse through the trellis and save 8 stages of Λ_0 and Λ_1 before performing extrinsic LLR computations. By saving eight stages of Λ_0 and Λ_1, we allows all 8 threads to compute LLRs in parallel efficiently. Each thread handles one stage of Λ_0 and Λ_1 to compute an LLR. Although this increases thread utilization, threads need to avoid accessing the same bank when computing extrinsic LLR. For example, 8 elements of Λ_0 for each stage is stored in 8 consecutive addresses. Since there are 16 memory banks, elements of even stages Λ_0 or Λ_1 with the same index would share the same memory bank. Likewise, this is true for even stages of Λ_0. Hence, sequential accesses to Λ_0 or Λ_1 to compute extrinsic LLR will result in four way memory bank conflicts. To alleviate this problem, we permute the access pattern based on thread ID as shown in Algorithm 6.

5.1.4 Interleaver

The interleaver is used in the second half iteration of the MAP decoding algorithm. In our implementation, a quadratic permutation polynomial (QPP) interleaver [35], which is proposed in the 3GPP LTE standard was used. Although the QPP interleaver is contention free since it can guarantee bank free memory access, where each subblock accesses a different memory bank. However, the memory access pattern is still random. Since the inputs are shared in device memory, memory accesses are not necessarily coalesced. We reduce latency by pre-fetching data into the shared

Algorithm 6 thread i computes $L_e(i)$

1: $\lambda_0 = \Lambda_0(i)$

2: $\lambda_1 = \Lambda_1(i)$

3: **for** $j = 1$ to 7 **do**

4: $index = (i + j)\&7$

5: $\lambda_0 = \max^*(\lambda_0, \Lambda_0(index))$

6: $\lambda_1 = \max^*(\lambda_1, \Lambda_1(index))$

7: $L_e = \lambda_1 - \lambda_0$

8: Compute write address

9: Write L_e to device memory

10: **end for**

memory. The QPP interleaver is defined as:

$$\Pi(x) = f_1 x + f_2 x^2 \pmod{N}. \tag{5.1}$$

Direct computation of $\Pi(x)$ using Equation (5.1) can cause overflow. For example, 6143^2 can not be represented as a 32-bit integer. The following equation is used to compute $\Pi(x)$ instead:

$$\Pi(x) = (f_1 + f_2 x \pmod{N}) \cdot x \pmod{N} \tag{5.2}$$

Another alternative is to compute $\Pi(x)$ recursively [13], which requires $\Pi(x)$ to be computed before we can compute $\Pi(x+1)$. This is not efficient for our design as we need to compute several interleaved addresses in parallel. For example, during the second half of the iteration to store extrinsic LLR, 8 threads need to compute 8 interleaved address in parallel. Equation (5.2) allows efficient address computation in parallel.

Although our decoder is configured for the 3GPP LTE standard, one can replace the current interleaver function with another function to support other standards. Furthermore, we can define multiple interleavers and switch between them on-the-fly since the interleaver is defined in software in our GPU implementation.

5.1.5 max* Function

Both natural logarithm and natural exponential are supported on CUDA. We support full-log-MAP as well as max-log-MAP [36]. We compute full-log-MAP by:

$$\overset{*}{\max}(a, b) = max(a, b) + ln(1 + e^{-|b-a|}) \tag{5.3}$$

and max-log-MAP is defined as:

$$\overset{*}{\max}(a, b) = max(a, b). \tag{5.4}$$

Throughput of full-log-MAP will be slower than the throughput of max-log-MAP. Not only is the number of instructions required for full-log-MAP greater than the number of instructions required for max-log-MAP, but also the natural logarithm and natural exponential instructions takes longer to execute on GPU compared to common floating operations, e.g. multiply and add. An alternative is using a lookup table in constant memory. However, this is even less efficient as multiple threads access different entries in the lookup table simultaneously, only the first entry will be a cached read.

5.2 Performance Results

To evaluate the performance of our Turbo decoder, we tested our Turbo decoder on a Linux platform with 8GB DDR2 memory running at 800 MHz and an Intel Core 2 Quad Q6600 running at 2.4Ghz. The GPU used in our experiment is a Nvidia

TESLA C1060 graphic card, which has 240 stream processors running at 1.3GHz with 4GB of GDDR3 memory running at 1600 MHz.

5.2.1 Decoder Performance

Since our decoder can change P, which is the number of sub-blocks to be decoded in parallel, we first look at how the number of parallel sub-blocks affects the overall decoder performance. In our setup, the host computer first generates the random bits and encodes the random bits using a 3GPP LTE Turbo encoder. After passing the input symbols through the channel with AWGN noise, the host generates LLR values which are fed into the decoding kernel running on GPU. For this experiment, we tested our decoder with $P = 32, 64, 96, 128$ for a 3GPP LTE Turbo code with $N = 6144$. In addition, we tested both full-log-MAP as well as max-log-MAP with the decoder performing 6 decoding iterations.

Figure 5.2 shows the bit error rate (BER) performance of the our decoder using full-log-MAP, while Figure 5.3 shows the BER performance of our decoder using max-log-MAP. In both cases, performance of the decoder decreases as we increase P. The performance of the decoder is significantly better when full-log-MAP is used. Furthermore, we see that even with parallelism of 96, where each sub-block is only 64 stages long, provides performance that is within 0.1dB of the performance of the optimal case ($P = 1$).

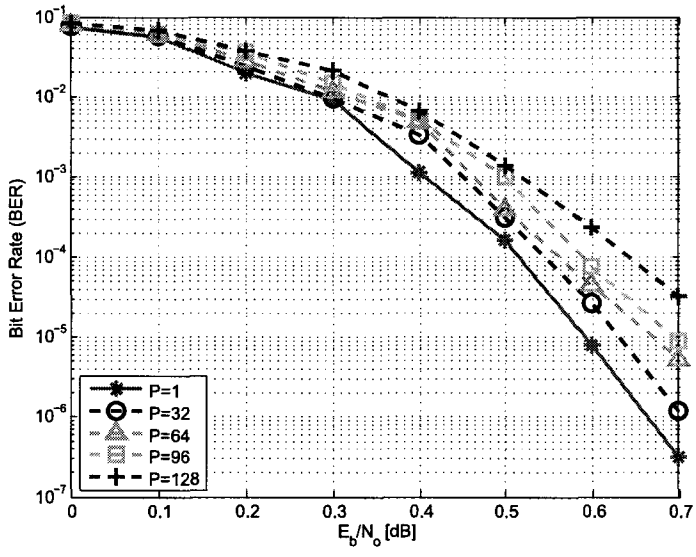


Figure 5.2 BER performance (BPSK, full-log-MAP)

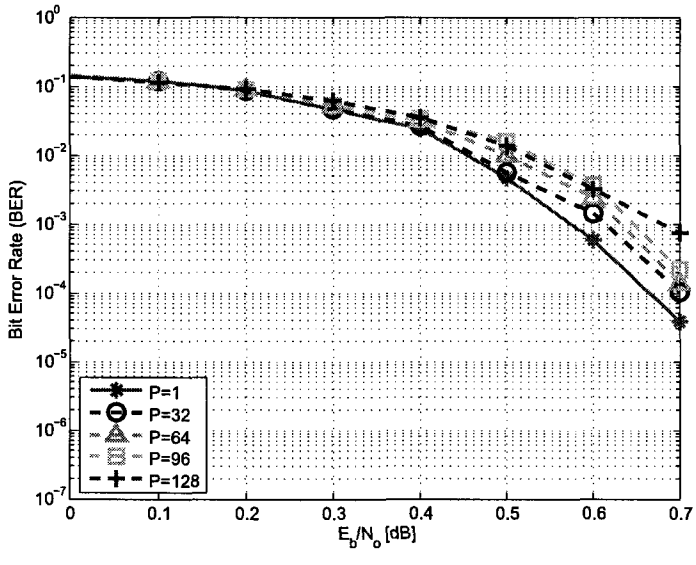


Figure 5.3 BER performance (BPSK, max-log-MAP)

5.2.2 Decoder Throughput

In this case, we measure the time it takes the decoder to decode a batch of 100 codewords. Since our decoder can support various code sizes, we can decode $N = 64, 1024, 2048, 6144$ with various numbers of decoding iterations and parallelism P. However, we noticed that performance of the decoder is only dependent on $W = \frac{N}{P}$. This is expected as decoding time is linearly dependent with the number of trellis stages that the decoder needs to traverse. Therefore, we report the decoder throughput as a function of W which can be used to find the throughput of different decoder configurations. For example, if $N = 6144$, $P = 64$, and the decoder performs 1 iteration, the throughput of the decoder is the throughput when $W = 96$. The throughput of the decoder is summarized in Table 5.3.

Throughput of the decoder is inversely proportional to the number iterations performed. The throughput of the decoder after m iterations can be approximated as T_0/m, where T_0 is the throughput of the decoder after 1 iteration.

Although throughput of full-log-MAP is slower than max-log-MAP as expected, the difference is small while full-log-MAP improves performance of the decoder significantly. Therefore, full-log-MAP is a better choice for this architecture.

Table 5.3 Throughput vs W

	Max-log-MAP throughput/ Full-log-MAP throughput (Mbps)			
# Iter	W=32	W=64	W=96	W=128
1	49.02/36.59	34.75/23.87	26.32/19.50	17.95/12.19
2	24.14/18.09	17.09/12.72	12.98/9.62	8.82/5.59
3	16.01/12.00	11.34/8.45	8.57/6.39	5.85/3.97
4	11.98/9.01	8.48/6.51	6.41/4.78	4.37/2.97
5	9.57/7.19	6.77/5.2	5.12/3.82	3.49/2.37
6	7.97/5.99	5.64/4.33	4.26/3.18	2.91/1.97

5.2.3 Architecture Comparison

Table 5.4 compares our proposed decoder with other programmable Turbo decoder solutions. As can be seen, our decoder with $W = 64$ compares favorably in terms of throughput and performance. We can support both the full-log-MAP (FLM) algorithm and the simplified max-log-MAP(MLM) algorithm while most other solutions only support the sub-optimal max-log-MAP algorithm.

Table 5.4 Our decoder vs other programmable Turbo decoders

Work	Architecture	MAP Algorithm	Throughput	Iter.
[37]	Intel Pentium 3	MLM	366 Kbps	1
[38]	Motorola 56603	MLM	48.6 Kbps	5
[38]	STM VLIW DSP	FLM	200 Kbps	5
[39]	TigerSHARC DSP	MLM	2.399 Mbps	4
[40]	TMS320C6201 DSP	MLM	500 Kbps	4
[12]	32-wide SIMD	MLM	2.08 Mbps	5
ours	Nvidia C1060	MLM/FLM	6.77/5.2Mbps	5

Chapter 6
Conclusion and Future Work

Both MIMO detector and Turbo decoders are used in current and upcoming standards to improve performance of the wireless systems. The inherently large decoding latency and a complex iterative decoding algorithm have made it very difficult to achieve high throughput in general purpose processors or digital signal processors. As a result, these communication blocks are implemented in ASIC or FPGA. In this thesis, we aim to show that GPUs, homogeneous multi-core processors, can handle the workload and achieve high throughput. Since not all algorithms map well on this architecture, we showed how to implement these processing blocks efficiently on GPU. Particularly, we presented a reconfigurable soft MIMO detector and a 3GPP LTE compliant Turbo decoder on GPU. In the case of MIMO detector, we showed the performance of multi-pass trellis traversal performs similar to 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 showed that this algorithm is well-suited to the GPU. In addition, we showed our detector's throughput compared well with the conventional fixed-point VLSI and FPGA implementations. In the case of Turbo decoder, we implemented a parallel

window algorithm. By dividing the codeword into many sub-blocks to be decoded in parallel, workload was partitioned across cores on GPU. We also presented how both performance and throughput was affected by sub-block size and faster throughput than other programmable devices even though the full-log-MAP algorithm is used. Since both MIMO detection and the decoder is done in software, we can reconfigure the detector and decoder to support different MIMO configuration and different code standards.

The architecture of GPU is driven by the demand for graphics toward photo-realism, and graphics processors are becoming more powerful with each revision. The next generation Nvidia graphics processor, Fermi [41], will increase performance while reducing programming complexity by offering the following changes. First the amount of shared memory is increased and a L1 cache and L2 cache are added. Each SIMD core now has 64KB of memory which can be partitioned between L1 cache and shared memory by the programmer. All SIMD cores are now connected to a unified L2 cache. The cache hierarchy will improve performance in several ways. For example, the increased amount of shared memory will improve the performance of Turbo decoder as we can prefetch more data on-die and decrease parallelism required to achieve good throughput. The unified L2 cache will reduce performance loss of the Turbo decoder by allowing the thread blocks to share data across cache instead of external device memory. Furthermore, although the architecture remains SIMD, Fermi allows

multiple kernels to execute concurrently. For example, MIMO detection kernel and Turbo decoding kernel execute concurrently with the decoding block. The advance in the GPU architecture will allow us to improve the current decoder by evaluating other partitioning and memory strategies to improve performance and throughput. Furthermore, this will allow us to implement a completely iterative MIMO receiver by combining this decoder with MIMO detector on GPU.

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. 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.

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. D. Garrett, B. Xu, and C. Nicol, "Energy efficient turbo decoding for 3G mobile," in *International symposium on Low power electronics and design.* ACM, 2001, pp. 328–333.

8. C. Chaikalis and J. Noras, "Reconfigurable turbo decoding for 3G applications," *Elsevier Signal Processing*, vol. 84, pp. 1957–1972, Oct. 2004.

9. M. Bickerstaff, L. Davis, C. Thomas, D. Garrett, and C. Nicol, "A 24Mb/s radix-4 logMAP turbo decoder for 3GPP-HSDPA mobile wireless," in *IEEE Int. Solid-State Circuit Conf. (ISSCC)*, Feb. 2003.

10. M. Martina, M. Nicola, and G. Masera, "A Flexible UMTS-WiMax Turbo Decoder Architecture," *IEEE Transactions on Circuits and Systems II*, vol. 55, April 2008.

11. M. Shin and I. Park, "SIMD processor-based turbo decoder supporting multiple third-generation wireless standards," *IEEE Trans. on VLSI*, vol. vol.15, pp. pp.801–810, Jun. 2007.

12. Y. Lin, S. Mahlke, T. Mudge, C. Chakrabarti, A. Reid, and K. Flautner, "De-

sign and implementation of turbo decoders for software defined radio," in *IEEE Workshop on Signal Processing Design and Implementation (SIPS)*, Oct. 2006.

13. Y. Sun, Y. Zhu, M. Goel, and J. R. Cavallaro, "Configurable and Scalable High Throughput Turbo Decoder Architecture for Multiple 4G Wireless Standards," in *IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, July 2008, pp. 209–214.

14. P. Salmela, H. Sorokin, and J. Takala, "A Programmable Max-Log-MAP Turbo Decoder Implementation," *Hindawi VLSI Design*, vol. vol.2008, pp. pp. 636–640, 2008.

15. C.-C. Wong, Y.-Y. Lee, and H.-C. Chang, "A 188-size 2.1mm^2 reconfigurable turbo decoder chip with parallel architecture for 3GPP LTE system," in *2009 Symposium on VLSI Circuits*, June 2009, pp. 288–289.

16. D.-S. Cho, H.-J. Park, and H.-C. Park, "Implementation of an efficient UE decoder for 3G LTE system," in *International Conference on Telecommunications*, June 2008.

17. J. Berkmann, C. Carbonelli, F. Dietrich, C. Drewes, and W. Xu, "On 3G LTE Terminal Implementation - Standard, Algorithms, Complexities and Challenges," in *International Wireless Communications and Mobile Computing Conference*, Aug. 2008.

18. J.-H. Kim and I.-C. Park, "A unified parallel radix-4 turbo decoder for mobile WiMAX and 3GPP-LTE," in *IEEE Custom Integrated Circuits Conference*, Sept. 2009, pp. 487–490.

19. 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.

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

21. U. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany, "The Imagine stream processor," in *Proceedings 2002 IEEE International Conference on Computer Design*, Sep. 2002, pp. 282–288.

22. W. J. Dally, U. J. Kapasi, B. Khailany, J. H. Ahn, and A. Das, "Stream processors: Progammability and efficiency," *ACM Queue*, vol. 2, no. 1, pp. 52–62, 2004.

23. *Advanced Micro Devices, ATI Compute Abstraction Layer (CAL) Programming Guide (v2.0)*, 2010. [Online]. Available: http://developer.amd.com/documentation

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

25. U. Fincke and M. Pohst, "Improved Methods for Calculating Vectors of Short Length in a Lattice, Including a Complexity Analysis," *Mathematics of Computation*, vol. 44, no. 170, pp. 463–471, April 1985.

26. 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.

27. ——, "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.

28. L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate," *IEEE Transactions on Information Theory*, vol. IT-20, pp. 284–287, Mar. 1974.

29. Y. L. C. de Jong and T. J. Willink, "Iterative tree search detection for mimo wireless systems," *IEEE Tran. on Comm.*, vol. 53, no. 6, pp. 930–935, 2005.

30. Y. Sun and J. R. Cavallaro, "A low-power 1-Gbps reconfigurable LDPC decoder

design for multiple 4G wireless standards," in *IEEE International SOC Conference*, Spet. 2008, pp. 367–370.

31. "Evolved Universal Terrestrial Radio Access (EUTRA) and Evolved Universal Terrestrial Radio Access Network (EUTRAN), 3GPP TS 36.300."

32. Y. Sun and J. R. Cavallaro, "Reconfigurable real-time mimo detector on gpu," in *IEEE 43rd Asilomar Conference on Signals, Systems and Computers (ASILOMAR'09)*, Nov. 2009.

33. 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.

34. 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.

35. J. Sun and O. Takeshita, "Interleavers for turbo codes using permutation polynomials over integer rings," *IEEE Trans. Inform. Theory*, vol. vol.51, pp. 101–119, Jan. 2005.

36. P. Robertson, E. Villebrun, and P. Hoeher, "A comparison of optimal and sub-optimal MAP decoding algorithm operating in the log domain," in *IEEE Int. Conf. Commun.*, 1995, pp. 1009–1013.

37. M. Valenti and J. Sun, "The UMTS Turbo Code and a Efficient Decoder Implementation Suitable for Software-Defined Radios," *International Journal of Wireless Information Networks*, vol. 8, no. 4, pp. 203–215, Oct. 2001.

38. H. Michel, A. Worm, M. Munch, and N. Wehn, "Hardware software trade-offs for advanced 3G channel coding," in *Proceedings of Design, Automation and Test in Europe*, 2002.

39. K. Loo, T. Alukaidey, and S. Jimaa, "High performance parallelised 3GPP turbo decoder," in *IEEE Personal Mobile Communications Conference*, April 2003, pp. 337–342.

40. Y. Song, G. Liu, and Huiyang, "The implementation of turbo decoder on DSP in W-CDMA system," in *International Conference on Wireless Communications, Networking and Mobile Computing*, Dec. 2005, pp. 1281–1283.

41. *NVIDIA Corporation, Fermi Compute Architecture White Paper*, 2010. [Online]. Available: http://www.nvidia.com/object/fermi_architecture.html