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

Tianpei Chen and Harry Leib
Department of Electrical and Computer Engineering, McGill University, Montreal, Quebec
tianpei.chen@mail.mcgill.ca harry.leib@mcgill.ca

***Abstract*— Large MIMO constitutes a principal technology for next generation wireless communication systems. Detection algorithms for large MIMO schemes tend to require long simulation times on conventional computer systems. In order to improve simulation efficiency of decoders in large MIMO systems, we propose to use General Purpose Graphic Processing Units (GPGPU). This paper considers the implementation of a Fixed Complexity Sphere Decoder (FCSD) for large scale MIMO systems using the framework of GPGPU-Computed Unified Device Architecture (CUDA). The proposed CUDA-FCSD can achieve considerable speedups over conventional CPU implementation while providing the same Bit Error Rate (BER) performance. Furthermore the proposed algorithm demonstrates the great potential of GPGPU to accelerate the simulation of large MIMO systems.**

Keywords: Large MIMO, fixed complexity sphere decoder (FCSD), general purpose graphic processing units (GPGPU), computed unified device architecture (CUDA)

I. Introduction

A present challenge in the wireless communication area is to bridge 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) technology is of immense research interest since it can help solving this problem owing to several advantages it offers. The core idea of MIMO is to use multiple antennas at the transmitting and receiving sides, that provide parallel data pipes, resulting in spatial multiplexing and diversity gains [1].

Large scale MIMO (also named massive MIMO) systems employ tens to hundreds of low-power low-price antennas at the base station, that serve many terminals at the same time [2]. Large scale MIMO can achieve the full potential of conventional MIMO systems, as well as provide additional reduction of latency and improvement of system robustness. The detector is one of the key components of MIMO systems. A well known detector for MIMO systems is the Sphere Decoder (SD) that performs as well as a Maximum Likelihood detector (MLD), with a dramatic complexity reduction [3] [4]. There are two major problems with the traditional SD decoder: complexity depending on Signal to Noise Ratio (SNR), and sequential nature, which conflict with the requirement of a constant data processing throughput and parallel implementation for practical systems. The Fixed Complexity Sphere Decoder (FCSD) overcomes these problems of the SD decoder by searching the solution along a fixed number of independent paths [5].

General Purpose Graphic Processing Units (GPGPU) computing is a common trend in industry and research in areas that need massive computations, such as computer vision [6], computer games [7], signal processing [8] [9], and finance [10]. Compared to a CPU, a GPU can naturally cope with computational intensive tasks, since most of the hardware resources are allocated for data processing and parallel computation, rather than caching and flow control [11]. In recent years GPU computing has provided significant advantages in data processing speeds over CPU computations [12]. The Computed unified Device Architecture (CUDA) [11] [12] 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 different applications.

In this paper, a GPU based implementation of Fixed Complexity Sphere Decoder (FCSD) is proposed, which is referred to as CUDA-FCSD. A heterogeneous programming solution for FCSD preprocessing and integrated blocked parallel path searching and decision making are presented. Using this implementation the speedup as well as Bit Error Rate (BER) performance of different sizes of MIMO systems and signal constellations are analyzed. The proposed CUDA-FCSD implementation demonstrates a great potential for large MIMO systems.

The rest of the paper is organized as follows. Section II briefly introduces CUDA, the GPU architecture and the programming model. In section III, we present details of the implementation of CUDA-FCSD. In section IV, the speedup and bit error rate (BER) performance of CUDA-FCSD are considered. Finally, conclusions are drawn in section V.

II. CUDA Programming Model and GPU Architecture

A. Single Instruction Multiple Thread Model

The 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. Such an approach 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, where they can synchronize their behavior and communicate. Fig. 1 illustrates a simple CUDA memory hierachy model.

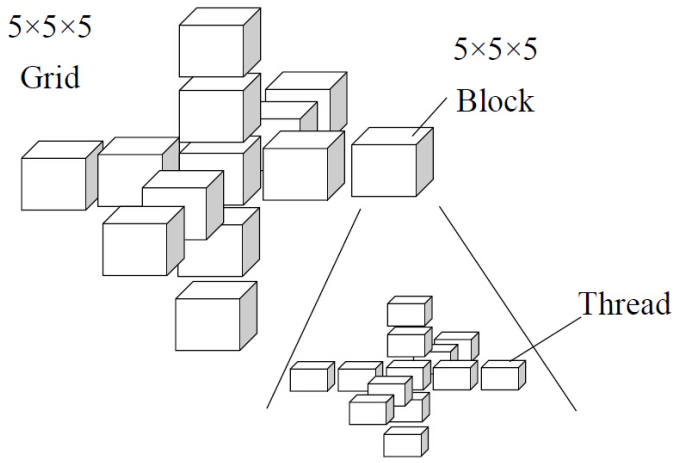

Fig. 1. CUDA 3D memory hierachy $5 \times 5 \times 5$ grid, with each block composed of $5 \times 5 \times 5$ threads. The threads can be organized into one-dimensional, two-dimensional or three-dimensional blocks. Similarly, blocks can also be grouped into a one-dimensional, two-dimensional, or three-dimensional grid.

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

B. Mircroarchitecture of GPU

We present the GPU-GeForce GTX 760 microarchitecture in Fig. 2 as an example. The GPU works as a coprocessor system, consisting of Stream Multiprocessors (SMs), which perform multiply-add arithmetic operations. Furthermore, each SM has the special functional units (SFU) that can perform more complex operations such as trigonometric functions and reciprocal square root. There are five types of memories: registers, shared memory, global memory, constant memory and texture memory. Registers are allocated dynamically and privately to threads, and provide most rapid access speeds. The shared memory, also called on-chip memory, is located on the SMs and can be shared among the threads in one block. Shared memory has low latency and limited capacity. Global memory, which is off-chip has large latency and large capacity. The other two types of memories, constant memory and texture memory, are located off-chip, but cached. Therefore the access speed of these two type of memory space is much faster than that of global memory. The variables invoked by *kernel* functions can be stored in either of the these five types of memories.

Because of the resource limitation of SM, the total number of threads that can be defined in one block is limited. On current CUDA supported GPUs, one block contains up to 1024 threads. Thus the total number of threads (T_{total}) we can use is $T_{total} = N_{SM} \times N_{threads/SM}$, where N_{SM} and $N_{threads/SM}$ denote the number of SMs and number of threads per SM.

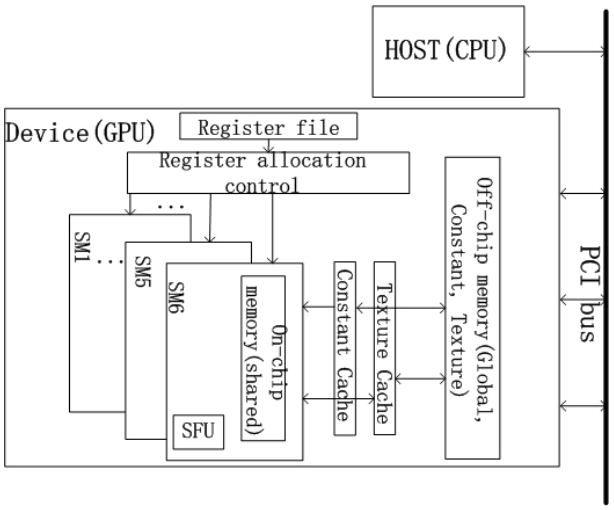

Fig. 2. High View of CUDA CPU Microarchitecture The GeForce GTX 760 has 6 SMs and the maximum number of threads allowed on each SM is 2048. Thus the total number of threads that can work in parallel is $2048 \times 6 = 12288$.

III. GPU Based Acceleration of FCSD

A. 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. Using a discrete time model, $\mathbf{y} \in \mathbb{C}^{N_r \times 1}$ is the received symbol vector written as:

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

where $\mathbf{s} \in \mathbb{C}^{N_t \times 1}$ is the transmitted symbol vector, with components that are mutually independent and taken from a signal constellation $\mathbb{O}$ (4-QAM, 16-QAM, 64-QAM) of size M. All the possible transmitted symbol vectors $\mathbf{s} \in \mathbb{O}^{N_t}$, with $\mathbb{E}[\mathbf{s}\mathbf{s}^H] = \mathbf{I}_{N_t} E_s$, where E_s denotes the symbol average energy, and $\mathbb{E}[\cdot]$ denotes the expectation operation. Furthermore $\mathbf{H} \in \mathbb{C}^{N_r \times N_t}$ denotes the Rayleigh fading channel propagation matrix with independent identity distributed (i.i.d) circularly

symmetric complex Gaussian components of zero mean and unit variance. Finally, $\mathbf{n} \in \mathbb{C}^{N_r \times 1}$ is the additive white Gaussian noise (AWGN) vector with zero mean components and $\mathbb{E}[\mathbf{n}\mathbf{n}^H] = \mathbf{I}_{N_r} N_0$, where N_0 denotes the noise power spectrum density, and hence $\frac{E_s}{N_0}$ is the signal to noise ratio (SNR).

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

B. Fixed Complexity Sphere Decoder

From (1), the maximum likelihood detector (MLD) for a MIMO system is specified by:

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

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

$$\mathbf{H} = \mathbf{Q}\mathbf{R}, \tag{3}$$

where $\mathbf{Q} \in \mathbb{C}^{N_t \times N_t}$ is a unitary matrix and $\mathbf{R} \in \mathbb{C}^{N_t \times N_t}$, is an upper triangular matrix [13], we can write (2) as:

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

Using the unconstrained estimator of $\mathbf{s}$, $\hat{\mathbf{s}} = \mathbf{G}\mathbf{y}$, $\mathbf{G} = (\mathbf{H}^H\mathbf{H})^{-1}\mathbf{H}^H$, and (3), we have

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

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

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

Similarly to SD the FCSD also performs depth first searching. At the first ρ node levels[1] the FCSD searches all the possible signal symbols in constellation $\mathbb{O}^{N_t}$, a process called the Full Expansion (FE) stage. Let $\mathbf{s}^F$ denote FCSD symbol vector candidates. For the symbols $s^F_{N_t}, s^F_{N_t-1} \dots s^F_{N_t-\rho+1}$, the FCSD has M branch expansions at each symbol node. As shown in Fig. 3, there are $\rho = 2$ FE node levels and each node level has $M = 4$ branches. According to (6), we have:

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

$i \in [1,2 \dots N_t]$, 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 employs decision feedback, and therefore there are only one 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, and the FCSD works as a constant number of multiple path tree searching algorithm.

[1] when $N_r = N_t$ the FCSD can achieve the same diversity as the MLD if $\rho \geq \sqrt{N_t} - 1$ [5]

In conclusion the FCSD symbol vector candidate $s^F_i = [s_1, s_2 \dots s_{N_t}]$ can be expressed as:

$$\begin{aligned} &FE \quad stage: \\ &s^F_i \in \mathbb{O}^{\rho}, \quad if \quad i = N_t, N_t - 1, \dots N_t - \rho + 1 \\ &SE \quad stage: \\ &s^F_i = \mathbb{Q}[\hat{s}_i + \sum_{j=i+1}^{N_t} \frac{r_{ij}}{r_{ii}}(\hat{s}_j - s^F_j)], \quad if \\ &i = N_t - \rho, N_t - \rho - 1, \dots 2, 1 \end{aligned} \tag{8}$$

where $\mathbb{Q}[\cdot]$ denotes the operation of signal constellation quantization. At the post processing stage, the FCSD compares the Euclidean distance of all the symbol vector candidates

$$E_{metric} = ||\mathbf{R}(\hat{\mathbf{s}} - \mathbf{s}^F)||^2, \tag{9}$$

where $\mathbf{s}^F$ denotes the symbol vector candidate. The symbol vector candidate with minimum E_{metric} is chosen as the solution. The total number of candidates is M^{ρ}.

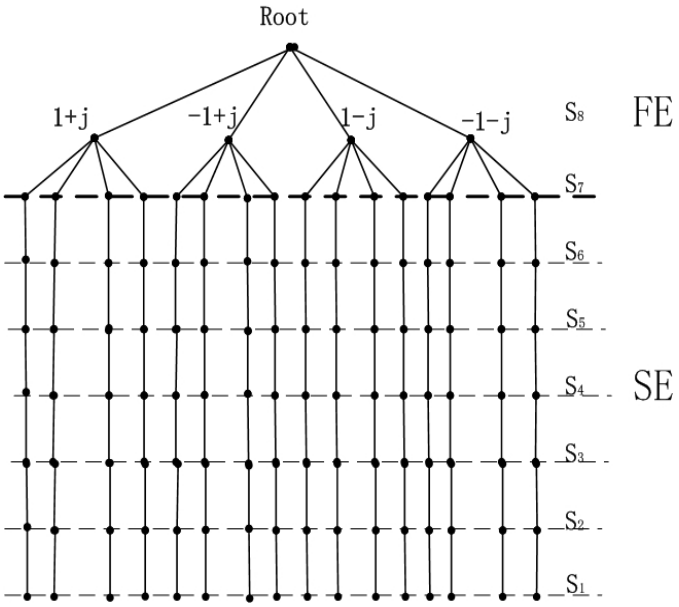

Fig. 3. Tree searching of 4-QAM FCSD for 8×8 MIMO system. The total number of branches is 4^2

C. CUDA-FCSD Preprocessing

A FCSD channel ordering strategy is used in order to avoid error propagation in the serial path searching process. This channel ordering strategy is based on post processing signal to noise ratio [14]

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

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 inverse of $\mathbf{H}^H\mathbf{H}$. The post processing SNR depends only on $(\mathbf{H}^H\mathbf{H})^{-1}_{mm}$. From [5], at the FE stage the "weakest" data stream, that is the data stream with the smallest φ_m, is detected first. At the SE stage, because of

the single searching branch of the node, the performance is tightly related to the post processing SNR. Therefore at the SE stage the data stream with the largest 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

$$p = \begin{cases} arg\max_k(\mathbf{H}_j^H\mathbf{H}_j)_{kk}^{-1} & FE \quad stage \\ arg\min_k(\mathbf{H}_j^H\mathbf{H}_j)_{kk}^{-1} & SE \quad stage \end{cases} \tag{11}$$

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

As mentioned in section II, our implementation is based on the heterogeneous programming principle, employing a mix of CPU and GPU technology. The flow control and logical operations reside at the host side, while the compute intensive work is implemented at the device side. Although the FCSD preprocessing has a serial nature and can not be vectorized, the computational intensive operations can be performed by the CUDA Basic Linear Algebra Subprograms (cuBLAS) [15] and corresponding GPU program implementations, as shown in Table I.

D. Parallel Acceleration of Paths Searching

Since the decision feedback searching paths have a serial nature, we match one path to one thread. In order to have the largest number of threads that can be parallelized, we use one dimensional blocks for widest expansion, and organize all the paths into several such blocks. For example for a 16×16 MIMO system with 16QAM, the number of paths we need to search is $M^\rho = 16^3 = 4096$, since $\rho = \lceil\sqrt{N_t}-1\rceil$, where $\lceil x\rceil$ represent the smallest integer greater than or equal to x. For largest SM occupancy we use 4 blocks for the path searching kernel where each block has 1024 threads which is the maximum number allowed per block. At each FCSD searching path, the following operations are performed in one thread:

- *Path searching*
  As in (8).
- *Euclidean Distance Calculation*

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

where E_u^k denotes the Euclidean distance of the kth symbol vector candidate. We integrate the post processing (12) into the path searching process (8) to reduce the iteration times per thread.

In order to avoid low speed host-device data transfer, we reduce data transfers as much as possible. As shown in Fig. 4 we perform data transfer only two times. For host to device, the propagation channel matrix $\mathbf{H}$, received symbol vector $\mathbf{y}$, as well as the sub symbol vector index list s_{index} at FE stage, are transferred. For device to host, the FCSD symbol vector solution $\mathbf{s}^F$ is transferred.

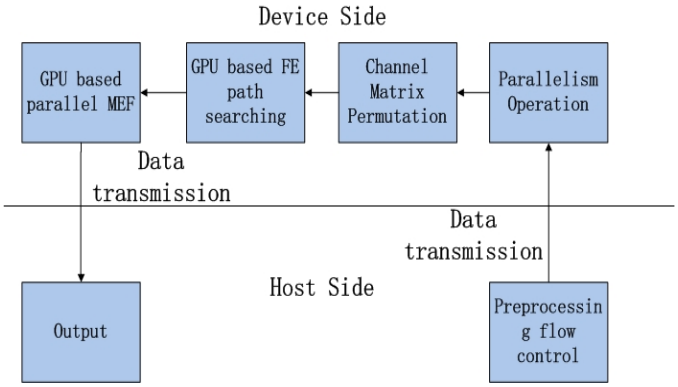

Fig. 4. Block Diagram of CUDA-FCSD Implementation

1) Data Preparation: There are three major factors that influence the configuration of the data sets in memory space.

- Number of Reading Writing (R/W) operations
- Valid program scope
- Storage occupancy

Unlike CPUs that have a flat memory model (the CPU core can access any memory location by a near clock rate speed) GPUs are computation intensive. Thus the memory R/W operations are time consuming that influence performance. Furthermore different memory types have different program scopes. Local memory, which is invoked in *kernel* functions is valid only in one thread. Shared memory is valid in one block. Constant, texture and global memories are valid both at host side and device side. On the other hand different from global memory, the constant and share memory are all capacity limited. Thus the memory occupancy is an important factor that needs to be considered.

At the host side we set four data sets that are stored in global memory. The $M^\rho \times N_t$ complex matrix s_{pm} that stores in each row one symbol vector candidate and its program scope should be during the whole *kernel* execution. The matrix s_{index} is used to store the index of all the possible sub-symbol vectors for FE stage using the full factorial method and requires a $\rho \times M^\rho$ integer matrix space. The M^ρ dimensional floating point vector is used to store the Euclidean distance of all the symbol vector candidates. The N_t dimensional complex vector s_{kernel} is used to store the FCSD solutions and has to be transferred to host side.

On the other hand, the upper triangular matrix $\mathbf{R}$ and the unconstrained estimator $\hat{\mathbf{s}}$ of (5) are read only and require large numbers of read operations. Since these two data sets need small sizes of storage, the constant memory, which is cached and read only, is used to store them.

2) Memory Accesss Pattern: When considering a large MIMO system, the large number of required R/W operations make the memory bandwidth a major bottleneck for performance. Although the off-chip memory theoretical bandwidth is extremely high (the theoretical bandwidth of GeForce GTX 760 is 192GB/s), in practice it can only be achieved by a proper R/W strategy.

Global memory access is implemented by a dynamic

TABLE I
PARALLEL COMPUTATION OPERATION OF FCSD PREPROCESSING

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

random access memory (DRAM) whose R/W is an extremely slow process. In a modern DRAM the R/W operations are performed by accessing a piece of consecutive memory location. As mentioned in section II, the warp is the basic schedule unit of a device, and memory space is accessed warp by warp. Thus the most effective global memory access pattern is to ensure a warp of threads can access a consecutive memory space. For the GeForce GTX 760, the DRAM will access a consecutive memory space of 128 bytes at one time. In Fig. 5 we present the global access pattern of s_{pm} as an example to illustrate global memory access pattern.

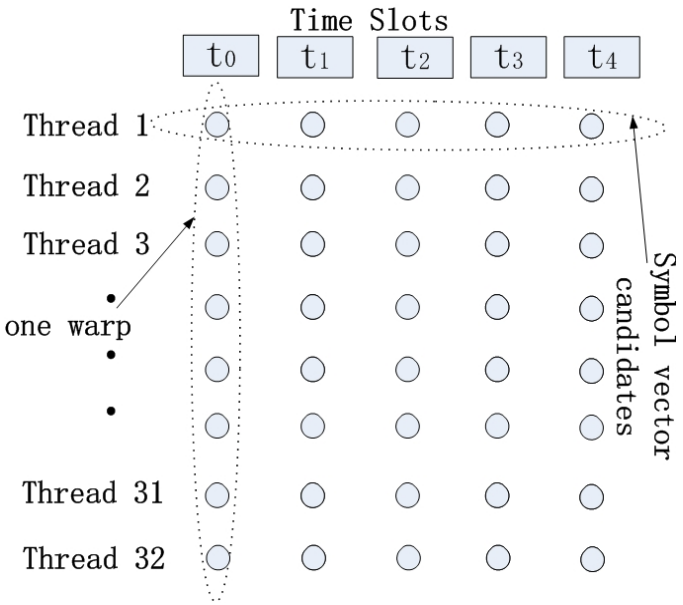

Fig. 5. Global Memory Access Pattern of s_{pm}

The complex matrix s_{pm} is stored in one piece of linear global memory, where each row represents one N_t complex symbols solution candidates of $\hat{\mathbf{s}}^F$ in (7). There are M^ρ rows, and this matrix is stored in column major (first column in the first piece of memory space and the second in the second piece of memory space etc.). As mentioned before, one path searching is performed by one thread, and therefore the address distance between two consecutive symbols in one thread is M^ρ. However for consecutive threads in one warp, the address they reach is continuous as shown in Fig. 5. Every 32 threads in one warp can confirm R/W operations to a piece of memory that is consecutive, and reduce DRAM access as much as possible resulting in a better memory bandwidth.

All the thread level temporary data that is used in (8), (12), and the symbol vector candidate during the path searching process, are stored in registers to get near clock rate access speed.

E. Larger MIMO Systems and Constellation Sizes

There are two problems that need to be considered for large MIMO systems and constellation size:

- Thread resource limitation: The number of threads that can be executed in parallel is limited. When the size of a MIMO system and constellation increases, it becomes impossible to parallelize all the searching paths in one *kernel* execution. For example, in a 32×32 MIMO system with 16QAM modulation, the number of paths that need to be searched is $M^\rho = 16^5 = 1048576$.
- Global memory limitation: Large MIMO systems and large signal constellations require a large global memory space to store the symbol vector candidates. For example, for the systems 36×36 with 16QAM, 144×144 with 4QAM, 20×20 with 64QAM, the required space of storing s_{pm} of section III-D.1 exceeds the limit (2 GB).

One solution to these problems is to employ partial parallelism. The detection task is divided into several *kernel* executions, where for each, only one potential symbol vector solution, as well as its corresponding Euclidean distance are preserved. This allows the global memory space to be reused to avoid exceeding its limit.

IV. SIMULATION RESULTS

A. Environment

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 physical cores, 3.20 GHz CPU clock rate, 8 GB RAM.
- *Central Processing Units-Monet* : Intel Core I7-3rd generation, 6 physical cores, 3.20 GHz CPU clock rate, 32 GB RAM.

2) Software:

- CUDA Driver Version / Runtime Version 6.5 / 6.0
- Integrated software environment for source code editing, automatic building and debugging: Nsight Eclipse Version 6.5

TABLE II
SPEEDUP PERFORMANCE OF DIFFERENT MIMO SYSTEMS USING 4 QAM

Array size	Time/s			Speedup	
	GeForce GTX 760	Modigliani	Monet	$\vartheta_{GTX760}/\vartheta_{Modigliani}$	$\vartheta_{GTX760}/\vartheta_{Monet}$
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 III
SPEEDUP PERFORMANCE OF DIFFERENT MIMO SYSTEMS USING 16 QAM

Array size	Time/s			Speedup	
	GeForce GTX 760	Modigliani	Monet	$\vartheta_{GTX760}/\vartheta_{Modigliani}$	$\vartheta_{GTX760}/\vartheta_{Monet}$
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 IV
SPEEDUP PERFORMANCE OF DIFFERENT MIMO SYSTEMS USING 64 QAM

Array size	Time/s			Speedup	
	GeForce GTX 760	Modigliani	Monet	$\vartheta_{GTX760}/\vartheta_{Modigliani}$	$\vartheta_{GTX760}/\vartheta_{Monet}$
8×8	12.28	10.69	13.38	0.87	1.09
16×16	468.92	2066.08	2689.00	4.41	5.73

B. Performance Evaluation

Monte-Carlo simulations were conducted for MIMO systems of different sizes and signal constellation. In this work the speedup is defined as $\vartheta_{GPU}/\vartheta_{CPU}$, where ϑ_{GPU} and ϑ_{CPU} denotes the number of symbols detected on GPU and CPU per second. Comparison is made between the GeForce GTX 760 and two types of CPUs, with speedup performance presented in Tables II, III and IV. All the operation times correspond to 1000 channel realizations.

From Table II, we see that for small MIMO systems of 8×8, 16×16, and 32×32 with 4QAM, the speed performance of GPU implementation is worse than that of CPU. The number of paths of these MIMO configurations that need to be searched is relatively small[2]. The CPUs are designed for serial code execution with special hardware (branch prediction units, multiple caches, etc.) that provide extremely good performance, while the GPUs can achieve good performance only when they are utilized in a massive parallel manner. Furthermore there is a short time for GPU to "warm up", for host-device data transmission, *kernel* function launching and host device synchronization. When comes to larger MIMO systems of 48×48, 64×64, 72×72, a large parallelism is achieved allowing the GPU to begin displaying its data processing power.

In Tables III and IV, 16QAM and 64QAM are considered. We see that speedups larger than 1 can be achieved with GPU implementation also with smaller MIMO systems. For a 16×16 system with 16QAM modulation, GTX 760 provides a 1.18 speedup over Modigliani and 1.57 speedup over Monet. For a 16×16 system with 64QAM, GTX 760 provides a 4.41 speedup over Modigliani and 5.73 speedup over Monet. The nature of FCSD is such that for lager signal constellations, massive parallelism can be reached also for small MIMO systems that are not extremely large.

It can be observed from Tables II, III and IV, that the GPU implementation shows its speed advantage over CPU implementation for increasing size of MIMO systems and signal constellations, indicating that GPUs provide advantages for large MIMO system with large signal constellations.

In Fig. 6 we present Monte Carlo simulation results for bit error rate (BER) performance, using the GPU implementation of different MIMO system configurations and constellation sizes. We consider uncoded complex MIMO systems with *M*QAM, where each antenna transmits different streams of binary data generated randomly and independently. The results have been obtained by using 100000 channel real-

[2] 8×8 MIMO system with 4QAM has 16 paths, 16×16 MIMO system with 4QAM has 64 paths and 32×32 MIMO system with 4QAM has 1024 paths

izations with at least 500 symbol errors accumulated. The simulation results show that FCSD-CUDA provides the same BER performance as its CPU version [5].

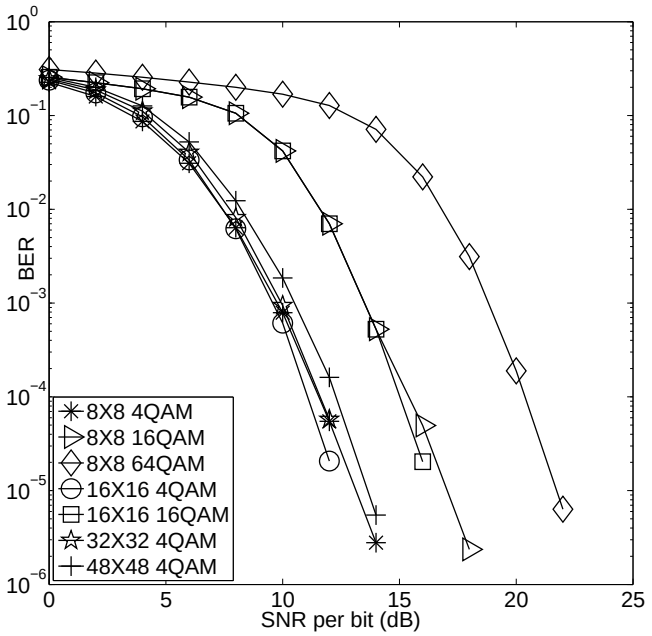

Fig. 6. BER performance for MIMO systems with various modulation schemes using the GPU implementation

V. Conclusions

This paper presents a GPU implementation of the fixed complexity sphere decoder (FCSD) for large MIMO systems. In order to exploit the computational capability of GPU in the simulation of large MIMO systems, the utilization of a heterogeneous programming model, accounting for resource limitations and memory configuration must be considered. The simulation results show that a considerable acceleration of the GPU implementation of FCSD can be reached for large MIMO systems and signal constellation sizes over CPU implementation, while maintaining the same BER performance. Since the GPU and CPU can work independently, the data processing power of GPU computing is an additional advantages provided by the traditional simulation platforms. This shows the potential of GPU computing to reduce the time associated with intensive simulations of 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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

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

[14] 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.

[15] C. NVIDIA, "Basic linear algebra subroutines (cuBLAS) library," 2013.