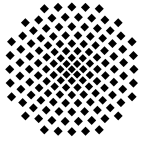

Universität Stuttgart

INSTITUT FÜR KOMMUNIKATIONSNETZE UND RECHNERSYSTEME
Prof. Dr.-Ing. Andreas Kirstädter

Copyright Notice

© 2009 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.

Institute of Communication Networks and Computer Engineering
University of Stuttgart
Pfaffenwaldring 47, D-70569 Stuttgart, Germany
Phone: ++49-711-685-68026, Fax: ++49-711-685-67983
Email: mail@ikr.uni-stuttgart.de, http://www.ikr.uni-stuttgart.de

A Novel Architecture using NVIDIA CUDA to speed up Simulation of Multi-Path Fast Fading Channels

Ahmed Fathy Abdelrazek, Matthias Kaschub, Christian Blankenhorn, Marc C. Necker
Institute of Communication Networks and Computer Engineering
Universität Stuttgart, Germany
eng.afathy@gmail.com, {kaschub,blankenhorn,necker}@ikr.uni-stuttgart.de

***Abstract*—The complexity of mobile communication systems is permanently increasing. This is due to the introduction and refinement of sophisticated communication mechanisms such as Multiple Input Multiple Output (MIMO), hybrid ARQ, channel-aware scheduling, or node cooperation. Many of these mechanisms require detailed multi-cell simulation models for their performance evaluation. This involves computationally extensive models of the wireless multi-path fading channel, which quickly becomes a bottleneck with respect to simulation time. It is therefore of great interest to reduce the amount of computation time spent in the channel calculation. In this paper, we present an attractive approach that offloads the channel computation to a massively parallel but inexpensive NVIDIA graphics card. We discuss the parallel architecture of the resulting simulation system, and study the involved synchronization and communication aspects. We show that the developed system achieves a speed-up factor of about 30 compared to an implementation on regular PC hardware.**

I. Introduction

In the past decades, the complexity of mobile communication systems has permanently increased. The demand for higher peak data rates and better spectral efficiencies has required the development of sophisticated communication techniques. Examples include Multiple Input Multiple Output [15] (MIMO), hybrid ARQ, channel-aware scheduling, or node cooperation mechanisms such as interference coordination [1]. The performance evaluation of these mechanisms requires detailed multi-cell simulation models that comprise many aspects of the physical layer. Especially the channel transfer functions are an important aspect. For OFDM based radio systems, this function has to be evaluated for all frequencies and requires therefore a huge amount of floating-point computation [11].

On conventional computer hardware, the computing time required for the simulation of these mechanisms is very high, leading to very long simulation times. The exploitation of parallelism is one possibility to reduce the simulation time. Dedicated hardware optimized for floating-point calculations may be used additionally. In particular, graphic cards provide an enormous amount of single-precision floating point computing power at low cost. However, this computing power used to be reserved for graphics acceleration. In [2], Adams et al. leveraged this power by formulating scientific code for solving Maxwell equations by means of OpenGL. This is a very cumbersome way of accessing the graphics card.

In 2007, NVIDIA released the CUDA architecture [3], which allows to access the graphics processors (GPU) on the graphics card by using a well-defined API. Memory access and programming is facilitated with standard C-language programming. CUDA provides an easy interface to a large floating point computing power, which opens the way to a large speed-up of the above discussed wireless network model components.

In this paper, we consider the implementation of multi-path channel models using the NVIDIA CUDA architecture. We discuss the parallel architecture of the resulting simulation system, and we study the involved synchronization and communication aspects. This architecture is compared to an implementation on a regular PC hardware, revealing a speed-up factor of about 30. The proposed system can therefore effectively increase the speed of wireless network simulations.

This paper is structured as follows. Section II introduces wireless channel models with a special focus on multi-path fading channels. Section III overviews the CUDA architecture and its capabilities. Subsequently, Section IV introduces the proposed wireless multi-path channel model implemented on the NVIDIA CUDA architecture. Finally, Section V evaluates its performance.

II. Wireless Channel Models

This section overviews wireless channel models in general and multi-path fading models in particular. Several basic effects have to be taken into account when modeling a wireless channel. First, *distance-dependent path loss* describes the reduction in power density as a function of propagation distance, frequency and other factors. The second effect is *multi-path propagation*, describing the effect that the signal at the receiver is composed of a large number of reflected radio waves. The different signal paths all experience different propagation delays. This causes the spread of the original signal in the time domain which is called delay spread. The multi-path effect can either increase or decrease the received signal strength, depending on whether the indvidual wavefronts interfere constructively or destructively. In particular, a *time varying channel* is obtained when there is a relative movement between transmitter and receiver. This motion produces Doppler shifts of incoming waves [4]. This shift is different for

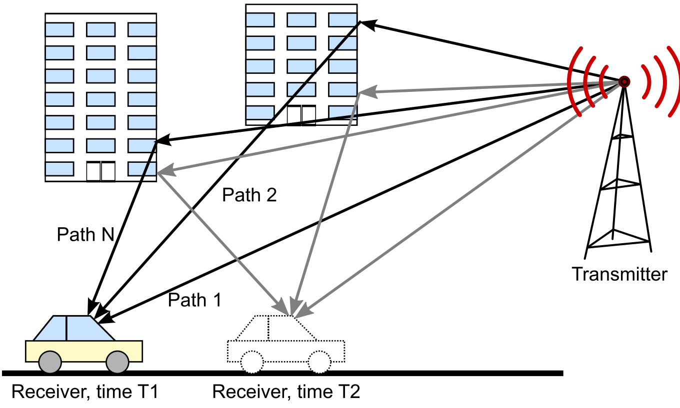

Fig. 1. Time varying multi-path propagation

every multi-path component depending on its angle of arrival, resulting in a *Doppler spectrum* and not only a single Doppler shift frequency for the received signal. All these effects are illustrated in Fig. 1.

A. Multi-Path Fading Models

Multi-Path fading in a wireless environment can be characterized by two main properties, namely the *frequency selectivity* and the *time variability*. The variation in time always occurs as soon as the terminal is moving and affects the amplitude and phase of the received signal. The variation depends on the speed of the mobile terminal and is also referred to as *fast fading*. If the delay spread of the channel is much smaller than the data symbol period, the channel is said to experience *flat fading*. In contrast, if *frequency-selective fading* is observed, different frequency components of the signal experience uncorrelated fading. While there exist several models for the wireless multi-path fading channel, we will study one particular model proposed by Höher in [9].

B. Höher Channel Model

Höher proposed a statistical discrete-time model for the WSSUS[1] multi-path fading channel in [9]. This is a widely used model and allows an easy implementation of the well know COST207 channel models, such as Typical Urban or Bad Urban. In his paper, Höher applied Forney's [5] discrete-time representation to Schulze's Monte Carlo-based channel model [8], [10]. He has shown that the instantaneous channel transfer function for the case of an optimum receiver filter on the basis of the Monte Carlo method is:

$$F_k(\omega) = lim_{N\to\infty} \frac{1}{\sqrt{N}} \sum_{n=1}^{N} e^{j(\theta_n + 2\pi f_{D_n} kT - \omega\tau_n)} \qquad (1)$$

where N is the number of propagation paths (taps) and θ_n, τ_n and f_{D_n} are random sequences each with a specific probability distribution. We can notice in this equation that, for every frequency ω and at every time (symbol) kT we need to loop over the N taps and do this complex computations which involve multiplication, addition, sine, co-sine, and finally square root calculation and division.

[1] wide-sense stationary with uncorrelated scattering

The Höher channel model has many advantages over other models, which make it well-suited for system-level simulation of radio systems. A relatively low number of N is sufficient to guarantee that the moments of the realization are close to the moments of the desired distribution, and single precision is enough for the model to give good results [6].

According to the Monte Carlo principle, it is convenient to establish a (portable) uniformly distributed noise generator with outputs $U_n \in [0, l[$, and to calculate ν_n by a functional transformation [14]:

$$\nu_n = g_\nu(u_n) = P_\nu^{-1}(u_n); \qquad 0 \le u_n < 1 \qquad (2)$$

Where ν_n, is a substitute for θ_n, f_{D_n}, and τ_n respectively, and the memoryless nonlinearity $g_\nu(u_n)$ is the inverse of the desired cumulative distribution function (cdf).

III. NVIDIA CUDA

GPUs are specifically designed for graphic applications and are very restrictive in terms of operation and programming. Because of their nature, GPUs are only effective in tackling problems, that can be solved by stream processing. Repeating the same instruction on different data sets achieves data parallelism, which is called Single-Instruction-Multiple-Data (SIMD). A SIMD architecture depends basically on the separation of control and data, such that a large number of simple math co-processors are under the control of a single master CPU. Thus, they can only process independent vertices and fragments. In this sense, GPUs are stream processors, that can operate in parallel by running a single kernel on many data records in a stream at once. A stream is simply a set of data records, e.g. vertices, that require similar computation.

A. General Purpose-GPU

GP-GPU is the idea of using a GPU in non-graphics applications, since not every application can be implemented as a graphics problem. Yet, only a special type of application that can benefit from GP-GPUs, those with floating-point intensive calculations that are repeated on a huge amount of data. Previous implementations on GP-GPUs suffer from problems such as [7]:

- **System bottlenecks**: Typical GPU programs rely heavily on the CPU to manage the complex parts of the control and data flow. The CPU-GPU communication is slower than the interaction among threads on the GPU. Therefore the latencies of these interaction limit the occupation of the GPU.
- **Instruction set**: GPUs offer huge peak performance gains. Yet, their architecture greatly constrains the programming of such devices. Therefore, representing the inherent complexity of the algorithms to be implemented is difficult and may lead to inefficient programs.
- **Programming model**: Previous GPUs could only be programmed through a graphics API, imposing a high learning curve to the novice. Further, the overhead for

non-graphics application is increased because of the inadequate API.

- **Memory access model**: The GPU DRAM can be read randomly. Yet, writing results back to memory is difficult, since GPUs usually present results on the computer screen.

B. NVIDIA CUDA Architecture

CUDA (Compute Unified Device Architecture) is a new hardware and software architecture for issuing and managing computations on the GPU. It allows for data-parallel computing without the need of mapping the problem to a graphics API. For a programmer CUDA greatly simplifies using the GPU as a co-processor for tasks different from graphics processing.

Memory hierarchy: A device is composed of a set of multi-processors, each is a SIMD unit with different on-chip memories. Compared to the device memory, these on-chip memories are usually very fast, but also very small. The different types of on-chip memory are optimized for different memory uses and differ in their hierarchy and visibility. *Registers* are available in each processor and are dedicated to threads, *shared memory* is shared by all the processors of a block, while the read-only *constant cache* and *texture cache* are shared by all the processors on the device. Threads can share data with each other through the on-chip shared memory, that has a very fast general read and write access. Applications can take advantage of shared memory by minimizing overfetch and round-trips to DRAM and therefore become less dependent on DRAM memory bandwidth. Shared memory is banked into 16 banks, that can be accessed simultaneously by 16 different threads in one cycle. To simplify programming, the device memory can also contain sections that are dedicated per thread (*local memory*). The different types of memory are summarized in table I.

MEMORY TYPE	ACCESS SCOPE	COMMENT
Register Memory	per thread	size decided at run time
Local Memory	per thread	size decided at run time
Shared Memory	per block	16K in 16 banks
Device Memory	global access	slow but huge
Constant Memory	global access	cached for fast access
Texture Memory	global access	cached for fast access

TABLE I
DIFFERENT MEMORY TYPES AND THEIR SCOPES

Programming paradigm: The CUDA software stack is composed of several layers - a hardware driver, an application programming interface (API) and its runtime, and two higher-level mathematical libraries. The hardware has been designed to support a lightweight driver and runtime layers, resulting in high performance.

CUDA provided a minimal set of extensions to the C-programming language to allow the programmer to target a part of the written code to be executed on the device. The C-language extension includes *function/variable type qualifiers* to specify whether a function/variable executes/resides on the host or on the device and *built-in variables* like blockId and threadId to identify threads and blocks. Runtime components are a set of APIs that give the programmer hands on device management, context management, memory management, and execution control.

C. CUDA Execution Model

To benefit from the processing power of the GPUs, a huge number of threads is needed to fill the pipeline and to employ all computing resources. GPU threads are different from the POSIX threads. All GPU threads share the same executable instructions (kernel), but each thread operates on different data sets. A kernel is executed by a group of threads called a *batch*. The batch of threads that executes a certain kernel is organized as a grid of *thread blocks*. A thread block is a group of threads that can cooperate by sharing data through the fast shared memory and synchronizing their execution. There is a limited number of threads that a block can contain. However, blocks executing the same kernel can be batched together into a grid, so that a larger number of threads can be launched in a single kernel invocation. This reduces the expense of thread cooperation, because threads in different blocks can not communicate with each other. Multiprocessor registers and shared memory are split among all threads within a grid. Therefore each multiprocessor can only process a limited number of blocks.

Every call specifies the execution configuration for the kernel, which defines the dimension of the grid and blocks, that will execute the kernel on the device. This model allows kernels to efficiently run without recompilation on various devices with different parallel capabilities: Depending on the parallel capabilities of a device, it may run all the blocks of a grid sequentially or in parallel or as a combination of both.

IV. COMPUTING MULTI-PATH FADING CHANNEL MODELS WITH CUDA

We show here, how the frequency selective multi-path fast fading algorithms can be mapped to the CUDA architecture. The main three challenges to the design for CUDA are: First, model independence of device capabilities and simulation environment. Second, the efficient use of device resources. And third, the real-time communication between the host and the device.

A. Simulation Scenario independant Computation

In Höher's equation Eq.1, the vectors for θ_n, τ_n, and F_{D_n} are of size N each, where N is the number of propagation paths. The value of N depends on the simulation scenario. This way, the amount of memory needed to store the random vectors depends on N. Therefore kernel execution configurations are different from one simulation experiment to another. To gain optimal performance the presented solution adapts itself according to the available resources of the device.

The execution of a kernel is designed to be handled with a fixed number of paths N_s, that is chosen depending on the available resources on the device. This way the vectors will

be with of fixed size N_s and the execution time of the kernel is deterministic. The processing of N paths is scheduled as a group of N_s paths, executed as a series of sequential kernel calls. The last kernel call in this series has extra work to do. It is responsible for the division by $\sqrt{N}$ and for copying the output data from the device to the host memory.

B. Efficient Usage of Resources

The on-chip shared memory is banked. This requires a special arrangement of data to achieve bank-conflict free execution. For every channel we need to store four vectors of size N_s each. Since there are 16 memory banks, we use 16 threads computing 16 different channels in parallel. So, vectors of every channel are not allocated as a contiguous block in the memory, but with 16 memory locations separating every two successive elements in a vector.

Accessing device memory degrades the performance. Following a certain memory access pattern can hide the memory access overhead. The access pattern proposed in [13] is found to be the most successful one. As a consequence of this special access pattern, the received results are not ordered and special data retrieval methods on the host are needed.

On one hand, CUDA requires a huge number of threads to fill the GPU's pipeline. On the other hand, the memory is shared among the threads, which decreases the available memory for the registers and the shared memory. By simple mathematics, the required amount of registers per thread is calculated to decide on the number of threads. With this number and the size of the shared memory we can determine the optimal value of N_s.

In addition to dividing the values of the N propagation paths into smaller kernels of N_s values to fit inside the shared memory, we have a thread that schedules one order received from the simulator into smaller orders that fit in the device memory. This happens when the memory requirements for the order will not fit into the device memory.

C. Synchronisation of simulation and channel computation

In an event-driven simulation the simulation time advances discontinuously and in non-equal steps. In contrast, the channel computation works on time spans of equal size, which require constant computation effort. Since the channel computation is independent of the simulation's progress, channels can be computed in advance. To limit the amount of memory needed to hold the channel data, the simulation thread provides the computation thread with the point in time, before which channel data can be deleted.

The simulation requires channel symbols at times before and after the simulation time. Ring buffers are used to store the channel data as shown in Fig. 2. The ring buffer is filled by the device and the simulation tool reads from it. The simulation tool advances the simulation time, and therefore the ring buffer rotates and some places becomes free, so the device can continue writing new values. Depending on the speed of the device and the simulation tool, blocking of simulation thread and device thread are controlled. To fully utilize CPU and GPU at the same time, it is necessary that the event-driven simulation and the channel computation take the same time. The ring buffer can average short variances of the time consumption, but in practice the slower component will determine the advance of simulation time.

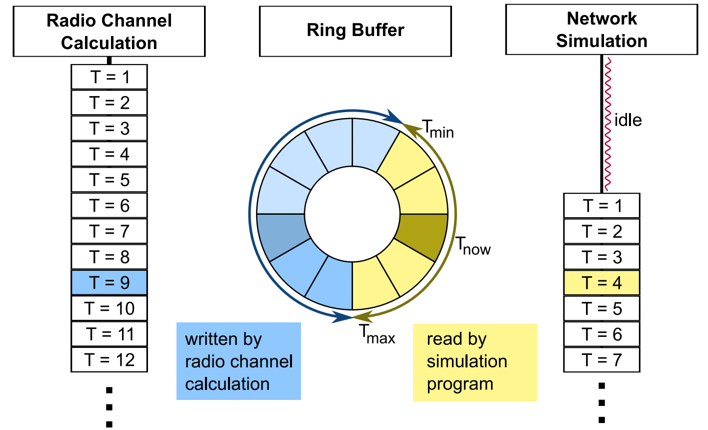

Fig. 2. System architectue for decoupling simulation and channel calculation

V. Performance evaluation and discussion

The theoretical execution time was calculated by analyzing the Höher equation (eq. 1) and calculating the number of clock cycles required for each part. The bandwidth of PCI-Express bus is a non-linear function of the data size being transferred. The bandwidth was calibrated at different operation points for accurate calculations of memory copying times. By using the values from [13] for the execution times of different instructions, we calculate the theoretical lower bound of the computation time.

The device we used in our evaluation was a NVIDIA GeForce 8800GTS with 500MHz core clock, 96 stream processors, and 320MB of memory. The device was connected via a 16x PCI-Express bus to an Intel 2.4GHz Core2Duo CPU with 2GB of RAM. Our application was built on top of an Ubuntu linux with NVIDIA CUDA driver 169.09, and CUDA SDK and CUDA toolkit releases 1.0. The channel computation is embedded into a simple event-driven network simulation using the IKR-Simlib [16].

For the results shown in Fig. 3 we used 3072 channels, 10 symbols, 1000 subcarriers, and a variable number of propagation paths (taps) N. The measured time corresponds to the real values obtained from the whole system and it is larger than the theoretical lower bound. This difference corresponds to the device memory accesses which are not accounted for in our estimation. Finally, we compare the execution time for the same model with the same parameters on an Intel Core2Duo processor.

As we can notice from Fig. 3, the difference between the theoretical lower bound and the measured values from the GPU is almost constant. At high values of N the memory overhead is negligible compared to the computation time. Thus, a speed-up factor of ≈ 30 is achieved. At low values

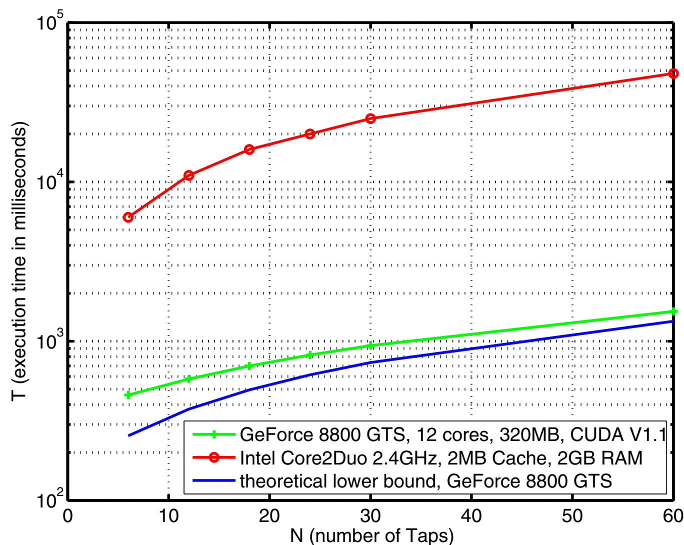

Fig. 3. Theoretical and real measured execution time on GPU

of N, the memory overhead is comparable to the instructions execution time. Therefore the speed-up factor is smaller.

Fig. 4 shows how the different model parameters affect the overall performance due to the scheduling decisions. For example, the scheduler divides the channels into groups. Every new group requires initialization time. In Fig. 4 the group size is 3072 channels. It is better to have a number of channels, that is a multiple of the group size.

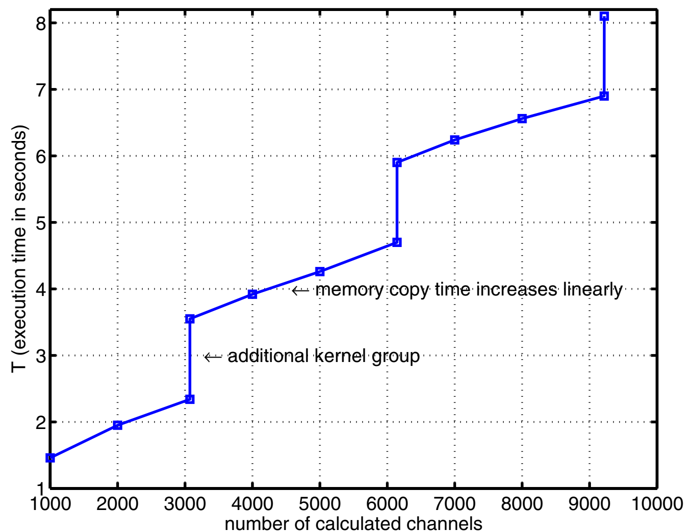

Fig. 4. Effect of extra channel group on the execution time

VI. Conclusion

In this paper, we propose to offload computing intensive wireless channel calculations in a wireless network simulator to a graphics card using NVIDIA CUDA. The proposed architecture efficiently utilizes the graphic processing resources by balancing the load on the GPUs and by efficiently using different memory types on the graphics card. The achieved performance of the channel computation is about 30 multiples faster compared to an Intel Core2Duo processor. The model efficiently performs the PC-CUDA communication while minimizing the device memory access overhead by following the access pattern explained in [13]. Moreover, the model features a memory bank conflict free execution. While our work is based on CUDA release 1.0, NVIDIA has lately announced the availability of the new releases 1.1, 2.0, and 2.1. The new releases support asynchronous memory operations, which can be used to further enhance the performance of the channel computation.

References

[1] Marc C. Necker, "Interference Coordination in Cellular OFDMA Networks", IEEE Network, vol. 22, iss. 6, Nov/Dec 2008.

[2] S. Adams, J. Payne, R. Boppana, "Finite Difference Time Domain (FDTD) Simulations Using Graphics Processors", in Proc. DoD High Performance Computing Modernization Program Users Group Conference, Pittsburgh, PA, June 2007.

[3] NVIDIA, CUDA, available at: http://www.nvidia.com/object/cuda_home.html.

[4] P. Dent, G. E. Bottomley, T. Croft. "Jakes fading model revisited." Electronics Letters, vol.29, no.13, pp.1162-1163, 24 June 1993.

[5] G.D. Forney, "Maximum-likelihood sequence estimation of digital sequences in the presence of intersymbol interference," IEEE Trans. Inform. Theory, vol.18, iss.3, pp. 363-378, May 1972.

[6] A. V. Oppenheim, R. W. Schafer. "Digital Signal Processing." Englewood Cliffs, NJ: Prentice-Hall, ch. 9, pp. 433-444, 1975.

[7] Ghuloum, Anwar. "The Problem(s) with GPGPU." Oct. 2007. http://blogs.intel.com/research/2007/10/the_problem_with_gpgpu.php

[8] H. Schulze, "Stochastic models and digital simulation of mobile channels" (in German), U.R.S.I./ITG Conf. in Kleinheubach 1988, Germany (FR), Proc. Kleinheubacher Berichte by the German PlT, Darmstadt, vol.32, pp.473-483, 1989.

[9] P. Höher, "A Statistical Discrete-Time Model for the WSSUS Multi-Path Channel", IEEE Transactions on Vehicular Technology. vol. 41, iss. 4, Nov. 1992.

[10] J. M. Hammersley and D. C. Handscomb, "Monte Carlo Methods". London: Methuen, 1964, reprinted 1975.

[11] K. Fazel and S. Kaiser, Multi-Carrier and Spread Spectrum Systems, John Wiley & Sons, ISBN 0-470-84899-5, 2003

[12] J.-P. M. G. Linnartz, "Wireless Communication Reference Website." Mar. 16, 2007. http://wireless.per.nl/reference/contents.htm

[13] NVIDIA, NVIDIA CUDA Compute Unified Device Architecture Programming Guide ver. 1.1, Nov. 29, 2007.

[14] R.F.W. Coates, G.J. Jannacec, and K.V. Lever, "Monte Carlo simulation and random number generation." IEEE J. Select. Areas Comm., vol. 6, iss. 1, pp. 58–66, Jan. 1988.

[15] H. Yang, "A Road to Future Broadband Wireless Access: MIMO-OFDM-Based Air Interface", IEEE Communications Magazine., pp. 53–60, Jan. 2005.

[16] IKR, IKR Simulation Library (SimLib), Institute of Communication Networks and Computer Engineering, Universität Stuttgart, available at: http://www.ikr.uni-stuttgart.de/IKRSimLib.