Forty-Seventh Annual Allerton Conference
Allerton House, UIUC, Illinois, USA
September 30 - October 2, 2009

Highly Parallel Decoding of Space-Time Codes on Graphics Processing Units

Kalyana C. Bollapalli*, Yiyue Wu†, Kanupriya Gulati*, Sunil Khatri* and A. Robert Calderbank†
* Department of Electrical & Computer Engineering, Texas A&M University, College Station, TX 77801
† Department of Electrical Engineering, Princeton University, Princeton, NJ 08544

***Abstract*—Graphics Processing Units (GPUs) with a few hundred extremely simple processors represent a paradigm shift for highly parallel computations. We use this emergent GPU architecture to provide a first demonstration of the feasibility of real time ML decoding (in software) of a high rate space-time block code that is representative of codes incorporated in 4th generation wireless standards such as WiMAX and LTE. The decoding algorithm is conditional optimization which reduces to a parallel calculation that is a natural fit to the architecture of low cost GPUs.Experimental results demonstrate that asymptotically the GPU implementation is more than 700 times faster than a standard serial implementation. These results suggest that GPU architectures have the potential to improve the cost / performance tradeoff of 4th generation wireless base stations. Additional benefits might include reducing the time required for system development and the time required for configuration and testing of wireless base stations.**

I. Introduction

In the space of ten years, space-time codes have progressed from invention to adoption in major wireless standards. For example, the Alamouti code [7] and the Golden code [8] are both incorporated in the WiMAX standard. Correlation of signals across multiple transmit antennas improves the reliability of communication over fading channels and increases downlink efficiency by reducing the variation of signal strength at the mobile terminal.

The reason to introduce structure at the transmitter is to simplify receiver signal processing. The Alamouti space-time code is an example of a complex orthogonal design (see [3]), and for such codes it it possible to implement Maximum Likelihood (ML) decoding with only linear processing at the receiver. However the Alamouti code is the only full rate complex orthogonal design, and given the importance of high data rates, it is natural to relax the requirement that linear processing at the receiver be able to separate all transmitted symbols. One path to higher data rates is multiplexing of codes, and this is how the Golden and Silver codes are constructed. We follow Sirianunpiboon et al [1] by employing conditional optimization as a decoding primitive for such codes. We note that sphere decoding [22] is an alternative approach. A particular advantage of conditional optimization is that it reduces to parallel calculation that is a natural fit to the GPU architecture. The focus of this paper is decoding of a particular code (the Silver code) by conditional optimization, but the same method applies much more generally. In fact it is possible to imagine a single decoding architecture that applies to all codes incorporated in a given standard and where the decoding algorithm does not change as the propagation environment varies back and forth from rich scattering to Line of Sight (LOS). Recently, decoding by conditional optimization [1] has been explored, which achieves exactly or essentially ML decoding performance with fixed lower complexity. For example, the Golden code and the Silver code, can also be decoded optimally or nearly optimally with quadratic complexity [1], [2]. Fast decodability makes these space-time codes very attractive in practice, especially in highly parallel systems such as a GPU platform.

The work of R. Calderbank and Y. Wu is supported in part by NSF under grant 0701226, by ONR under grant N00173-06-1-G006, and by AFOSR under grant FA9550-05-1-0443.

Graphics Processing Units (GPUs) with a few hundred extremely simple processors represent a paradigm shift for highly parallel computations. Current GPUs are designed to operate in a Single Instruction Multiple Data (SIMD) fashion, which allows the simple processors to share control logic. In recent times, the application of GPUs for general purpose computations has been actively explored [23], [24], [25], [26], [30], [31]. The rapid increase in the number and diversity of scientific communities exploring the computational power of GPUs for data intensive applications has motivated GPU manufacturers to more powerful, easily programmable and flexible GPUs. Figure 1 presents a comparison between the rate at which compute power has been scaling in successive generations of CPUs and GPUs. [32]. The development of open-source programming tools

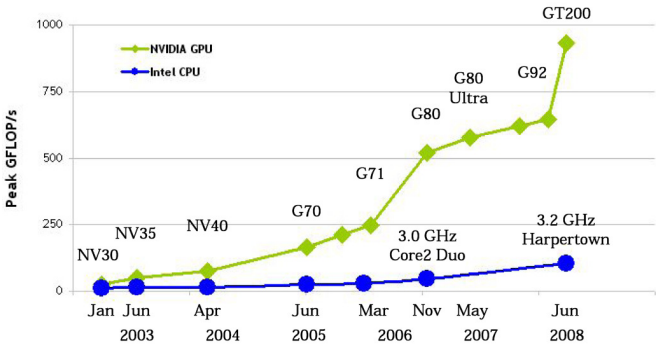

Fig. 1. Comparison of performance scaling in GFLOP/s

and languages for interfacing with GPU platforms has accelerated growth by making it easier to develop general purpose GPU (GPGPU) applications. GPU architectures have been rapidly evolving towards higher performance, larger memory size and larger memory bandwidths at relatively lower costs. The theoretical performance of the GTX 280 GPU (used in the GeForce GTX 280 graphic card) is more than 930 Gflops [27]. This high computing power largely results from a fully pipelined and highly parallel architecture, with extremely high memory bandwidths. The memory bandwidth of the NVIDIA GeForce GTX 280 GPU is 141.7 GB/s. In contrast the theoretical performance of a 3 GHz Pentium 4 CPU is 12 Gflops, with a memory bandwidth of 6 GB/s to main memory.

Massively parallel computation increases programming complexity; code execution patterns and memory access patterns of the GPU threads need to be carefully crafted in order to benefit from the parallelism of the GPU platform. Failure to do this will degrade

performance and a key challenge faced by GPU programmers is to translate computational challenges into the hardware and software constraints presented by the GPU platform.

Here the application focus is ML or essentially ML decoding of high rate space-time block codes via conditional optimization. This is a technique used widely in statistical estimation and signal processing to reduce complexity by optimizing over some subset of the parameters conditioned on the remaining parameters. It is particularly well matched to the GPU architecture, and when we compare with standard computer architecture we are able to reduce the time required for WiMAX decoding by factor of 700. We envision three scenarios in WiMAX where our GPU implementation might find application.

- *Base Station*: Base stations receive uplink data from mobile stations. When large data transfer rates are chosen, the decoding task is very computationally intensive. The situation is aggravated with increasing bandwidths (since the number of data tones is proportional to spectral bandwidth for WiMAX). We demonstrate the practicability of offloading decoding to our GPU decoder by showing that data received in one WiMAX frame can be decoded before the next frame arrives.
- *Configuration and Test*: We show that it is feasible to incorporate our GPU implementation in reference test equipment that is required for base station configuration.
- *System Development*: High rate space-time codes are incorporated as options in the WiMAX standard. Implementation by manufacturers requires extensive simulation to accurately measure Bit Error Rates (BER) and error floors. Increasing the speed of software decoding by more than two orders of magnitude has the potential to reduce design cycle times and might lead to wider adoption of more sophisticated codes.

We expect that the gains reported in this paper for the WiMAX standard will extend to the LTE family of standards.

Section II reviews decoding by conditional optimization and Section III describes the application of this method to the Silver code. Section IV provides a high level introduction to the GPU architecture and associated software programming model. The WiMAX frame structure and the implementation of ML decoding on the GPU architecture are described in Section V. Experimental results for two different platforms are presented in Section VI and Section VII provides conclusions.

II. Decoding by Conditional Optimization

Conditional optimization [1] is a decoding primitive that is particularly well matched to high rate space-time codes that are obtained by multiplexing simple blocks. This family of codes includes the Golden code [8], [9], [10] the Rabiei–Al-Dhahir code [18]. Conditional optimization can serve as an ML or essentially ML decoder for all codes incorporated in the WiMAX standard. Multiplexing of simple blocks at the transmitter leads to a received signal of the form in Equation 1. Throughout this paper boldface upper-case letters denote matrices, non-boldface upper-case letters denote scalars, boldface lower-case letters denote vectors and non-boldface lower-case letters denote scalar variables. The notation $(\cdot)^*$, $(\cdot)^\dagger$ and $\|\cdot\|_{\mathrm{F}}$ denote complex conjugation, matrix adjoint, and Frobenius norm respectively.

$$\mathbf{r} = \mathbf{P}\mathbf{c} + \mathbf{Q}\mathbf{s} + \mathbf{n} \tag{1}$$

The signal $\mathbf{r}$ is 2-dimensional since we assume 2 antennas at the receiver, the transmitted signals $\mathbf{c} = (x_1, x_2)^T$ and $\mathbf{s} = (x_3, x_4)^T$ are drawn from a QAM constellation Λ, and the matrices $\mathbf{P}$ and $\mathbf{Q}$ represent the channels induced at the receiver by the blocks that are multiplexed at the transmitter. The noise $\mathbf{n}$ is assumed to be complex Gaussian with zero mean and variance σ^2.

The likelihood function for equation (1) is

$$p(\mathbf{r}|\mathbf{c},\mathbf{s}) = \exp\left(-\frac{1}{2\sigma^2}\|\mathbf{r} - \mathbf{P}\mathbf{c} - \mathbf{Q}\mathbf{s}\|_{\mathrm{F}}^2\right) \tag{2}$$

and the ML solution is given by

$$(\hat{\mathbf{c}}, \hat{\mathbf{s}}) = \underset{\mathbf{c},\mathbf{s}\in\Lambda^2}{\operatorname{argmax}}\, p(\mathbf{r}|\mathbf{c},\mathbf{s}). \tag{3}$$

Exhaustive search over $\mathbf{c}$ and $\mathbf{s}$ provides ML decoding with complexity $O(|\Lambda|^4)$ and conditional optimization reduces the cost of ML decoding to $O(|\Lambda|^2)$. Absent this reduction, ML decoding in the context of the WiMAX standard is not feasible. Conditional optimization first decides the ML solution of $\mathbf{c}$ given $\mathbf{s}$ as

$$\hat{\mathbf{c}}(\mathbf{s}) = \lceil \tilde{\mathbf{c}}(\mathbf{s}) \rfloor_\Lambda \tag{4}$$

where $\tilde{\mathbf{c}}(\mathbf{s}) = (\mathbf{P}^\dagger\mathbf{P})^{-1}\mathbf{P}^\dagger(\mathbf{r} - \mathbf{Q}\mathbf{s})$ and $\lceil\cdot\rfloor_\Lambda$ is the rounding operation for each entry of finding its nearest constellation point on the constellation Λ, also known as slicing. For example, if symbols are taken from QPSK(4-QAM), then $\lceil(0.5 - 0.3i, -1.5 + 1.2i)\rfloor = (1 - i, -1 + i)$.

Conditional optimization then obtains the solution for $\mathbf{s}$ as

$$\hat{\mathbf{s}} = \underset{\mathbf{s}\in\Lambda^2}{\operatorname{argmax}}\, \|\mathbf{r} - \mathbf{P}\hat{\mathbf{c}}(\mathbf{s}) - \mathbf{Q}\mathbf{s}\|_{\mathrm{F}}^2 \tag{5}$$

and it is clear that decoding complexity is $O(|\Lambda|^2)$.

Remark: Sirianunpiboon et al. [1] showed that if $\mathbf{P}^\dagger\mathbf{P} = \mathbf{I}$ and $\mathbf{Q}^\dagger\mathbf{Q} = \mathbf{I}$, then ML decoding is achieved by conditional optimization. These conditions are not satisfied by the Golden code but decoding performance is still indistinguishable from that of ML decoding (see [2]).

III. Decoding of the Silver Code by Conditional Optimization

We demonstrate speedups that result from implementing conditional optimization on the GPU architecture by focusing on a single representative code.

A. The Silver Code

The Silver code is described by a 2×2 matrix where the columns represent different time slots, the rows represent different antennas, and the entries are the QAM symbols to be transmitted." The encoding rule is

$$\mathbf{S} = \begin{pmatrix} x_1 & -x_2^* \\ x_2 & x_1^* \end{pmatrix} + \mathbf{V}\begin{pmatrix} x_3 & -x_4^* \\ x_4 & x_3^* \end{pmatrix} \tag{6}$$

where

$$\mathbf{V} = \frac{1}{\sqrt{7}}\begin{pmatrix} 1-i & 1-2i \\ 1+2i & -1-i \end{pmatrix}$$

is a unitary matrix optimized for cubic shaping.

The Silver code is a full rate and full diversity code for 2×2 MIMO systems. It achieves a non-vanishing minimum determinant $1/7$ [17].

B. The Induced Channel at the Receiver

We consider a system with two transmit antennas and two receive antennas with (h_1, h_2) as the channel gains between the two transmit antennas and the first receive antenna and (g_1, g_2) as the channel gains between the two transmit antennas and the second receive antenna. Let (r_{11}, r_{12}) and (r_{21}, r_{22}) be the two received signal vectors with

the components as the signals received over two consecutive time slots. Then we have

$$(r_{11}, r_{12}) = (h_1, h_2)\begin{pmatrix} x_1 & -x_2^* \\ x_2 & x_1^* \end{pmatrix} + (\tilde{h}_1, \tilde{h}_2)\begin{pmatrix} x_3 & -x_4^* \\ x_4 & x_3^* \end{pmatrix} + (n_{11}, n_{12}) \tag{7}$$

and

$$(r_{21}, r_{22}) = (g_1, g_2)\begin{pmatrix} x_1 & -x_2^* \\ x_2 & x_1^* \end{pmatrix} + (\tilde{g}_1, \tilde{g}_2)\begin{pmatrix} x_3 & -x_4^* \\ x_4 & x_3^* \end{pmatrix} + (n_{21}, n_{22}) \tag{8}$$

where $(\tilde{h}_1, \tilde{h}_2) = (h_1, h_2)\mathbf{V}$, $(\tilde{g}_1, \tilde{g}_2) = (g_1, g_2)\mathbf{V}$ and $n_{11}, n_{12}, n_{21}, n_{22}$ represent additive complex Gaussian noise.

We can rewrite (7) and (8) as

$$\begin{pmatrix} \mathbf{r}_1 \\ \mathbf{r}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{H} \\ \mathbf{G} \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} \tilde{\mathbf{H}} \\ \tilde{\mathbf{G}} \end{pmatrix}\begin{pmatrix} x_3 \\ x_4 \end{pmatrix} + \begin{pmatrix} \mathbf{n}_1 \\ \mathbf{n}_2 \end{pmatrix} \tag{9}$$

which takes the form

$$\mathbf{r} = \mathbf{Pc} + \mathbf{Qs} + \mathbf{n} \tag{10}$$

where $\mathbf{r} = (\mathbf{r}_1, \mathbf{r}_2)^T$, $\mathbf{c} = (x_1, x_2)^T$, $\mathbf{s} = (x_3, x_4)^T$, $\mathbf{P} = (\mathbf{H}, \mathbf{G})^T$, $\mathbf{Q} = (\tilde{\mathbf{H}}, \tilde{\mathbf{G}})^T$ and $\mathbf{n} = (\mathbf{n}_1, \mathbf{n}_2)^T$.

We observe that $\mathbf{H}, \tilde{\mathbf{H}}, \mathbf{G}, \tilde{\mathbf{G}}$ are Alamouti blocks and

$$\mathbf{P}^\dagger\mathbf{P} = \frac{1}{2}\left(||\mathbf{H}||_F^2 + ||\mathbf{G}||_F^2\right)\mathbf{I}_2 \tag{11}$$

$$\mathbf{Q}^\dagger\mathbf{Q} = \frac{1}{2}\left(||\tilde{\mathbf{H}}||_F^2 + ||\tilde{\mathbf{G}}||_F^2\right)\mathbf{I}_2 \tag{12}$$

where $\mathbf{I}_2$ is the 2×2 identity matrix.

C. Quadratic Decoding by Conditional Optimization

The following steps are required to decode the Silver code by conditional optimization.

- Step 1: Choose $\bar{\mathbf{s}} = (\bar{x}_3, \bar{x}_4)^T$, where $\bar{x}_3$ and $\bar{x}_4$ from the constellation Λ.
- Step 2: Given $\bar{\mathbf{s}}$, calculate $\bar{\mathbf{c}}$ as

$$\bar{\mathbf{c}}(\bar{\mathbf{s}}) = \left\lceil \frac{2\mathbf{P}^\dagger}{||\mathbf{H}||_{\mathrm{F}}^2 + ||\mathbf{G}||_{\mathrm{F}}^2}\left(\mathbf{r} - \mathbf{Q}\bar{\mathbf{s}}\right) \right\rfloor_\Lambda \tag{13}$$

- Step 3: Calculate the square metric

$$M_{\bar{\mathbf{s}}} = ||\mathbf{r} - \mathbf{P}\bar{\mathbf{c}}(\bar{\mathbf{s}}) - \mathbf{Q}\bar{\mathbf{s}}||^2 .$$

- Step 4: Repeat Step 2 and Step 3 searching over all possible $\bar{\mathbf{s}}$ (combinations of x_3 and x_4) and decide on the one with the least square metric as

$$\begin{cases} \hat{\mathbf{s}} = \underset{\bar{x}_3, \bar{x}_4 \in \Lambda}{\operatorname{argmin}} M_{\bar{\mathbf{s}}}, \\ \hat{\mathbf{c}} = \bar{\mathbf{c}}(\hat{\mathbf{s}}). \end{cases} \tag{14}$$

IV. GPU Hardware and Software Model

We next discuss the architectural features of the NVIDIA GeForce GTX 280 GPU device, which is the GPU used in our implementation. This brief discussion is provided to enable a better understanding of our implementation of the Silver code decoding by conditional optimization on the GPU. Additional details of the GTX 280 can be found in [27], [28].

A. Hardware and Memory Model

1) Hardware Model: The GeForce GTX 280 architecture has a total of 240 processor cores which are distributed such that there are 30 multiprocessors per chip and 8 processors per multiprocessor. Since the GPU operates in a SIMD fashion, all the software threads running on the 8 processors in a multiprocessor execute the same instruction, but may operate on different data. We next describe the memory organization of the GTX 280 device.

2) Memory Model: There are four types of on-chip memories on each multiprocessor [27], [28].

- Local 32-bit registers. The total number of registers per multiprocessor is 16384 (64 KB). All registers are accessible to all processors in the multiprocessor. Registers are allocated to threads and cannot be shared across threads.
- Shared memory that is shared by all the processors of a multiprocessor. The size of this shared memory per multiprocessor is 16 KB, and it is organized into 16 banks. Contiguous words are located in successive banks, wrapping around after the 16th word. This helps achieve a high memory bandwidth when contiguous words are accessed. Shared memory has low access latency when contiguous words are accessed.
- A read-only constant memory that is shared by all the processors. Constant memory is cached separately by each multiprocessor. The amount of constant cache is 8 KB per multiprocessor. The total amount of constant memory is 64 KB.
- A read-only texture memory. This memory is cached separately in each multiprocessor. The size of texture cache is 8 KB per multiprocessor.

Global memory is read/write and is not cached. The global memory access latency for reading/writing a single floating point value can be 400 to 600 clock cycles. A considerable amount of this global memory access latency can be hidden if there are sufficient arithmetic instructions that can be issued while waiting for a global memory access to complete. Further, coalesced accesses (i.e. accesses which are aligned) of 32-bit, 64-bit, or 128-bit quantities should be performed, in order to increase the throughput and to maximize the bus bandwidth utilization.

B. Software Model

CUDA (Compute Unified Device Architecture) is a new hardware and software architecture which is used for interfacing with the GPU device. It allows the user to issue and manage computations on the GPU without the need to map them to traditional graphics APIs [27], [28]. In CUDA, the GPU is viewed as a compute device capable of executing a large number of threads in parallel. Threads are the atomic units of parallel computation. The GPU device operates as a co-processor to the main general purpose processor, or host. Data-parallel, compute intensive portions of an application running on the host can be offloaded onto the GPU device. Such a portion of code is compiled into the instruction set of the GPU device and the resulting program is called a *kernel*. The kernel is executed on the GPU device, and each instance of the kernel is called a *thread*. A thread block (also referred to as a block) is a batch of threads that can cooperate efficiently by sharing data through fast shared memory, and synchronizing their execution to coordinate memory accesses. Synchronization points can be specified in the kernel. In such a case, during execution threads in a block are suspended until

all threads reach the same synchronization point. Threads are grouped into warps, which are further grouped in blocks. Threads have one, two or three dimensional identity numbers or *threadIDs*. This helps in organizing the competition for problems which have an underlying one, two or three dimensional geometry. The GeForce GTX 280's synchronization paradigm is efficient, but it is local to a thread block. Threads belonging to different thread blocks cannot synchronize.

V. WiMAX Decoding on the GPU

A. WiMAX Specifications

WiMAX is a wireless broadband solution that offers very high peak data rates, scalable bandwidth as well as data rate support. It utilizes adaptive modulation and coding, orthogonal frequency division multiple access (OFDMA) and dynamic per-user resource allocation. Data communication between the base station and a mobile station occurs via spatio-temporal blocks called *frames*. A frame is the information communicated between base and mobile stations over a predefined spectral bandwidth, for a predetermined duration. Table I lists the OFDM parameters for WiMAX, as approved by the WiMAX forum.

It can be observed from Table I that irrespective of the bandwidth, the OFDM symbol duration remains constant across all Mobile WiMAX profiles. The constant symbol duration is a result of fixed sub-carrier spacing in mobile WiMAX profiles. Note that the symbol duration is agnostic to the QAM constellation size used for modulation as well. This feature not only helps in linear scaling of data rate with bandwidth but also allows hardware designers to use a standardized design for all the profiles. The symbol duration is always fixed at 102.9μs for all the Mobile WiMAX profiles.

The available bandwidth for data is divided between downlink (from base station to mobile stations) and uplink (from mobile stations to base station) data transfers. WiMAX specifies two ways of dividing bandwidth between uplink (UL) and downlink (DL). They are:

- *Frequency Division Duplexing* (**FDD**): Here the available bandwidth is divided between UL and DL by allocating separate tones for UL and DL data transfer.
- *Time Division Duplexing* (**TDD**): Here the available bandwidth is divided between UL and DL by allocating different time slots for UL and DL data transfers.

Although both FDD and TDD specifications are part of the WiMAX standard, TDD is favored by majority of implementations because of the following advantages:

- Flexibility in choosing UL to DL data rate ratios.
- The ability to exploit channel reciprocity.
- The ability to implement in non-paired spectrum.
- Less complex transceiver design.

This is the reason we restrict our attention to the TDD mode in WiMAX. Figure 2 illustrates a WiMAX frame in TDD mode with OFDMA. The OFDMA nature of Figure 2 is illustrated by different DL/UL bursts for different mobile stations.

A WiMAX frame length is determined by the base station and can vary from 2ms to 20ms. In our experiments we restrict ourselves to 4 representative frame lengths of WiMAX (4, 5, 10 and 20 ms). A frame is divided into two sub-frames, a DL sub-frame followed by an UL sub-frame after a small guard interval. The DL to UL sub-frame ratio may be varied to support different traffic profiles. As shown in Figure 2, the DL sub-frame begins with a preamble that is used for physical-layer procedures, such as time and frequency synchronization and initial channel estimation [29]. The preamble

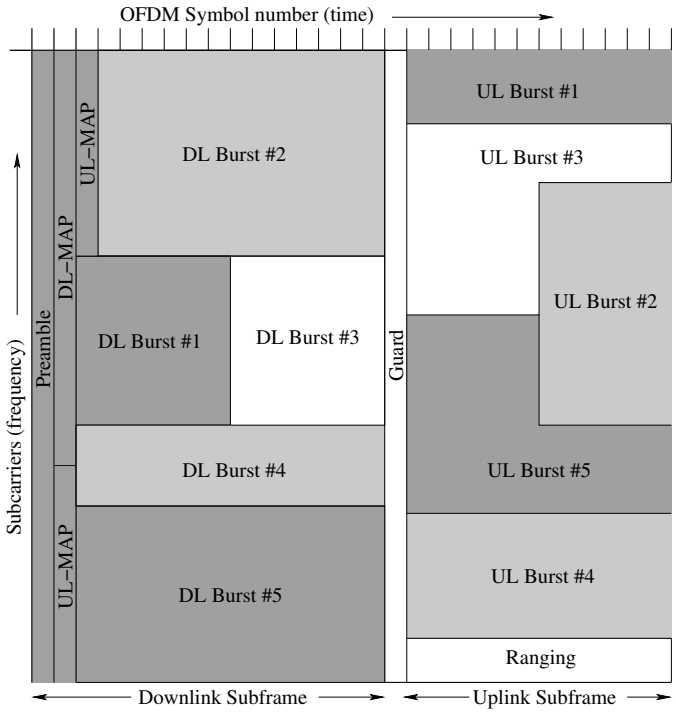

Fig. 2. WiMAX frame structure in TDD mode[29]

is followed by frame control header (FCH), which provides frame configuration information. FCH contains information like the MAP message length, the modulation and coding scheme, and the usable sub-carriers. Multiple users are allocated data regions (bursts) within the frame, and these allocations are specified in the MAP messages (DL-MAP and UL-MAP). MAP messages are broadcast following the FCH. MAP messages include the burst profile for each user, which defines the modulation and coding scheme used in that link. Since MAP messages contain critical information that needs to reach all users, it is often sent over a very reliable link, such as BPSK modulation with a block length 2 repetition code. The MAP messages are an effective way for the base station to inform the various users of its allocations and burst profiles on a per-frame basis. We next discuss our implementation of decoding by conditional optimization on the GTX 280 GPU.

B. GPU Implementation

The algorithm is split into four sections as shown in Figure 3. A single horizontal line in any section indicates a single-threaded computation, while multiple horizontal lines indicate a multi-threaded task.

The individual parts are described below:

- *Data Transfer to Device*: This phase of the algorithm is single threaded and executes on the CPU. All the data required for decoding is transfered to the global memory of the GPU. This is done through CUDA API calls. Through CUDA, a memory controller initiated data transfer is performed from the CPU's main memory to the GPU's global memory. The data transfered includes:
  - *Channel Matrix*: In the WiMAX standard the channel is estimated once per OFDM frame. The channel matrix ($\mathbf{H}$, $\mathbf{G}$ from Equation (9) and other accompanying matrices ($\tilde{\mathbf{H}}$ and $\tilde{\mathbf{G}}$ from Equation (9), $\mathbf{P}$, $\mathbf{Q}$, $\mathbf{P}^{\dagger}$ and $\mathbf{Q}^{\dagger}$ from Equations (11)

Parameter	Fixed WiMAX OFDM-PHY	Mobile WiMAX Scalable OFDMA-PHY			
FFT size	256	128	512	1,024	2,048
Number of used data sub-carriers	192	72	360	720	1,440
Number of pilot sub-carriers	8	12	60	120	240
Number of null/guardband sub-carriers	56	44	92	184	368
Channel bandwidth (MHz)	3.5	1.25	5	10	20
Sub-carrier frequency spacing (kHz)	15.625	10.94			
Useful symbol time (μs)	64	91.4			
Guard time assuming 12.5% (μs)	8	11.4			
OFDM symbol duration (μs)	72	102.9			
Number of OFDM symbols in 5ms frame	69	48			

TABLE I
OFDM PARAMETERS USED IN WIMAX[29]

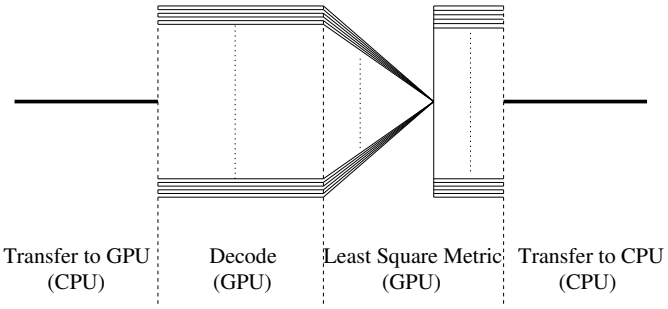

Fig. 3. Thread level description of the algorithm

and (12) are computed on the CPU and are transfered to the GPU. Since these matrices are shared across all the GPU based decoding threads, computing the channel matries on the CPU is more efficient.

  - *Received OFDM Symbols*: All the received symbols that are to be decoded together are transfered to the GPU as 2×2 matrices. These symbols are the output of the FFT unit.

- *Decode*: The decode phase of the algorithm is implemented with conditional optimization. This part of the algorithm is computationally intensive and is executed exclusively on the GPU. Multiple threads are launched on the GPU as illustrated in Figure 3. The decoding algorithm has linear time complexity in the size of the input (i.e. the number of input symbols). Each thread computes $\hat{\mathbf{c}}$ for a value of $\hat{\mathbf{s}}$ (Equation (14)). Based on the size of the QAM constellation, threads handling a single decoding are either grouped into 1 thread block (16-QAM), many thread blocks (64-QAM) or a fraction of a thread block (QPSK). Threads are identified based on their *threadID*. The *threadID* in a block is a three dimensional quantity (dimensions X_T, Y_T and Z_T) and the *blockID* is a two dimensional (dimensions X_B and Y_B) quantity. The X_T and Y_T dimensions of the *threadID* identify the x_3 and x_4 values from Equations (7) and (8) assigned to the thread. The Z_T dimension of *threadID* and the X_B and Y_B dimensions of the *blockID* together identify the symbol set (in global memory) assigned to the thread. Each thread in the decode kernel computes Steps 1, 2 and 3 of Section III-C. The square metric from Step 3 of Section III-C and the estimated solution are stored back into global memory. The assigned storage location in global memory is identified by the Z_T dimension of *threadID* along with X_B and Y_B dimensions of the *blockID*.
- *Least Square Metric*: This part of algorithm is also executed exclusively on the GPU. The Least Square Metric phase of the algorithm implements Step 4 of Section III-C. The values of square metrics computed in Step 3 of Section III-C are copied into shared memory. Subsequently with logarithmic time complexity, the value of the smallest square metric is identified (using the ideas in [33]). This phase of the algorithm is depicted by the converging lines in Figure 3. Once the least square metric value is identified the solution with the identified least metric value is looked up in parallel in global memory. This is depicted in Figure 3 by the parallel lines shown to the right of the converging lines (i.e. the least square metric computation). The solution is written back to global memory at a designated location. The storage location in global memory is identified by the Z_T dimension of *threadID* along with the X_B and Y_B dimensions of the *blockID*.
- *Data Transfer to Host*: This phase of the algorithm is single threaded and is executed on the CPU. Decoded symbols that were copied to global memory in previous step (Step 4) are transfered to the main memory of the CPU. This is carried out using CUDA API calls.

1) Overlapped Compute and Transfer: The *Transfer to Device* and the *Transfer to Host* parts of the algorithm (Figure 3) run on the general purpose processor, while the *Decode* and the *Least Square Metric* of the algorithm run on the GPU. This provides the opportunity to overlap transfers with computation. This is illustrated in Figure 4.

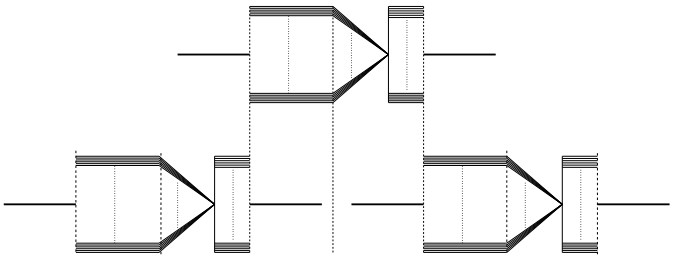

Fig. 4. Thread level description of overlapped transfer and decode

The total memory available on the GPU is divided into fragments. While data in one fragment is decoded, the data in another fragment

can be transfered to main memory and be filled with new symbols to be decoded. This improves the overall throughput. This technique increases throughput by a large amount when the transfer time of the algorithm is not an insignificant part of the total runtime. All results in Section VI are generated assuming overlapping transfer and computation.

VI. Experimental Results

We perform all our experiments on two separate platforms (PF1 and PF2) whose details are listed in Table II. We compare our GPU based implementation (running on both PF1 and PF2) with a single threaded CPU based implementation in terms of performance. In all our results we assume the worst case situation (i.e. all packets need to be decoded in one frame duration).

Parameter	PF1	PF2
Processor	Intel Pentium 4	Intel Core2 Quadro
Memory	2 GB	4 GB
GPU	GTX 280	Tesla C1060
Bus Interface	PCIe 1.0a	PCIe 2.0[1]
GPU Memory	1 GB	4 GB
GPU Memory Speed	1 GHz	800 MHz

TABLE II
Configuration of Test Platforms

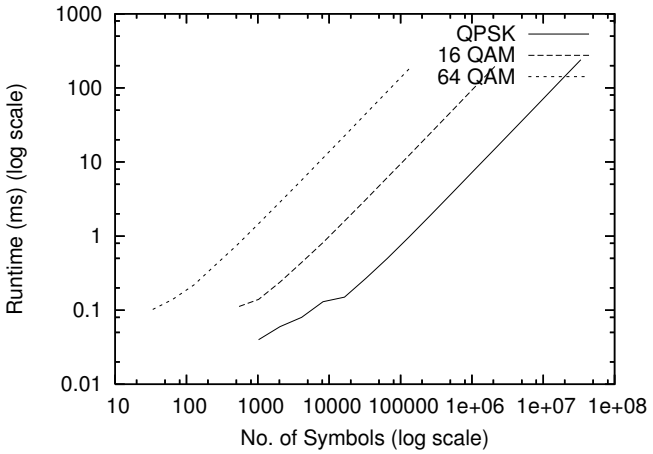

Fig. 5. Runtime vs. Number of QAM Symbols

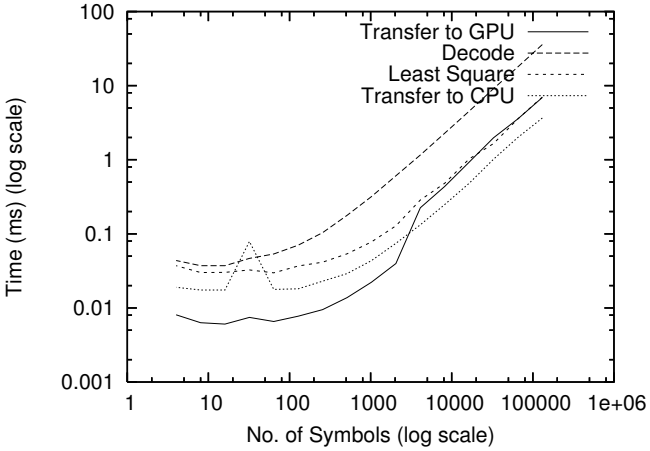

Fig. 6. Time vs. Number of QAM Symbols

Figure 5 plots the total runtime on PF2 to decode data when different modulation schemes are used. Both the runtime (y axis) and the number of QAM symbols to decode (also referred to as a *fragment* on x axis) are plotted in logarithmic scale. The runtime includes initialization time, decoding time, and all data transfer times. The initialization time includes time spent in various CUDA API calls and the resource allocation to initiate communication with the GPU. It can be observed that the algorithm scales linearly with the size of the input. For small fragment sizes the initialization time dominates the total runtime. It can be observed in Figure 5 that for small sizes of fragments the runtime on the GPU does not scale linearly in the size of the input.

Figure 6 splits up the runtime on PF1 for decoding 16-QAM into 4 parts: a) data transfer to GPU, b) decoding , c) least square metric calculation and d) solution transfer to CPU. Note that for small fragment sizes individual components of the runtime do not scale linearly. Both the axes are in logarithmic scale.

Table III lists the time taken to decode different number of symbols on a CPU (using the CPU of PF1), and the corresponding speedup observed on the two GPU platforms. The information for PF2 in Table III is derived from Figure 5. As the fragment size increases, the total runtime becomes linear in the size of the input. PF1 is slower than PF2 in decoding QPSK symbols but is faster than PF2 in decoding 16-QAM and 64-QAM symbols. This is due to the fact that PF1 has a slower PCI Express bus than PF2. Hence the data transfer time in case of QPSK modulation dominates the decode time on the GPU, on PF1. For the 16-QAM and 64-QAM modulation schemes, the runtime is dominated by the decode time. Hence PF1 performs better due to faster access to global memory in PF1 compared to PF2. For a input data sizes larger than those mentioned in Table III, the input can be split into fragment sizes that fall in the linear region. Processing these fragments sequentially ensures that the total runtime is still linear in the size of input. For example, the task of decoding 16777216 symbols could be split into 2 tasks of decoding 8388608 symbols each. If time T were taken to decode one fragment of size 8388608, decoding two fragments of size 8388608 would result in a runtime of $2T$. Thus, for large input data sizes speedup saturates at the asymptotic value. Our experiments show that the asymptotic speedup for QPSK on PF2 is above 510× and for 16-QAM and 64-QAM on PF1 is above 700×.

As reported in Table I not all tones in the bandwidth available are used for data. Among the tones available, a user is usually allocated a fraction of tones as described in the UL-MAP and DL-MAP. Additionally, each user could be directed to use different modulation schemes. In such a scenario, a single OFDM symbol (the signal received at the antennas from all the tones during a OFDM symbol duration) could contain data encoded in different modulation schemes. In Table IV we list the number of tones that can be decoded on a GPU (using PF2) at the rate at which symbols are received. We report this for various modulation schemes. For modulation schemes where *all* the data tones can be decoded the

[1]PCI Express 2.0 delivers 4× more memory bandwidth to main memory as compared to PCI Express 1.0a

	QPSK			16-QAM			64-QAM		
No. of QAM Symbols	Runtime CPU (s)	Speedup GTX 280 (PF1)	Speedup Tesla C1060 (PF2)	Runtime CPU (s)	Speedup GTX 280 (PF1)	Speedup Tesla C1060 (PF2)	Runtime CPU (s)	Speedup GTX 280 (PF1)	Speedup Tesla C1060 (PF2)
8388608	30.86	190.18×	514.92×	462.56	721.83×	591.76×	7214.08	709.22×	626.75×
4194304	15.86	196.6×	529.09×	231.28	721.83×	591.76×	3607.04	709.22×	626.75×
2097152	7.78	191.3×	518.32×	115.64	721.83×	591.76×	1803.52	709.22×	626.75×
1048576	3.9	186.56×	517.9×	57.82	721.83×	591.76×	901.76	709.22×	626.75×
524288	1.97	180.61×	522.55×	28.91	721.83×	591.76×	450.88	709.22×	626.75×
262144	1	173.95×	528.66×	14.46	716.67×	591.02×	225.44	709.22×	626.75×
131072	0.52	166.34×	540.75×	7.22	720.27×	587.99×	112.72	709.22×	626.75×
65536	0.33	232.61×	675.44×	3.61	720.68×	584.94×	56.36	709.22×	626.75×
32768	0.18	238.8×	698.46×	1.8	691.51×	579.36×	28.18	709.22×	626.75×
16384	0.09	256.8×	626.7×	0.9	668.84×	566.57×	14.12	708.6×	627.06×
8192	0.04	363.41×	374.39×	0.45	603.37×	548.22×	7.06	707.49×	624.54×
4096	0.02	285.46×	316.65×	0.22	606.27×	511.64×	3.56	695.52×	625.64×
2048	0.01	154.62×	197.27×	0.11	515.31×	477.24×	1.81	706.06×	630.62×
1024	-	-	-	0.05	406.35×	400.62×	0.92	720.29×	632.5×
512	-	-	-	0.02	265.52×	246.38×	0.49	656.64×	642.51×
256	-	-	-	0.01	162.4×	-	0.34	892.5×	836.47×
128	-	-	-	-	-	-	0.17	800.09×	811.59×
64	-	-	-	-	-	-	0.08	587.67×	652.67×
32	-	-	-	-	-	-	0.04	385.88×	461.37×
16	-	-	-	-	-	-	0.02	226.14×	-
8	-	-	-	-	-	-	0.01	145.76×	-

TABLE III
DECODING TIME FOR VARIOUS MODULATION SCHEMES AND THEIR SPEEDUP ON THE GPU PLATFORMS

decode runtime is reported. From Table IV, we can infer the largest ratio of the DL to UL duration such that decoding can be done in one frame duration. For modulation schemes where a fraction of the data tones can be decoded, we list the number of tones that can be decoded (in parentheses). For example, in the case of a 4ms frame (38 OFDM symbols) and bandwidth of 1.25MHz (72 data tones, from Table I), the total number of QAM symbols in a frame are $2\times72\times38$ (number of antennas $\times$ data tones $\times$ OFDM symbols). Our decode time for such a frame using 16-QAM modulation scheme is 0.51 ms, and therefore we can sustain any UL:DL ratio. A second example, in case of 64-QAM with a 20ms frame and bandwidth of 1.25MHz, our decoder can decode 38 tones. Therefore, we can sustain a UL:DL ratio of $(38) : (72 - 38) \approx 1 : 1$.

VII. CONCLUSION

We have provided a first demonstration of the feasibility of real time ML decoding (in software) of a high rate space-time code that is representative of codes incorporated in 4th generation wireless standards such as WiMAX and LTE. The decoding algorithm is conditional optimization and we have taken advantage of the fact that it reduces to a parallel calculation that is a natural fit to the architecture of low cost Graphics Processing Units. Experimental results demonstrate that asymptotically the GPU implementation is more than 700 times faster than a standard serial implementation on a CPU. These results suggest that GPU architectures have the potential to improve the cost / performance tradeoff of wireless base stations. Additional benefits might include reducing the time required for system development and the time required for configuration and testing of wireless base stations.

REFERENCES

[1] S. Sirianunpiboon, Y. Wu, A. R. Calderbank and S. D. Howard, "Fast optimal decoding of multiplexed orthogonal designs by Conditional Optimization," *submitted to IEEE Transactions on Information Theory*, May 2008.

[2] S. Sirianunpiboon, A. R. Calderbank and S. D. Howard, "Fast essentially maximum likelihood decoding of the Golden code," *submitted to IEEE Transactions on Information Theory*, June 2008.

[3] V. Tarokh, H. Jafarkhani, and A. R. Calderbank, "Space-Time Block Coding from Orthogonal Designs," *IEEE Transactions on Information Theory*, July 1999.

[4] Y. Wu and L. Davis, "Fixed-point fast decoding of the Silver code," in Proc. *Australian Communications Theory Workshop*, February 2009.

[5] Y. Wu and A. R. Calderbank, "Construction of high rate super-orthogonal space-time block codes," *International Conference on Communications*, June 2009.

[6] E. Biglieri, Y. Hong and E. Viterbo, "Silver space-time trellis-coded modulation," in Proc. *European Signal Processing Conference*, Lausanne, August 2008.

[7] S. Alamouti, V. Tarokh and P. Poon, "Trellis-coded modulation and transmit diversity: Design criteria and performance evaluation," in Proc. *IEEE International Conference on Universal Personal Communications*, vol. 2, pp. 917-920, October 1998.

[8] J.-C. Belfiore, G. Regaya, and E. Viterbo, "The Golden code: A 2×2 full-rate space-time code with nonvanishing determinants,", *IEEE Transactions on Information Theory*, Vol. 51, pp. 1432-1436, April 2005.

[9] H. Yao and G. W. Wornell, "Achieving the full MIMO diversity multiplexing frontier with rotation-based space-time codes," in Proc. *Allerton Conference on Communication, control and Computing*, October 2003.

[10] P. Dayal and M. K. Varanasi, "Optimal two transmit antenna space-time code and its stacked extensions," *IEEE Transactions on Information Theory*, Vol. 49, pp. 1073-1096, May 2003.

No. of Data subcarriers	Modulation Scheme	38 OFDM Symbols (4 ms)	48 OFDM Symbols (5 ms)	97 OFDM Symbols (10 ms)	194 OFDM Symbols (20 ms)
72	QPSK	0.04 ms	0.05 ms	0.1 ms	0.2 ms
in	16-QAM	0.51 ms	0.64 ms	1.3 ms	2.6 ms
128	64-QAM	(38)	(38)	(38)	(38)
192	QPSK	0.1 ms	0.13 ms	0.27 ms	0.53 ms
in	16-QAM	1.36 ms	1.72 ms	3.47 ms	6.93 ms
256	64-QAM	(38)	(38)	(38)	(38)
360	QPSK	0.2 ms	0.25 ms	0.5 ms	1 ms
in	16-QAM	2.55 ms	3.22 ms	6.5 ms	13 ms
512	64-QAM	(38)	(38)	(38)	(38)
720	QPSK	0.39 ms	0.49 ms	1 ms	1.99 ms
in	16-QAM	(565)	(565)	(565)	(565)
1024	64-QAM	(38)	(38)	(38)	(38)
1440	QPSK	0.78 ms	0.99 ms	1.99 ms	3.99 ms
in	16-QAM	(565)	(565)	(565)	(565)
2048	64-QAM	(38)	(38)	(38)	(38)

TABLE IV
DECODING TIME ON GPU FOR VARIOUS MODULATION SCHEMES (USING PF2)

[11] A. Hottinen and O. Tirkkonen, "Precoder designs for high rate space-time block codes," in Proc. *Conference on Information Sciences and Systems*, Princeton, March 2004.

[12] A. Hottinen, K. Kuchi and O. Tirkkonen, "A space-time coding concept for a multi-antenna transmitter," in Proc. *Canadian Workshop on Information Theory*, Vancouver, June 2001.

[13] O. Tirkkonen and A. Hottinen, "Square-matrix embeddable space-time block codes for complex signal constellations," *IEEE Transactions on Information Theory*, vol. 48, no. 2, pp. 384-395, February 2002.

[14] O. Tirkkonen and R. Kashaev, "Combined information and performance optimization of linear MIMO modulations," in Proc. *IEEE International Symposium on Information Theory*, Lausanne, Switzerland, June 2002.

[15] J. Paredes, A. B. Gershman, and M. G. Alkhanari, "A 2×2 space-time code with non-vanishing determinants and fast maximum likelihood decoding," in Proc. *IEEE International Conference on Acoustics, Speech, and Signal Processing*, Honolulu, Hawaii, USA, pp. 877-880, April 2007.

[16] A. Hottinen, O. Tirkkonen and R. Wichman, *Multi-antenna transceiver techniques for 3G and beyond,* WILEY publisher, UK, March 2003.

[17] C. Hollanti, J. Lahtonen, K. Ranto, R. Vehkalahti and E. Viterbo, "On the algebraic structure of the Silver code: a 2×2 perfect space-time block code," in Proc. *IEEE Information Theory Workshop*, Porto, May 2008.

[18] P. Rabiei and N. Al-Dhahir, "A new information lossless STBC for 2 transmit antennas with reduced-complexity ML decoding," in Proc. *IEEE Conference on Vehicular Technology*, September 2007.

[19] J. M. Paredes, A. B. Gershman and M. Gharavi-Alkhansari, "A new full-rate full-diversity space-time block code with nonvanishing determinants and simplified maximum-likelihood decoding," *IEEE Transactions on Signal Processing*, vol 56, No. 6, pp. 2461-2469 June 2008.

[20] F. Oggier, G. Rekaya, J.-C. Belfiore and E. Viterbo, "Perfect space-time block codes," *IEEE Transactions on Information Theory*, vol. 52, pp. 3885-3902, September 2006.

[21] E. Biglieri, Y. Hong, and E. Viterbo, "On fast-decodable space-time block codes," *submitted to IEEE Transactions on Information Theory*, arXiv:07082804, August 2007.

[22] M. Pohst, "On the computation of lattice vectors of minimal length, successive minima and reduced basis with applications," *ACM SIGSAM*, vol. 15, pp. 37-44, 1981.

[23] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "GPU cluster for high performance computing," in SC 2004: *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, (Washington, DC, USA), p. 47, IEEE Computer Society, 2004.

[24] J. Owens, "GPU architecture overview," in *SIGGRAPH 2007: ACM SIGGRAPH 2007 courses*, (New York, NY, USA), p. 2, ACM, 2007.

[25] D. Luebke, M. Harris, N. Govindaraju, A. Lefohn, M. Houston, J. Owens, M. Segal, M. Papakipos, and I. Buck, "GPGPU: general-purpose computation on graphics hardware," in SC 2006: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, (New York, NY, USA), p. 208, ACM, 2006.

[26] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Philips, "GPU Computing," in *Proceedings of the IEEE*, vol. 96(5), May 2008.

[27] "NVIDIA CUDA Homepage". http://developer.nvidia.com/object/cuda.html

[28] "NVIDIA CUDA Introduction". http://www.beyond3d.com/content/articles/12/1

[29] "WiMAX Fundamentals". http://www.wimax.com/commentary/wimax_weekly

[30] Kanupriya Gulati and Sunil P. Khatri, "Accelerating Statistical Static Timing Analysis Using Graphics Processing Units," *IEEE/ACM Asia and South Pacific Design Automation Conference*, pp. 260-265, Jan 19-22 2009.

[31] Kanupriya Gulati and Sunil P. Khatri, "Towards Acceleration of Fault Simulation using Graphics Processing Units," *IEEE Design Automation Conference*, pp. 822-827, June 8-13 2008, Anaheim, CA.

[32] "CUDA Pregramming Guide". http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf

[33] Mark Harris "Optimizing Parallel Reduction in CUDA". http://developer.download.nvidia.com