GPU Acceleration of Fixed Complexity Sphere Decoder for Large MIMO Uplink Systems

Tianpei Chen, Harry Leib
Department of Electrical and Computer Engineering
McGill University
Montreal, Quebec

Abstract

Large MIMO constitutes a principal techniques for next generation wireless communication systems. Detection algorithms for large MIMO tend to require long simulation times on conventional computer systems, In order to reduce the simulation times for decoders of large MIMO systems, we propose to use General Purpose Graphic Processing Units (GPGPU). In this paper Fixed Complexity Sphere Decoder (FCSD) is implemented for the large scale MIMO system based on the dominant framework of GPGPU-Computed Unified Device Architecture (CUDA). The proposed CUDA-FCSD can achieve considerable speedup over conventional CPU implementation while reaching the same Bit Error Rate (BER) performance. Furthermore the proposed algorithm demonstrates great potential to accelerate large MIMO systems.

1 Introduction

The present challenge in the wireless communication area is the sharp contrast between the increasing demand of mobile network speed for the transmission of rich media content with better QoS (quality of service), and the shortage of radio frequency spectrum. Multiple Input Multiple Output (MIMO) is thought to be a very promising technology that can help to solve this problem owing to several advantages it can offer. The core idea of MIMO is to use multiple antennas that at the transmitting and receiving sides, that provide parallel data pipes in space resulting in spatial multiplexing gains [1]. For example the LTE standard supports up to 8 antennas at the base station [2].

Large scale MIMO (also named massive MIMO) systems employ hundreds of low-power low-price antennas collocated at the base station and serving several tens of terminals at the same time [3]. This technology originated from multiuser MIMO (MU-MIMO) systems [4]. Large scale MIMO will inherit and enhance all the advantages of conventional MIMO (spatial diversity gain and spatial multiplexing gain), as well as reduction of latency and improvement of system robustness. A common detector for MIMO system is the Sphere Decoder (SD) that perform as well as a Maximum Likelihood detector (MLD) with a dramatic complexity reduction [5] [6]. There are two major problems with the traditional SD decoder: complexity depending on Signal to Noise Ratio (SNR), and sequential nature, which are contradictory with the requirement of a constant data processing throughput and parallel implementation for practical system. The Fixed Complexity Sphere Decoder (FCSD) overcomes these two disadvantages of the SD decoder by searching the solution along a fixed number of independent paths [7].

General purpose Graphic processing Units Computing (GPGPU) has become a common trend in industry and research in areas that need massive computations, such as computer vision [8], computer games [9], signal processing [10] [11] and finance [12]. Compared to CPU, the GPU can naturally cope with computational intensive tasks, since most of the hardware resource are allocated for data processing and parallel computation, rather than caching and flow control. In recent years GPU computing has provided significant advantages in data processing speeds over CPU computations [13]. In 2011, the NVIDIA GPUs have become the fastest supercomputer in the world [14], Computed unified Device Architecture (CUDA) is a programming model for GPGPU that was proposed by NIVIDIA and has become the dominant framework for GPGPU programming. It is based on popular programming languages(C, C++, Java, Python, Fortran, Direct Compute), such that the application developers in different areas do not need to learn a new programming language or get to know the hardware details. CUDA employs the Single Instruction Multiple Threads (SIMT) programming model, enables a GPU to implement parallel data processing, and has become a parallel computing platform for wide community of different applications.

In this paper, a GPU based implementation of Fixed Complexity Sphere Decoder is proposed, A heterogeneous programming solution for FCSD preprocessing and blocked parallel path searching solution are presented, the speedup as well as Bit Error Rate (BER) performance of different sizes of MIMO systems and signal constellations are presented and analyzed, The

proposed CUDA-FCSD implementation demonstrates a great potential for large MIMO systems.

The rest of the paper is organized as follows. Section 2 describes the MIMO systems model. Section 3 introduces CUDA GPU architecture and the programming model. In section 4, we present details of the implementation of CUDA-FCSD. In section 5, the speedup and bit error rate (BER) performance of CUDA-FCSD are presented and analyzed. Finally conclusions are drawn in section 6.

2 MIMO System Model

We consider a complex uncoded spatial multiplexing MIMO system with N_r receive and N_t transmit antennas, $N_r \geq N_t$, over a flat fading channel, corresponding to the discrete time model:

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

$\mathbf{s} \in \mathbb{C}^{N_t \times 1}$ is the transmitted symbol vector, with components that are mutually independent and taken from the signal constellation $\mathbb{O}$ (4-QAM, 16-QAM, 64-QAM), of size M. The transmitted symbol vectors forms a N_t complex signal vector space $\mathbb{O}^{N_t}$. $\mathbb{E}[\mathbf{s}\mathbf{s}^H] = \mathbf{I}_{N_t} E_s$, where E_s denotes the symbol average energy, and $\mathbb{E}[\cdot]$ denotes the expectation operation. Furthermore, $\mathbf{y} \in \mathbb{C}^{N_r \times 1}$ is the received symbol vector, $\mathbf{H} \in \mathbb{C}^{N_r \times N_t}$ denotes the Rayleigh fading channel propagation matrix with independent identity distributed(i.i.d) circularly symmetric complex Gaussian zero mean components of unit variance, $\mathbf{n} \in \mathbb{C}^{N_r \times 1}$ is the additive white Gaussian noise (AWGN) samples with zero mean components and $\mathbb{E}[\mathbf{n}\mathbf{n}^H] = \mathbf{I}_{N_r} N_0$, N_0 denotes the average noise power spectrum density and $\frac{E_s}{N_0}$ denotes the signal to noise ratio (SNR).

Assume we have the perfect channel state information (CSI), meaning that $\mathbf{H}$ is known at the receiver, as well as the SNR. The task of MIMO decoder is to estimate $\mathbf{s}$ based on $\mathbf{y}$ and $\mathbf{H}$.

3 CUDA GPU Architecture and Programming Model

3.1 Memory Hierarchy

CUDA programming model is a C programming extension, with special defined syntax and structure. The serial code is executed on the host (CPU) side, and massive parallel data processing is performed on the device (GPU) side, this is defined as a heterogeneous programming model. A thread is the basic programming unit of CUDA, a block of cooperative threads form a thread block, in one block all the threads can share one pitch of on chip memory (shared memory), all the threads executed in one block can synchronize their action meaning the threads are permitted to run to the next instruction only when all the threads have reached the same instruction node. Blocks can be grouped into a one-dimensional, two-dimensional, or three-dimensional grid. Similarly, The threads can also be organized into one-dimensional, two-dimensional or three-dimensional blocks. The organization of the block and grid shape and dimension is determined by the application, Figure 1 illustrates the 3 dimensional thread hierarchy of CUDA.

3.2 Single Instruction Multiple Thread (SIMT) Model

Threads are scheduled, managed, created and executed in a group of 32 threads called warp, the programs executes on the device side are defined as *kernel* function, when a *kernel* function is invoked, the thread blocks are partitioned into warps, the threads in one warps executes the same instruction simultaneously, the different warps are scheduled by warp scheduler, when one warp is waiting for data loading the other warps begin to execute, this is called Single Instruction Multiple Thread (SIMT) model. This programming architecture is askin to Single Instruction Multiple Data (SIMD), however unlike the SIMD vectoring machines, SIMT can branch the behavior of each threads to achieve thread-level parallelism. The GPU use massive number of threads to hide the long latency of global memory instead of using large cache as CPU. In a word CPU cores are designed to minimize latency for each thread at a time, whereas GPUs are designed to handle a large number of concurrent, lightweight threads in order to maximize throughput.

3.3 Mircroarchitecture of GPU

In Figure 2 we use the GPU-GeForce GTX 760 as an example to show the microarchitecture of GPU. The GPU works as a coprocessor system,

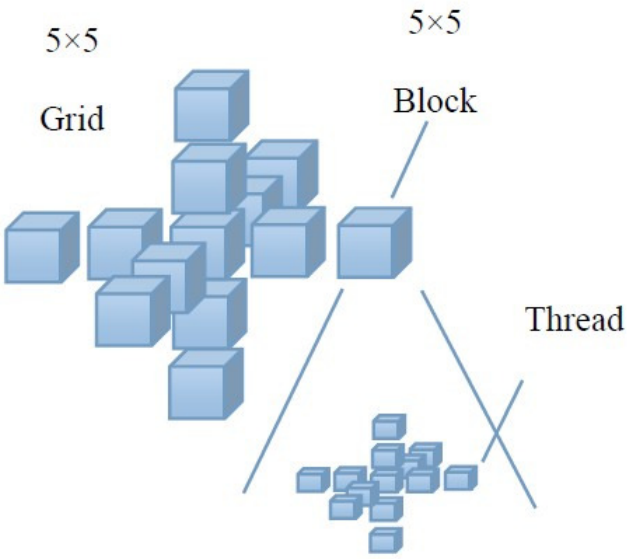

Figure 1: CUDA 3D thread hierachy
$5 \times 5 \times 5$ grid and each block contains $5 \times 5 \times 5$ threads

consisting of Stream Multiprocessors (SM). The GeForce GTX 760 has 6 SM that perform the multiply-add arithmetic operation. Furthermore, each SM has the special functional units(SFU) that can perform more complex arithmetics such as trigonometric functions and reciprocal square root. The shared memory, also called on-chip memory located on SM and has low latency and limited capacity. Global memory, which is off-chip memory of the GPU has long latency and large capacity. Another two types of the memory, texture memory and constant memory, are located off-chip, but cached. Therefore the loading speed of the data located in these two type of memory space is much faster than that from global memory. Constant memory is read only and with latency approximately as small as the registers. The variables invoked in *kernel* functions can be stored in either of the these four types of memories. Registers are allocated dynamically and privately to threads, the access speed of registers is the faster than all the other four types of memories.

Because of the resource limitation of SM the total number of threads can be defined in one block is limited. On current CUDA supported GPUs, one block can contains up to 1024 threads, and all the blocks are executed on SM. Thus the total number of threads (T_{total}) we can make use of is $T_{total} = N_{SM} \times N_{threads/SM}$, N_{SM} and $N_{threads/SM}$ denote the number of SM and number of threads per SM. The GeForce GTX 760 allows 2048 threads in one SM, thus the total number of threads that can work in parallel is $2048 \times 6 = 12288$. In concrete applications, the programmer can allocate more than 12288 threads for a kernel, but the threads will work in serial, which will influence the program performance.

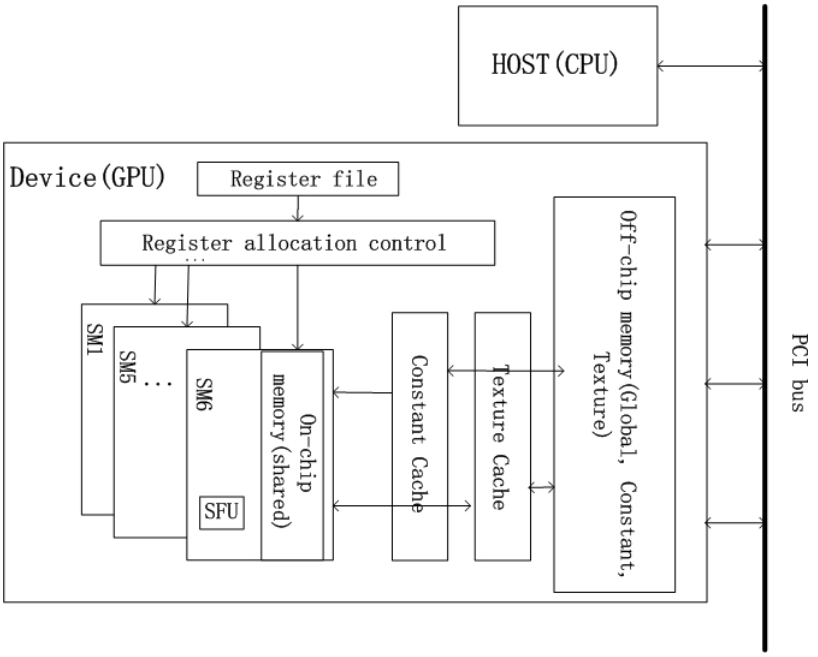

Figure 2: High View of CUDA CPU Microarchitecture

4 GPU Based Acceleration of FCSD

In this section we present the parallel implementation of fixed complexity sphere decoder using GPU, From (1), the maximum likelihood detector (MLD) for a MIMO system is specified by:

$$\mathbf{s}_{ML} = arg \min_{\mathbf{s} \in \mathbb{O}^{N_t}} ||\mathbf{y} - \mathbf{Hs}||^2 \tag{2}$$

We consider a MIMO system with $N_t = N_r$. Performing the QR decomposition to the channel propagation matrix

$$\mathbf{H} = \mathbf{QR} \tag{3}$$

where $\mathbf{Q} \in \mathbf{C}^{N_t \times N_t}$ is an unitary matrix and $\mathbf{R} \in \mathbf{C}^{N_t \times N_t}$, is an upper triangular matrix, the basing on [15], we can write (2) as:

$$\begin{aligned} \mathbf{s}_{ML} &= arg \min_{\mathbf{s} \in \mathbb{O}^{N_t}} ||\mathbf{Q}^H\mathbf{y} - \mathbf{Q}^H\mathbf{Hs}||^2 \\ &= arg \min_{\mathbf{s} \in \mathbb{O}^{N_t}} ||\mathbf{Q}^H\mathbf{y} - \mathbf{Rs}||^2 \end{aligned} \tag{4}$$

Consider the unconstrained estimation of $\mathbf{s}$, $\hat{\mathbf{s}} = \mathbf{Gy}$, $\mathbf{G} = (\mathbf{H}^H\mathbf{H})^{-1}\mathbf{H}^H$ based on (3), we have

$$\hat{\mathbf{s}} = \mathbf{Gy} = (\mathbf{R}^H\mathbf{R})^{-1}\mathbf{R}^H\mathbf{Q}^H\mathbf{y} = \mathbf{R}^{-1}\mathbf{Q}^H\mathbf{y} \tag{5}$$

Therefore $\mathbf{Q}^H\mathbf{y} = \mathbf{R}\hat{\mathbf{s}}$, and (4) can be written as

$$\mathbf{s}_{ML} = arg \min_{\mathbf{s} \in \mathbb{O}^{N_t}} ||\mathbf{R}(\hat{\mathbf{s}} - \mathbf{s})||^2 \tag{6}$$

Just like SD the FCSD also performs depth first searching, at the first ρ node levels the FCSD search all the possible signal symbols in constellation $\mathbb{O}^{N_t}$, which is called Full Expansion(FE) stage, therefore for the symbol nodes $\widetilde{s}_{N_t}\widetilde{s}_{N_t-1}\ldots\widetilde{s}_{N_t-\rho+1}$, the FCSD has M branches expansion at each symbol node, as shown in Figure 3, there are $\rho = 2$ FE nodes and each node level has $M = 4$ branches. (6) can be written in the form:

$$\hat{s}_i^F = arg \min_{s \in \mathbb{O}^{N_t}} \{\sum_{i=1}^{N_t} |r_{ii}(\hat{s}_i - s_i) + \sum_{j=i+1}^{N_t} r_{ij}(\hat{s}_j - s_j)|^2\} \tag{7}$$

$i \in [1, 2 \ldots N_t]$, where r_{ij} denotes the component at the ith row and jth column of the upper triangular matrix $\mathbf{R}$, based on (7) at the remaining $N_t - \rho$ levels of symbols, the FCSD performs decision feed back, therefore there are only 1 branch expansion for each symbol node at this stage, this stage is called Single Expansion (SE). Thus the total number of the branches is fixed (For example in Figure 3 the total branch number is 4^2), the FCSD works as a constant number of multiple path tree searching algorithm. In conclusion the FCSD symbol vector candidate $s_i^F = [s_1, s_2 \ldots s_{N_t}]$ can be expressed as:

$$s_i^F = \begin{cases} \Omega \in \mathbb{O}^\rho \quad i = N_t, N_t - 1, \ldots N_t - \rho + 1 \\ \mathbb{Q}[\hat{s}_i + \sum_{j=i+1}^{N_t} \frac{r_{ij}}{r_{ii}}(\hat{s_j} - s_F^i)] \quad i = N_t - \rho, N_t - \rho - 1, \ldots 2, 1 \end{cases}$$

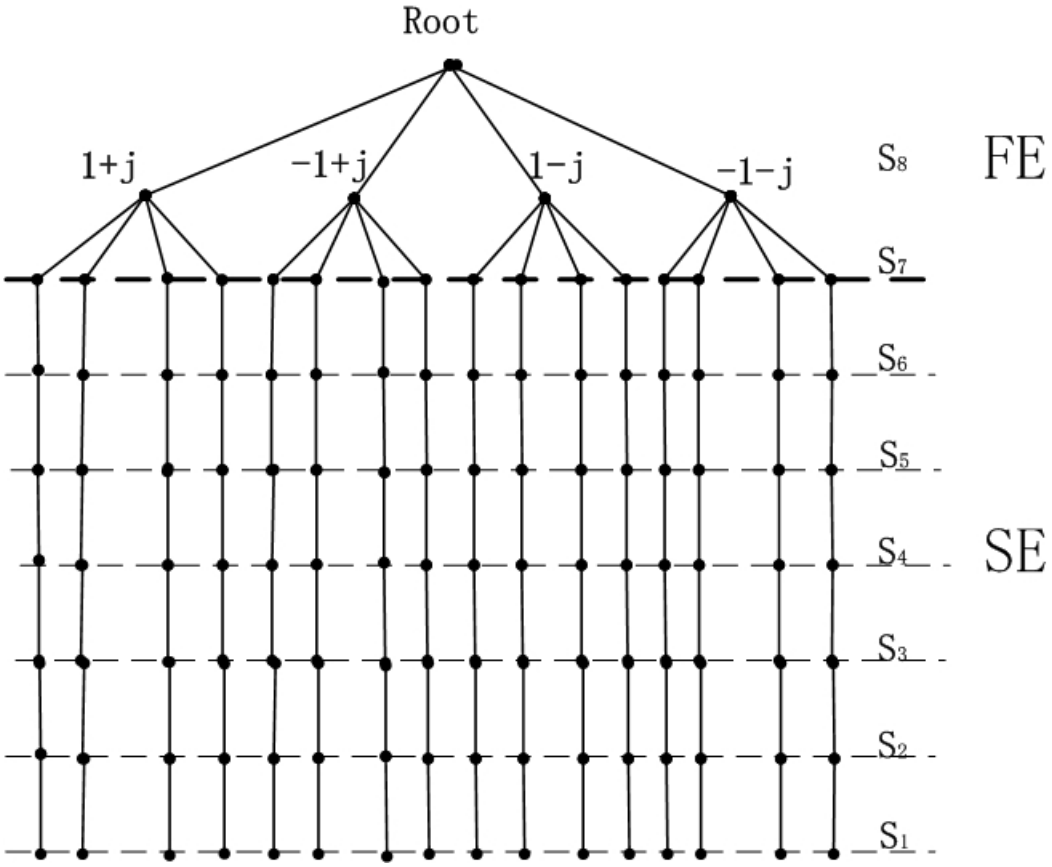

Figure 3: Tree searching of 4-QAM FCSD for 8x8 MIMO system

$\mathbb{Q}[\cdot]$ denotes the signal constellation quantization. At the post processing stage, FCSD compares the Euclidean distance of all the symbol vector candidates E_{metric}, which, according to 2, can be written as:

$$E_{metric} = ||\mathbf{R}(\hat{\mathbf{s}} - \widetilde{\mathbf{s}})||^2 \tag{8}$$

Where the $\widetilde{\mathbf{s}}$ denotes the symbol vector candidate. The total number of candidates is:

$$M^{\rho} \tag{9}$$

The symbol vector candidate with the minimum E_{metric} is chosen as the solution.

4.1 CUDA-FCSD Preprossing

As shown in Figure 3, each path of the FCSD searching has a serial execution nature. In order to avoid error propagation, FCSD ordering was applied

to the propagation channel based on post processing signal to noise ratio [16]:

$$\varphi_m = \frac{E_s}{\sigma^2(\mathbf{H}^H\mathbf{H})^{-1}_{mm}} \tag{10}$$

where φ_m denotes the post processing signal to noise ratio of the mth data stream, and $(\mathbf{H}^H\mathbf{H})^{-1}_{mm}$ denotes the diagonal elements of the inversion of $\mathbf{H}^H\mathbf{H}$. When the SNR is determined, the post processing SNR is only determined by $(\mathbf{H}^H\mathbf{H})^{-1}_{mm}$. From the heuristic in [7], for the FE stage the robustness of detector performance is not influenced by the post processing SNR. Thus at FE stage the "weakest" data stream, that is, the data stream with the smallest φ, is detected first. At SE stage, because of the single searching path branch of the node, the performance is tightly related to the post processing SNR. Therefore at SE stage the data streams with the larger post processing SNR is detected first in order to avoiding error propagation. In conclusion the FCSD ordering works iteratively and obeys the following rule: at the jth step ($j = 1, 2 \ldots N_t - 1$), s_p denotes the symbol to be detected, where

$$p = \begin{cases} arg\max_k(\mathbf{H}_j^H\mathbf{H}_j)^{-1}_{kk} & FE \quad stage \\ arg\min_k(\mathbf{H}_j^H\mathbf{H}_j)^{-1}_{kk} & SE \quad stage \end{cases}$$

and $\mathbf{H}_j \in \mathbb{C}^{N_r \times (N_t - j + 1)}$ denotes the renewed channel matrix. At the jth iteration with the column corresponding to previous detected data streams removed from $\mathbf{H}_{j-1}$.

As mention in section 3, our implementation is based on heterogeneous programming principle-achieving performance with a mix of CPU and GPU technology, the flow control and logical operations are reside at the host side, the compute intensive works is implemented at device side. Although the FCSD preprocessing has a serial nature and can not be vectorized, the computational consumptive operations can be performed by the CUDA Basic Linear Algebra Subprograms(cuBLAS) and corresponding GPU program implementations, as shown in table 1.

Figure 4 illustrates the block diagram of the CUDA-FCSD implementation.

4.2 Parallel Acceleration of Blocked-Paths Searching

We consider 16×16 MIMO system, the modulation scheme is 16QAM, the number of path we need to search is $\mathbb{O}[M^\rho] = 16^3 = 4096$, $\rho = \lceil \sqrt[2]{N_t} - 1 \rceil$, as mentioned in section 3 the maximum number of threads that can be used

Operations	complex matrix-matrix multiplication	complex matrix-vector multiplication	matrix inverse	cholesky factorization
GPU subroutine	*cublasCgemm*	*cublasCgemv*	*GJE*	*chol*
Formula	$\mathbf{H}^H\mathbf{H}$	$(\mathbf{H}^H\mathbf{H})^{-1}\mathbf{H}^H y$	$\mathbf{H}_j^H\mathbf{H}_j)^{-1}$	$\mathbf{H}^H\mathbf{H} = \mathbf{R}^H\mathbf{R}$

Table 1: Parallel Computation Operation of FCSD Preprocessing

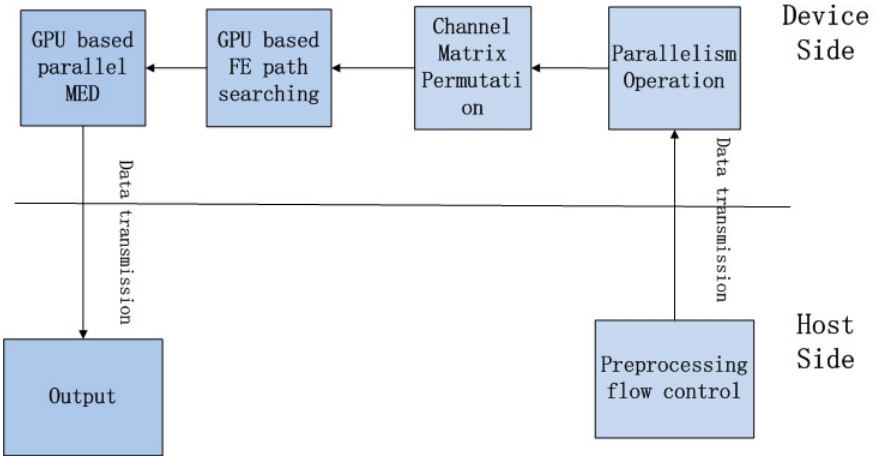

Figure 4: Block Diagram of CUDA-FCSD Implementation

in GPU is limited by the number of SM and maximum number of threads per SM, as to GeForce GTX 760. we have 12288 threads totally. The decision feedback searching paths have serial nature, we match one path to one thread. In order to have largest amount of threads that can be paralleled, we use one dimension blocks for widest expansion and blocked all the paths into several one dimension blocks. For largest SM occupation we define 4 blocks for path searching kernel and each block has 1024 threads which is the maximum number of thread allowing per block. At each FCSD searching path, the following data operation is performed in one thread:
1. *Decision Feedback Equalization*

$$\begin{aligned} s_i^k \in \mathbb{O}^\rho \quad if \quad i &= N_t, N_t-1, \ldots N_t-\rho+1 \quad FE \quad stage \\ s_i^k = \mathbb{Q}[\hat{s}_i^k + \sum_{j=i+1}^{N_t} \frac{r_{ij}}{r_{ii}}(\hat{s}_j^k - s_j^k)] \quad if \quad i &= N_t-\rho, N_t-\rho-1, \ldots 2, 1 \quad SE \quad stage \end{aligned} \tag{11}$$

$\mathbf{s}_j^k$ denotes the jth symbol of the kth symbol vector candidate.
2. *Euclidean distance Calculation*

$$E_u^k = \sum_{i=1}^{i=N_t} \sum_{j=i}^{j=N_t} r_{ij}(\hat{s}_j - s_j) \tag{12}$$

E_u^k denotes the kth Euclidean distance of the corresponding symbol vector candidate. The symbol vector candidate with minimum Euclidean distance is chosen as the solution.

4.2.1 Data Preparation

There are three major factors that will influence the configuration of the data sets configuration in memory space.
1. Amount of Reading&Writing (R/W) operation per thread.
2. Valid scope
3. Storage occupancy

Unlike CPUs that have a flat memory model meaning the CPU core can access any memory location without restriction, GPUs are computation intensive and only have small cache for special memory space (texture, constant). Thus the memory reading&writing (R/W) strategy of GPU is the crutial factor that influences the application performance. Different memory types has different valid scope, local memory which is invoked in kernel function is valid only in one thread, share memory is valid in one block, constant, texture and global memory are valid both at host side and device side. On the other hand different from global memory (2047 MBytes in GeForce GTX 760), the constant and share memory are all capacity limited (only 65536 bytes totally and 49152 bytes per block in GeForce GTX 760). Thus the occupancy of memory is one determinant factor needed to be considered.

At host side we set 4 data sets that are stored in global memory which are storage consumptive and must have valid scope for both host and device side, s_{pm} is a $M^\rho \times N_t$ complex matrix that each row stores one symbol vector candidate and whose valid scope should be during the whole *kernel* execution. s_{index} is used to store the index of all the possible sub symbol vectors for FE stage using full factorial method, requires a $\rho \times M^\rho$ integer matrix space. E_u is a M^ρ floating point vector that is used to store the Euclidean distance of all the symbol vector candidates. s_{kernel} is a complex vector used to store the FCSD solution which length is N_t and has to be transferred to

host side.

On the other hand, upper triangular matrix $\mathbf{R}$ in (6) and unconstrained estimation $\hat{\mathbf{s}}$ in (5) are read only and require large amount of read operations. (for the 16×16 $16QAM$ MIMO system the R/W of $\mathbf{R}$ and $\hat{\mathbf{s}}$ are about 370 and 266 per thread), these two data set need small size of storage(for the 16×16 $16QAM$ MIMO system 1088 bytes for $\mathbf{R}$ and 128 bytes for $\hat{\mathbf{s}}$) we make use of the read only constant memory to store the data which are capacity limited and do not need write operation in path searching , as we mentioned in section 3, the constant memory is cached so that it is much faster than global memory.

In table 2 we present the amount of R/W operations and the storage that needed for different data variables in FCSD path searching. The amount of the R/W operation is calculated based on the (11) and (12), for complex values ($\mathbf{R}$ $\hat{\mathbf{s}}$, s_{pm}, s_{kernel}) are stored using CUDA build in variable data type *cuComplex* which counts 8 bytes. Floating point data set E_u is stored using variable type *float*, which counts for 4 bytes, s_{index} is the integer list which is stored using variable type *int*, which counts 1 byte.

data	$\mathbf{R}$	$\hat{\mathbf{s}}$	s_{pm}	s_{index}	E_u	s_k
R/W operations/thread	$(\frac{3}{2}(N_t)^2 - \frac{N_t}{2} + \rho - \rho^2)$	$(\frac{2(N_t)^2+2N_t-\rho^2-\rho}{2})$	N_t	ρ	N_t	N
Storage/bytes	$4(N_t+1)N_t$	$8N_t$	$8M^\rho N_t$	$N_t M^\rho$	$4N_t M^\rho$	8

Table 2: Amount of R/W Operations and Storage Requirement of Different Data Set

4.2.2 Memory Accesss Pattern

When considering large MIMO array, the large amount of R/W operations determine that the memory bandwidth is the major bottleneck of the performance, although the theoretical bandwidth of off-chip memory is extremely high (in GeForce GTX 760 the bandwidth of off-chip memory is around $192GB/s$), however the effect memory bandwidth can only be achieved by proper R/W strategy.

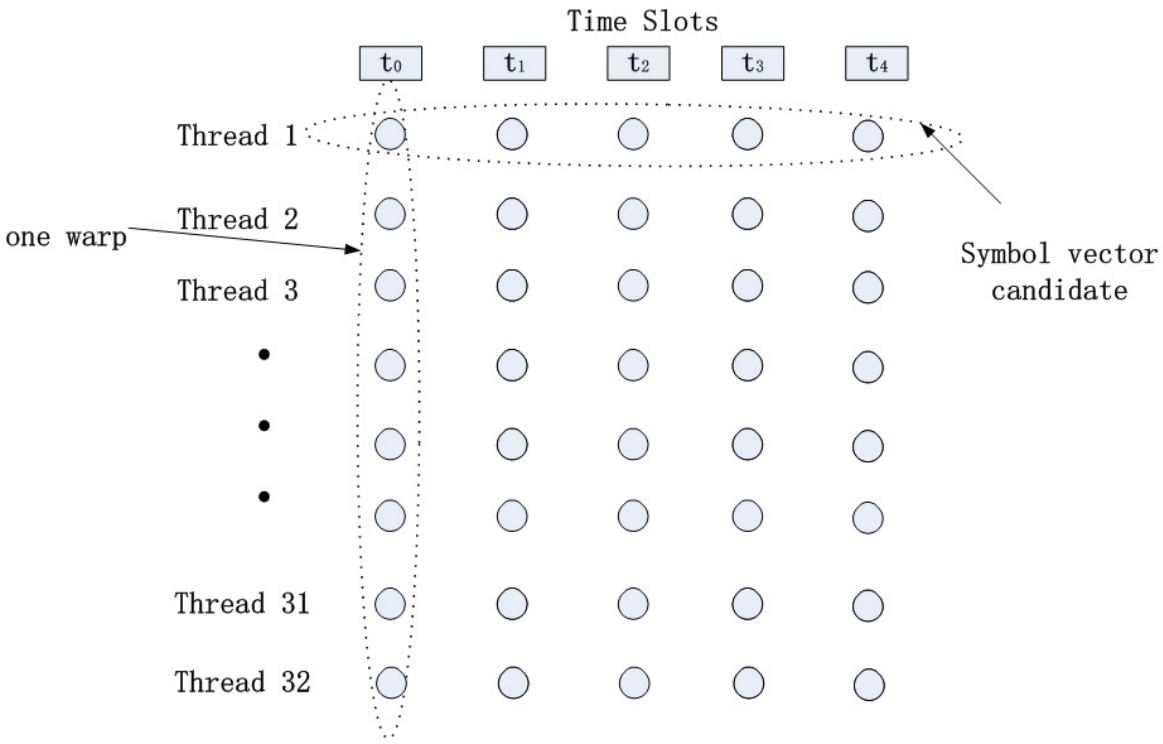

Figure 5: Global Memory Access Pattern of s_{pm}

Global memory is implemented by dynamic random access memory (DRAM) whose R/W is extremely slow process, modern DRAM R/W operations are performed by access a pitch of consecutive memory location, as mentioned in section (3), the warp is the basic schedule unit of device, with one instruction recall one warp of threads that execute at the same time, the most favorable global memory access pattern is to keep one warp of threads that access consecutive memory as much as possible. For the GeForce GTX 760, the DRAM will access a consecutive memory address that has 128 bytes length, In Figure 5 we take the global access pattern of the s_{pm} as an example to illustrate how we coalesce the the global memory. We store s_{pm} in one pitch of linear global memory, with each row store one $N_t = 16$ complex symbol vector solution candidate of $\hat{\mathbf{s}}^F$ in (7), totally there are $M^\rho = 4096$ rows, this matrix is stored in column major. As mentioned before one path searching is performed by one thread, thus the address two consecutive symbol operated in one thread is 4096, however for consecutive threads the address they reach is continues as shown in Figure 5, each 32 threads in one warp can make R/W operation to a pitch of continues memory, which is $32 \times 8 = 256$ bytes, counts for two times of DRAM access.

As we mentioned in section (3), the registers are allocated to the threads dynamically and have much faster access speed than that of global memory. Therefore, for all the transient data that used for accumulator in (11), (12), and the symbol vector candidate during the path searching process of each thread, are stored temporarily in registers to hide the latency of R/W of global memory.

4.2.3 Data Transfer Minimization

The data transfer between the host and the device side is much more slower than the inner device data transferring, therefore, in our application we minimize the data transferring times, we make data transfer only once for propagation channel matrix $\mathbf{H}$, received symbol vector $\mathbf{y}$ in (1) as well as the sub symbol vector index list s_{index} at FE stage. Loop fusion is applied to post processing stage, we integrate the post processing (12) into path searching kernel function to reduce the iteration times per thread and avoid the transferring of E_u in (12).

4.3 Going to Larger MIMO System

Consider the uplink 32×32 MIMO system with 16QAM modulation scheme, the path number that need to be searched is $M^{\rho} = 16^5 = 1048576$. The maximum number of threads that can be execute in *kernel* function of the GeForce GTX 760 is 12288, therefore we need to consider execute the *kernel* function in loop, in order to avoid device synchronization latency, we synchronize and return the control to host side in order to reduce the synchronization expense. Under this condition we use 12 linear blocks and each block has 1024 threads, we execute the *kernel* function for 86 times and get the final solution, when it comes to the large MIMO systems and large signal constellations such as 36×36 16QAM, 144×144 4QAM, 20×20 64QAM, the global memory that need to used to store s_{pm} in section (4.2.1) will out of bound (2 GB), multiple kernels with each one deal with a small subset of symbol vector candidates and kernel concurrency [13] can be applied to solve this problem.

5 Simulation Results

5.1 Environment

5.1.1 Device

Graphic Processing Units: GeForce GTX 760, GPU clock rate: 1.08 GHz , memory clock rate: 3 GHz , number of SM: 6, maximum number of threads per stream multiprocessor: 2048, total amount of global memory: 2 GB, total amount of share memory per block: 49 KB, total amount of constant memory: 66 KB.
Central Processing Units-Modigliani: Intel Core i5-4th generation, 4 cores 3.20 GHz CPU clock rate, 8 GB RAM
Central Processing Units-Monet : Intel Core i7-3rd generation, 12 cores 3.20 GHz CPU clock rate, 32 GB RAM

5.1.2 Software

CUDA Driver Version / Runtime Version 6.5 / 6.0
Nsight Eclipse Version 6.5

5.2 Performance and Evaluation

Array size	Time/s			Speedup	
	GeForce GTX 760	Modigliani	Monet	GTX 760/Modigliani	GTX 760/Mo
8×8	7.39	0.10	0.11	0.01	0.01
16×16	16.26	0.92	0.98	0.06	0.06
32×32	38.32	31.27	34.60	0.82	0.90
48×48	71.95	262.00	285.14	3.64	3.96
64×64	322.90	1753.30	1940.10	5.43	6.01
72×72	1285.41	8727.69	9641.80	6.79	7.50
84×84	6454.02	47095.33	49962.01	7.30	7.74

Table 3: speedup performance of different MIMO systems using 4 QAM

We made simulation to different sizes MIMO systems and signal constellation and comparison is made between the Geforce GTX 760 and two kinds of CPU, the speedup performance is presented in table 3, 4 and 5. All the

Array size	Time/s			Speedup	
	GeForce GTX 760	Modigliani	Monet	GTX 760/Modigliani	GTX 760/Mo
8×8	13.20	0.67	0.93	0.05	0.07
16×16	26.80	31.68	41.95	1.18	1.57
20×20	192.70	740.50	978.26	3.84	5.08
32×32	4980.13	28209.53	38106.00	5.66	7.65
36×36	5568.26	35240.16	47290.00	6.33	8.61

Table 4: speedup performance of different MIMO systems using 16 QAM

Array size	Time/s			Speedup	
	GeForce GTX 760	Modigliani	Monet	GTX 760/Modigliani	GTX 760/Mon
8×8	12.28	10.69	13.38	0.87	1.09
16×16	468.92	2066.08	2689.00	4.41	5.73

Table 5: speedup performance of different MIMO systems using 64 QAM

operation times we test is based on 1000 channel realizations, the transmitted data is generated randomly and modulated by M quadrature amplitude modulation (QAM) scheme.

In table 3, for small MIMO systems 8×8 and 16×16, the speed performance of GPU implementation is even worse than that of CPU. The number of path of these antenna configurations that need to be searched is relatively small (16 for 8×8 system and 64 for 16×16 system), the CPU implementation has already had a good performance, the CPUs are designed for serial code execution with the special hardwares (branch prediction units, multiple caches, etc) and have extremely good performance at it, however the GPUs can achieve a good performance only when they are utilized in parallel manner, there is a short time for GPU to "warm up", for host-device data transmission, launch kernel function and host device synchronization. Furthermore, for small number of threads, the memory R/W latency is not well hided. For the 32×32 systems, the speed performance of GPU and CPU implementation is close, when comes to larger MIMO systems 48×48, 64×64, 72×72, a large parallelism is achieved so that the GPU begin to display its data processing power, the speedup can be from 2.50 to 4.20.

In table 4 and 5, 16QAM and 64QAM are considered, the speedup of GPU implementation is reached from a small MIMO size, For 16×16 system using 16QAM 1.18 speedup over Modigliani and 1.57 speedup over Monet.

For 8×8, 64QAM, 0.87 speedup over Modigliani and 1.09 speedup over Monet. For lager signal constellation, the massive parallelism can be reached for relatively small MIMO systems.

It can be observed from table 3, 4 and 5, GPU implementation shows its speed advantage over CPU implementation for increasing size of MIMO systems and signal constellations, indicating this GPU implementation's suitability for large MIMO system and large signal constellation.

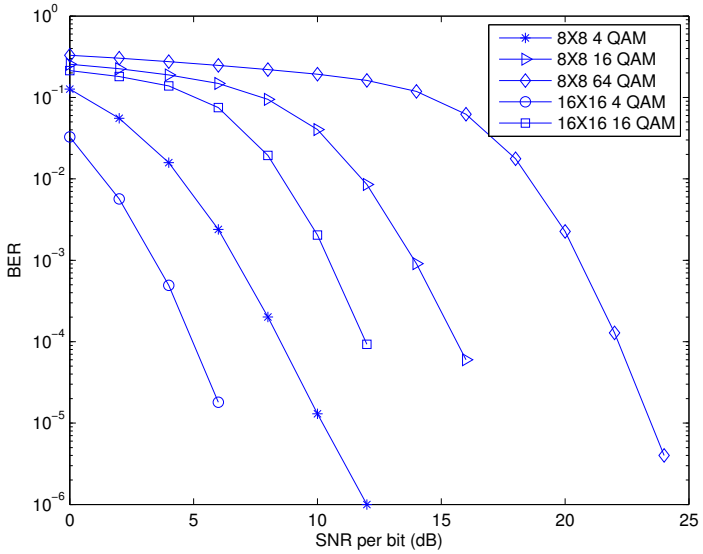

Figure 6: BER performance of different sizes of MIMO systems and modulation scheme by GPU

Figure 6 shows the Monte Carlo simulation results of bit error rate (BER) performance for different MIMO systems and constellation sizes, we consider uncoded spatial multiplexing MIMO systems, the transmitted binary sequence is generated randomly and mutually independent, modulated by M-QAM modulation scheme, the results have been obtained using 100000 channel realizations and 500 symbol errors accumulated.

6 Conclusion

This paper presents the GPU implementation of fixed complexity sphere decoder for a large MIMO system, which is a time consuming task for tra-

ditional computation platform. In order to exploit the computational capability of GPU in the simulation of large MIMO systems, the utilization of heterogeneous programming model, thread resource limitation and memory configuration are considered in a comprehensive way. The simulation shows considerable acceleration of GPU implementation of FCSD can be reached for large MIMO systems and large signal constellation sizes while the same BER performance can be obtained. In addition, the GPU and CPU can work independently, the data processing power of GPU computing is an extra bonus of the traditional simulation platform. This shows the potential of GPU computing for computational consumptive simulation in large MIMO area. The future work contains taking into consider the kernel execution and data transferring concurrence in order to optimize memory allocation strategy for large MIMO systems.

References

[1] C. Oestges and B. Clerckx, *MIMO wireless communications: from real-world propagation to space-time code design*. Academic Press, 2010.

[2] E. Dahlman, S. Parkvall, J. Skold, and P. Beming, *3G evolution: HSPA and LTE for mobile broadband*. Academic press, 2010.

[3] F. Rusek, D. Persson, B. K. Lau, E. G. Larsson, T. L. Marzetta, O. Edfors, and F. Tufvesson, "Scaling up mimo: Opportunities and challenges with very large arrays," *Signal Processing Magazine, IEEE*, vol. 30, no. 1, pp. 40–60, 2013.

[4] E. G. Larsson, O. Edfors, F. Tufvesson, and T. L. Marzetta, "Massive mimo for next generation wireless systems," *arXiv preprint arXiv:1304.6690*, 2013.

[5] E. Viterbo and J. Boutros, "A universal lattice code decoder for fading channels," *Information Theory, IEEE Transactions on*, vol. 45, no. 5, pp. 1639–1642, 1999.

[6] B. Hassibi and H. Vikalo, "On the sphere-decoding algorithm i. expected complexity," *Signal Processing, IEEE Transactions on*, vol. 53, no. 8, pp. 2806–2818, 2005.

[7] L. G. Barbero and J. S. Thompson, "Fixing the complexity of the sphere decoder for mimo detection," *Wireless Communications, IEEE Transactions on*, vol. 7, no. 6, pp. 2131–2142, 2008.

[8] J. Fung and S. Mann, "Using graphics devices in reverse: Gpu-based image processing and computer vision," in *Multimedia and Expo, 2008 IEEE International Conference on*. IEEE, 2008, pp. 9–12.

[9] W. Blewitt, G. Ushaw, and G. Morgan, "Applicability of gpgpu computing to real-time ai solutions in games," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 5, no. 3, pp. 265–275, 2013.

[10] W. J. van der Laan, A. C. Jalba, and J. B. Roerdink, "Accelerating wavelet lifting on graphics hardware using cuda," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 1, pp. 132–146, 2011.

[11] C. Xanthis, I. Venetis, A. Chalkias, and A. Aletras, "Mrisimul: A gpu-based parallel approach to mri simulations," *Medical Imaging, IEEE Transactions on*, vol. 33, no. 3, pp. 607–617, March 2014.

[12] S. Grauer-Gray, W. Killian, R. Searles, and J. Cavazos, "Accelerating financial applications on the gpu," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. ACM, 2013, pp. 127–136.

[13] C. Nvidia, "Programming guide," 2008.

[14] S. Cook, *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2013.

[15] G. H. Golub and C. F. Van Loan, *Matrix computations*. JHU Press, 2012, vol. 3.

[16] P. W. Wolniansky, G. J. Foschini, G. Golden, and R. Valenzuela, "V-blast: An architecture for realizing very high data rates over the rich-scattering wireless channel," in *Signals, Systems, and Electronics, 1998. ISSSE 98. 1998 URSI International Symposium on*. IEEE, 1998, pp. 295–300.