

# Parallel Acceleration of Fixed Complexity Sphere Decoder for Large MIMO Uplink System based on GPU

Tianpei Chen, Harry Leib

Department of Electrical and Computer Engineering

McGill University

Montreal, Quebec

October 29, 2014

## **Abstract**

Large MIMO constitutes a principal techniques for next generation wireless communication systems. Detection algorithms for large MIMO tend to require long simulation times and on conventional computer systems, In order to reduce the simulation times for decoders of large MIMO systems. In this paper we propose

to use General Purpose Graphic Processing Units (GPGPU) to implement Fixed Complexity Sphere Decoder (FCSD) for the large scale MIMO system based on the dominant framework of GPGPU-Computed Unified Device Architecture (CUDA). The proposed CUDA-FCSD can achieve up to five times 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 decoder for MIMO system is the Sphere Decoder (SD) that perform as well as a Maximum Likelihood (ML) decoder 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 is 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 com-

putation, rather than caching and flow control. In recent years GPU computing has provided significant advantages in data processing speeds over CPU computations [13]. Computed unified Device Architecture (CUDA) is a programming model for GPGPU that was proposed by Nvidia and is becoming 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 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 results of different influence factors 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:

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

$\mathbf{s} \in \mathbb{C}^{N_t \times 1}$  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  $\mathbb{O}^{N_t}$ .  $\mathbb{E}[\mathbf{s}\mathbf{s}^H] = \mathbf{I}_{N_t}E_s$ ,  $E_s$  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 the C programming extension, with special defined syntax and structure. The serial code executes on the host (CPU) side, and massive parallel data processing is performed on the device (GPU) side, 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.

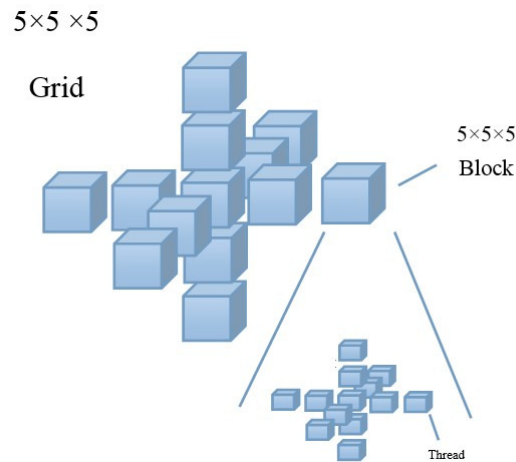


Figure 1: CUDA 3D thread hierarchy

$5 \times 5 \times 5$  grid and each block contains  $5 \times 5 \times 5$  threads

### 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). 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, 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 is on chip memory located at SM and has low latency and limited capacity. Global memory, which is off-chip memory of the GPU with long latency and large capacity. Another two types of the memory, called 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 to threads.



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  $T_{total} = N_{threads} \times N_{threads/SM}$ ,  $N_{SM}$  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.

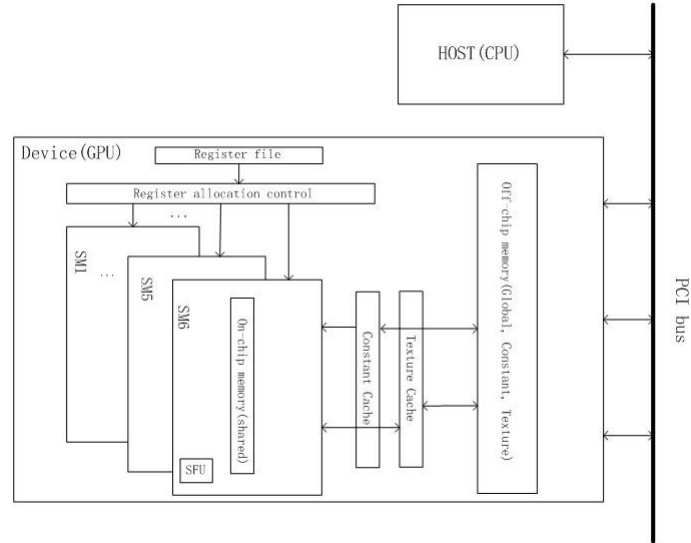


Figure 2: High View of CUDA CPU Microarchitecture

## 4 GPU Based Acceleration of FCSD

In this section we present the parallel implementation of fixed complexity sphere decoder

by GPU, first from (1), the maximum likelihood (ML) solution is given by:

$$s_{ML} = \arg \min_{s \in \mathbb{O}^{N_t}} \|y - \mathbf{H}s\|^2 \quad (2)$$

we consider the array  $N_t = N_r$ , performing QR decomposition to channel propagation matrix

$$\mathbf{H} = \mathbf{Q}\mathbf{R} \quad (3)$$

where  $\mathbf{Q} \in \mathbf{C}^{N_t \times N_t}$  is the unitary matrix,  $\mathbf{R} \in \mathbf{C}^{N_t \times N_t}$ , is the upper triangular matrix,

the basing on [?], the (2) can be changed to

$$\begin{aligned} s_{ML} &= \arg \min_{s \in \mathbb{O}^{N_t}} \|\mathbf{Q}^H y - \mathbf{Q}^H \mathbf{H}s\|^2 \\ &= \arg \min_{s \in \mathbb{O}^{N_t}} \|\mathbf{Q}^H y - \mathbf{R}s\|^2 \end{aligned} \quad (4)$$

Consider the unconstrained estimation of  $s$ ,  $\hat{s} = \mathbf{G}y$ ,  $\mathbf{G} = (\mathbf{H}^H \mathbf{H})^{-1} \mathbf{H}^H$  denotes the zero

forcing equalizer, basing on 3, we have

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

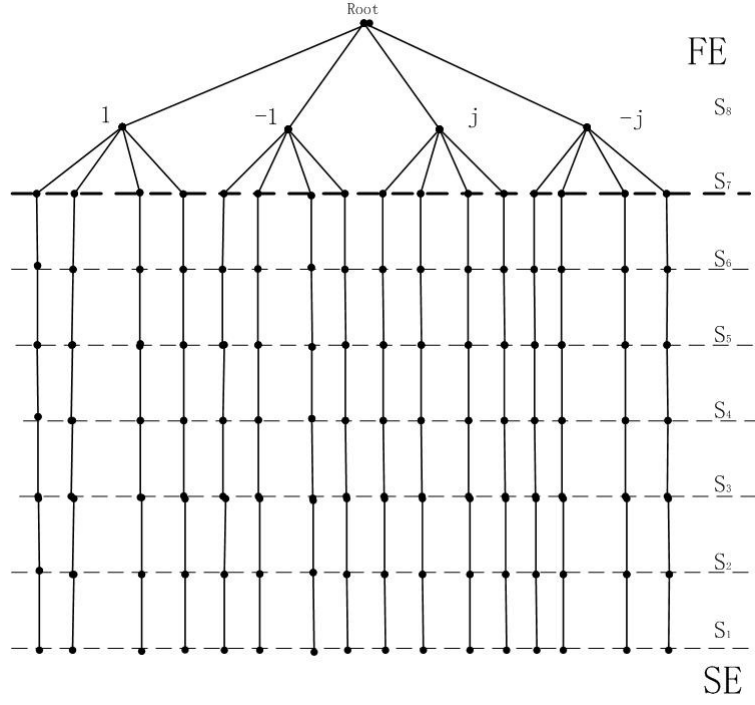


Figure 3: Tree searching of QPSK FCSD for 8x8 MIMO system

Therefore  $\mathbf{Q}^H y = \mathbf{R} \hat{s}$ , the 4 can be changed to

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

the Fixed Complexity Sphere Decoder works as a constant number of multiple path tree searching. As shown in Figure 3, just like SD the FCSD also performs depth first searching, at the first  $\rho$  level the FCSD search all the possible signal symbols  $\tilde{s}_{N_t} \tilde{s}_{N_t-1} \dots \tilde{s}_{N_t-\rho+1}$  in constellation  $\mathbb{O}^{N_t}$ , which is called Full Expansion(FE) level, the remain  $N_t - \rho$  levels of symbols, the FCSD performs decision feed back equalization (DFE), from (6) the solution

of FCSD can be given:

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

$i \in [1, 2 \dots N_t]$ ,  $r_{ij}$  denotes the element at the  $i$ th row and  $j$ th column of the upper triangular matrix  $\mathbf{R}$ , this process is called Single Expansion (FE). thus the FCSD solution  $s_{FCSD} = [s_1, s_2 \dots s_{N_t}]$  can be expressed as:

$$s_{FCSD}^i = \begin{cases} \in \mathbb{O}^\rho & i = N_t, N_t - 1, \dots, N_t - \rho + 1 \\ \mathbb{Q}[\hat{s}_i + \sum_{j=i+1}^{N_t} \frac{r_{ij}}{r_{ii}}(\hat{s}_j - s_{FCSD}^j)] & i = N_t - \rho, N_t - \rho - 1, \dots, 2, 1 \end{cases}$$

$\mathbb{Q}[]$  denotes the signal constellation based quantization. At the post processing stage,

FCSD takes the symbol vector candidate with the minimum Euclidean metric  $E_{metric}$ :

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

The  $\tilde{s}$  denotes the symbol vector candidate, the total number of candidates is:

$$M^\rho \quad (9)$$

## 4.1 CUDA-FCSD Pre-Prossing

As shown in figure.4, each path of the FCSD searching has a serial execution nature, in order to avoid error propagation of decoder, FCSD ordering was applied to the propagation channel based on post processing signal to noise ratio [?], which can be expressed as:

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

$\varphi_m$  denotes the post processing signal to noise ratio of the  $m$ th data stream,  $(\mathbf{H}^H\mathbf{H})_{mm}^{-1}$  denotes the diagonal elements of the inversion of  $HHH$  matrix  $\mathbf{H}^H\mathbf{H}$ , thus when the SNR is determined, the post processing SNR is only determined by  $(\mathbf{H}^H\mathbf{H})_{mm}^{-1}$ , 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 is detected first, that is, the data stream with the smaller  $\varphi$  is detected earlier. At SE stage, because of the single searching path of the symbol, 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 earlier in order to avoiding the error propagation, in conclusion the FCSD ordering works iteratively and obeys the following rule: At the  $j$ th step ( $j = 1, 2 \dots N_t - 1$ ),  $s_p$  denotes the symbol to

be detected, where

$$p = \begin{cases} \arg \max_k (\mathbf{H}_j^H \mathbf{H}_j)_{kk}^{-1} & FEstage \\ \arg \min_k (\mathbf{H}_j^H \mathbf{H}_j)_{kk}^{-1} & SEstage \end{cases}$$

$\mathbf{H}_j \in \mathbb{C}^{N_r \times (N_t - j + 1)}$  denotes the renewed channel matrix, at the  $j$ th iteration with the column corresponding to previous detected data streams removed from  $\mathbf{H}_{j-1}$ . the unconstrained estimation  $\hat{s}$  in 5 is given by:

$$\hat{s} = (\mathbf{H}^H \mathbf{H})^{-1} \mathbf{H}^H y \quad (11)$$

Although the FCSD pre-processing has a serial nature and can not be vectorized, the computational consumptive operations 4.1 11 can be implement by the CUDA Basic Linear Algebra Subprograms(cuBLAS), the matrix inverse(Gaussian Jordan Elimination) and cholesky factorization(Row Major Factorization) are also implemented by GPU, as shown in Table 1.

Table 1: Parallel Computation Operation of FCSD Pre-processing

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

Figure 4 shows the block diagram of the CUDA-FCSD implementation, as mention in section 3, our implementation is based on heterogeneous programming principle, the flow control and logical operations are reside at the host side, the compute intensive works is implemented at device side.

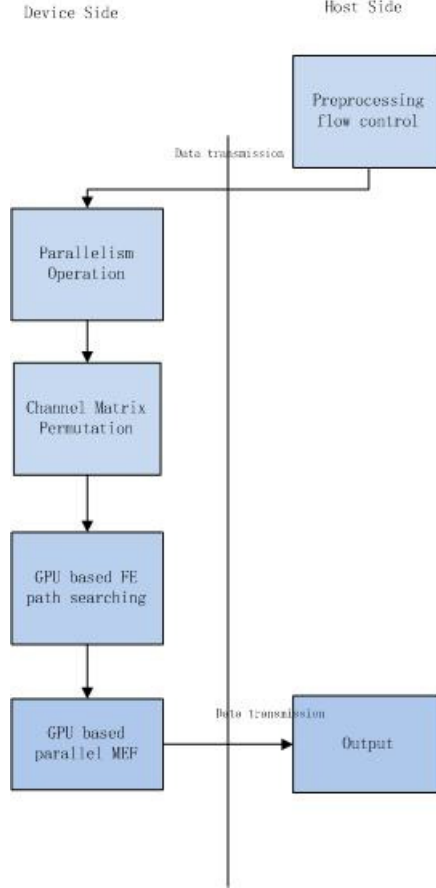


Figure 4: Block Diagram of CUDA-FCSD Implementation

## 4.2 Parallel Acceleration of Blocked-Paths Searching

We consider  $16 \times 16$  MIMO system, the modulation scheme is 16QAM, the path we need to search is  $\mathcal{O}[M^\rho] = 16^3 = 4096$ ,  $\rho = \lceil \sqrt[3]{N_t} - 1 \rceil$ , as mentioned in section 3 the maximum

number of threads that can be used 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. For each path, decision feedback path searching has a serial nature, therefore we match 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. At each path, we perform the following data operation:

1. *Decision Feedback Equalization*

$$\begin{aligned}
s_i^k &\in \mathbb{O}^\rho \quad \text{if } i = N_t, N_t - 1, \dots, N_t - \rho + 1 \quad FE \\
s_i^k &= \hat{s}_i^k + \sum_{j=i+1}^{N_t} \frac{r_{ij}}{r_{ii}} (\hat{s}_j^k - s_j^k) \quad \text{if } i = N_t - \rho, N_t - \rho - 1, \dots, 2, 1 \quad SE
\end{aligned} \tag{12}$$

$s_j^k$  denotes the  $j$ th symbol of the  $k$ th symbol vector candidate.

2. *Euclidean Metric Calculation*

$$E_u^k = \sum_{i=1}^{i=N_t} \sum_{j=i}^{j=N_t} r_{ij} (\hat{s}_j - s_j) \tag{13}$$



$E_u^k$  denotes the  $k$ th Euclidean metric of the corresponding symbol vector candidate  $\tilde{s}^k$ .

Then we take the symbol vector  $s^\theta$  as the solution, where

$$\theta = \arg \min_m (E_u^m) \quad m = 1, 2 \dots M^\rho \quad (14)$$

#### 4.2.1 Data Preparation

There are three major factors that will influence the configuration of the data set configuration in memory space.

1. Amount of Reading&Writing operation per thread.
2. Program scope of the data.
3. Storage Requirement of data.

The memory reading&writing (R/W) strategy of GPU is the crucial factor that influence the application performance because the GPU is data processing intensive and only has small cache for special memory space(constant, texture), another consideration is the validation life circle of data, local memory which is invoked in kernel function is valid 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), so

that for our application the configuration of memory is determined by the capacity requirement of the data. Here in table 2 we present the amount of R/W operations and the storage that needed for critical data variables in FCSD path searching. The amount of the R/W operation is calculated based on the (12) and ((refEu metric), for complex value  $(\mathbf{R}, \hat{s}, s_{potentialmatrix}, s_{kernel})$  is stored use *float2* which counts 8 bytes, floating point data set  $E_u$  is stored in *float*, which counts for 4 bytes,  $s_{index}$  is the integer list which is stored use *int*, which counts 1 byte.

At host side we set 4 vector data set that are stored linearly in global memory which has program scope for both host side and device side,  $s_{potentialmatrix}$  is used to store the  $M^\rho \times N_t$  complex matrix that each row stores one symbol vector candidate and whose valid scope is through the whole kernel function.  $s_{index}$ , which 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  $M^\rho$  floating point vector that is used to store the Euclidean metric of all the symbol vector candidates, obviously the valid scope of it is for the whole kernel function.  $s_{kernel}$  is a complex vector used to store the FCSD solution which length is  $N_t$  and has to be transferred to host side, thus it is also needed to be stored in global memory.

On the other hand, upper triangular matrix  $\mathbf{R}$  in (6) and unconstrained estimation  $\hat{s}$  in (5) requires a large amount of R operations and are read only. (for the  $16 \times 16$  16QAM

MIMO system the R/W of  $\mathbf{R}$  and  $\hat{s}$  are about 370 and 266 per thread), these two data set need small size of storage(for the  $16 \times 16$  16QAM MIMO system 1088 bytes for  $\mathbf{R}$  and 128 bytes for  $\hat{s}$  ) 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.

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

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

#### 4.2.2 Memory Access 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.

Global memory is implemented by dynamic random access memory (DRAM) whose R/W is extremely slow process, modern DRAM R/W data 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 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  $s_{potentialmatrix}$  as an example to illustrate how we coalesce the the global memory, we store  $s_{potentialmatrix}$  in one pitch of linear global memory, with each row store one  $N_t = 16$  complex symbol vector solution candidate of  $\hat{s}_{FCSD}$  in (7), totally there are  $M^p = 4096$  rows, this matrix is stored in column major. As mentioned before we operate one path in 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  $32 \times 8 = 256bytes$ , counts for two times of DRAM access. Another available resource for memory bandwidth increase is to make use of share memory, for all the transient variables that used for accumulator in (12) and (13), because of the small capacity consumption and large amount of R/W operations.

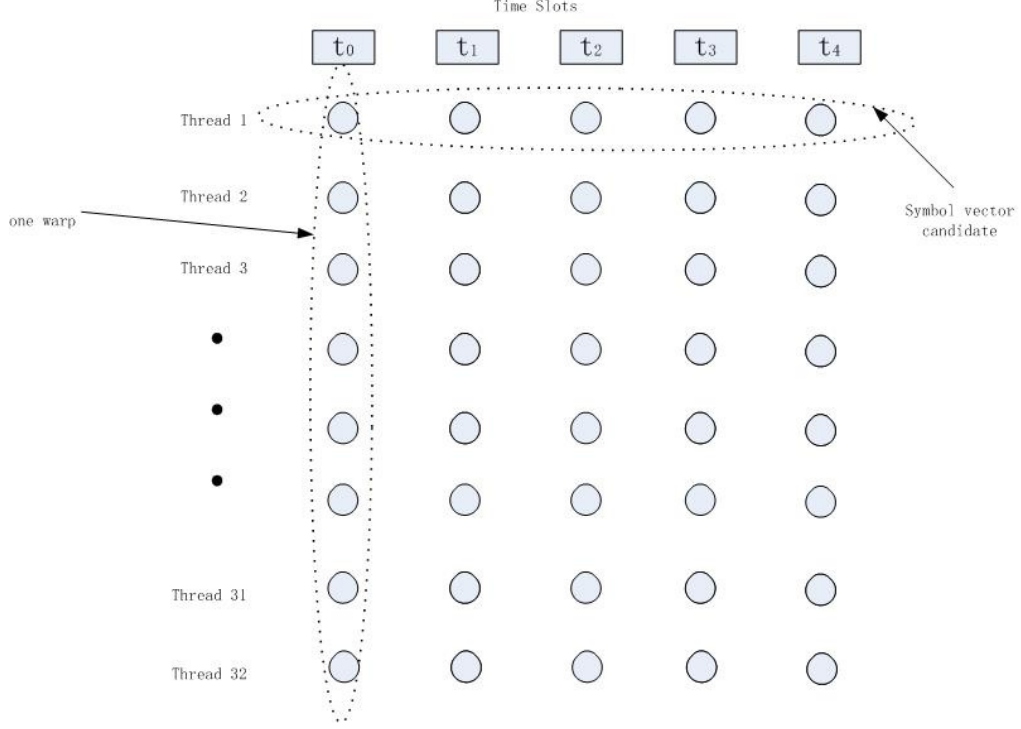


Figure 5: Global Memory Access Pattern of  $s_{potentialmatrix}$

#### 4.2.3 Data Transfer Minimization

The data transfer between the host side 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  $y$  in (1) as well as the index list  $s_{index}$  at FE stage, we integrate the post processing (13) into path searching kernel function to avoid the transferring of  $E_u$  in (13).

### 4.3 Going to Larger MIMO System

Consider the uplink  $32 \times 32$  MIMO system with 16QAM modulation scheme, from 9, the path number that need to be searched is  $16^5 = 1048576$ , as mentioned in section (3), the maximum number of threads that can be execute in kernel function of GeForce GTX 760 is 12288, therefore we need to consider serial the one kernel into a certain times of 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.

## 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

## 5.2 Software

CUDA Driver Version / Runtime Version 6.5 / 6.0

Nsight Eclipse Version 6.5

## 5.3 Performance and Evaluation

Table 3: Speedup Performance of different MIMO Array size

Array size	Time/s			Speedup	
	GTX 760	modigliani	monet	GTX/modigliani	GTX/monet
8	0.46	0.09	0.08	0.20	0.17
16	1.56	3.2	3.57	2.05	2.29
20	31	74	81	2.39	2.61
25	38	112	122	2.94	3.21
32	611	2836	3033	4.64	4.96

We test 5 array sizes and make comparison between GTX 760 and two kinds of CPU, all the operation times we test is based on 100 channel realizations, the transmitted data is generated randomly and modulated by 16 QAM modulation scheme, as we can see in

3, for small size array, like  $8 \times 8$ , GPU has no advantage over CPU, for small array size, the number of path that need to be searched is relatively small, the CPU has already had a good performance, there is a short time for GPU to "warm up", for data transferring, launch kernel function and host device synchronization, further more, for small number of threads, the memory R/W latency is not well hided. When the array size grows large, a large parallelism is achieved so that the GPU begin to display its data processing power, and with the array size increasing, this speedup tends to be much more considerable.

## 6 Conclusion

In this paper, we present the GPU implementation of fixed complexity sphere decoder for a large MIMO system, which is a time consuming task for traditional platform, we have reached up to nearly 5 times acceleration of FCSD and this speedup grows higher with the increasing array size. 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 the simulation and consumptive computational work of large MIMO area.



## 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.