

GPU-accelerated Evaluation Platform for High Fidelity Network Modeling

Zhiguo Xu and Rajive Bagrodia
Mobile Systems Lab
Computer Science Department
University of California, Los Angeles
Email: {zhiguo, rajive}@cs.ucla.edu

Abstract

High-fidelity simulations of mixed wired and wireless network systems are dependent on detailed simulation models, especially in the lower layers of the network stack. However, detailed modeling can result in prohibitive computation cost. In recent years, commercial graphics cards (GPUs) have drawn attention from the general computing community due to the superior computation capability. In this paper, we present our experience with using commercial graphics cards to speed up execution of network simulation models. First, we propose a general simulation framework supporting GPU-accelerated simulation models. Software abstraction is designed to facilitate the use and development of GPU-based models. Second, we implement and evaluate two simulation models using GPUs. We observed that using the GPUs can yield significant performance improvements for large configurations of the model, as compared with pure CPU-based computations, with no degradation in the accuracy of the results. This benefit is particularly impressive for models that include significant data parallel computations. However, we also observed that the overhead introduced by GPUs make them less effective in improving execution time of other network models. This study suggests that besides parallel computing and grid computing, network simulations can also be scaled by reaping computation capability of GPUs and, potentially, other external computational hardware.

1 Introduction

Network simulators have been widely used for performance evaluation of protocols and applications due to its scalability, flexibility and repeatability. At the same time, fidelity remains a top concern among network researchers. For example, Takai et al. [15] show lack of detailed physical models and radio models can adversely affect accuracy in performance prediction of wireless network proto-

cols and applications. In recent years, there is clear consensus that detailed and accurate simulation models, especially for lower layers of the network protocol stack, are essential for high-fidelity performance evaluation. On the other hand, with the integration of more detailed models, the computation cost of simulation based performance evaluation may grow prohibitively high. In the past, researchers have extensively studied parallel computing and grid computing to speed up simulation-based evaluations. Recently we observe that hardware acceleration has also been used in various application areas, partially due to the fact that the computation capability of specialized hardware, e.g. graphics processing units (GPU), DSP platform etc., has been advancing rapidly. In this paper, we look at ways to reap the computational power of specialized hardware for acceleration of network simulation.

The graphics processing unit (GPU) is a dedicated graphics rendering device with substantial capability for processing graphics data sets. Consider two off-the-shelf graphics cards: GeForce 7900GTX processes up to 1.44 billion vertices per second and Radeon X1900 up to 1.3 billion vertices per second. Further, the advantage in computational power of GPUs against CPU has been widening, as projected in Figure 1. As of early 2006, nVidia GeForce 7800GTX had achieved 313 GFLOPS while high-end Pentium 4 can only deliver 25.6 GFLOPS. The GPU also has extremely fast access to texture memory, or its local on-board storage. The texture memory bandwidth in commercial cards is comparable to cache bandwidth in CPU. The nVidia GeForce 7900GTX supports memory bandwidth of 52.4GB/sec while L2 cache bandwidth of about 100GB/sec in Intel Pentium 4.

The superior processing capability and fast on-board memory access are attributed to the stream processing model with spatial parallelism [8]. The rendering process, or the graphics hardware pipeline [4], includes three stages: application, geometry processing and rasterization, as illustrated in Figure 2. For modern GPUs, the last two stages are programmable, or can be customized with user-defined

programs, known as vertex shader and fragment shader respectively. The vertex shader is executed on each point in the geometry processing stage while the fragment shader is used on each pixel in the rasterization stage. Substantial parallelism is built into modern GPUs. For example, in GeForce G70 series, there are 8 vertex processors and 24 fragments processors.

In recent years, GPUs have gained many followers for general-purpose computation, as surveyed by Owens et al in [13], because of better price to performance ratio compared with CPU, faster evolution pace and improved flexibility and programmability. The community (see <http://gpgpu.org>) has collectively accumulated valuable experiences for GPU-based general-purpose computing. Most general-purpose computing in GPU leverages the programmable fragment processing stage since fragment processors have much higher arithmetic rates than vertex processors. In the following, we summarize the steps involved in mapping computation from CPU to GPUs, particularly, in the context of high-fidelity network modeling: (1) The target application is studied and segmented into independent parallel sections. Each of these section can be considered as a kernel and implemented as a fragment shader in GPUs. The inputs and outputs of each kernel program are mapped and stored as 2D textures in GPUs' texture memory. (2) In GPUs, specify area for rendering, which is equivalent to define range of arrays or matrix for output. (3) The rasterizer generates a fragment for every pixel location. Each element in the output arrays or matrix is bound to a unique pixel location. (4) All fragments are processed parallel as defined in the fragment shader, which means that each element in the output arrays or matrix is applied with the same operation, e.g. addition, multiplication or other compound ops. (5) A value or vector of values per fragment are rendered to off-screen buffer, which is output for one pass. The rendering buffer can be used as input texture for next pass. It was a daunting task to write vertex and fragment shader or program by using graphics hardware-dependent assembly languages. Many high-level shading languages, e.g. Cg, GLSL, HLSL, Sh, Brook etc., have been developed to facilitate GPU programming.

Exploiting GPUs for network simulations has its own challenges. First, systematically identify the data parallelism inherent in network simulations. Fujimoto et al. [5] suggest that in parallel discrete event simulation (PDES) the event-level causal constraints limit use of conventional parallel computing; in contrast, our study targets the data parallelism. Second, software abstraction is a necessity for using and developing GPU-accelerated simulation models; network researchers are not familiar with GPUs-based computing but demand short evaluation cycle. This dictates a programming framework for GPUs with well-defined APIs that can be easily used by network modelers.

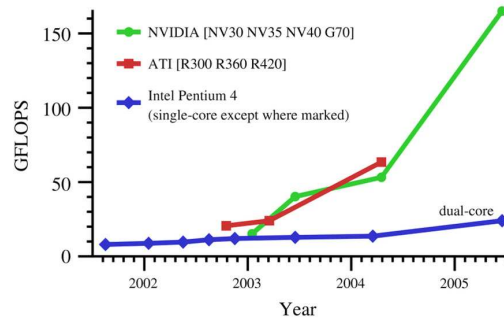


Figure 1. Computational power of GPU and CPU (courtesy of Ian Buck, Stanford Univ.)

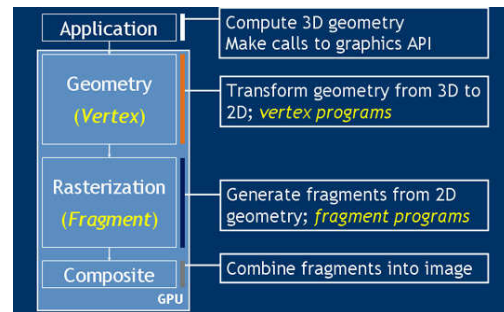


Figure 2. Simplified graphics hardware pipeline

This paper presents our experiences with using computer graphics hardware for acceleration of simulation-based performance evaluation of network protocols and applications. The rest of the paper is organized as follows. Section 2 presents the framework for GPU-accelerated network simulation. In Section 3, we describe implementations of two GPU-accelerated simulation models. In Section 4, we compare performance of the GPU-based simulation models with the CPU-based counterparts. More than 10 times speedup in large scenarios is observed in both cases without loss in evaluation accuracy. Further, we discuss the circumstances where GPU-based implementations outperform CPU-based ones as well as the limitations on the use of GPUs in Section 5. We summarize related work including hardware-assisted wireless network simulation, multi-paradigm evaluation platform, and uses of GPUs for general-purpose computing, and highlight distinction between our work and existing studies in Section 6. Section 7 concludes the paper.

2 Architecture for GPU-accelerated network simulations

In this section, we propose a framework for GPU-accelerated network simulation. The goal is to design an evaluation platform architecture that efficiently utilizes the computational processors of GPUs and CPU, I/O, memory and other resources available in commodity desktops. And

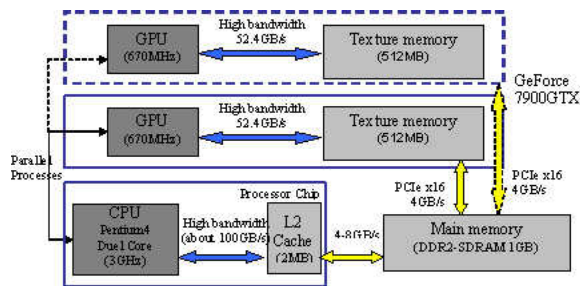


Figure 3. Commodity desktop equipped with multiple GPUs

the architecture must be extensible and composable. First, we can extend the evaluation platform by exploiting advances in more parallel computing hardware and architectures, e.g. application-specific FPGA [17], Cell processor etc. Second, modular design for the evaluation platform can not only encapsulate operations of individual parallel hardware but also make easy applying appropriate computing partition policies, either automatically or manually. A run-time library for high-fidelity networking models in GPU is provided so that average network researchers can easily use and develop GPU-accelerated modules.

In a commodity desktop equipped with off-the-shelf graphics cards, as illustrated in Figure 3, CPU and GPU units can be exploited simultaneously for computing; with nVidia SLI technology, two or more GPUs can be integrated in a single system. CPUs and GPUs are suitable for different types of computing due to significant differences in their architecture. A CPU has extremely fast but small on-chip cache, e.g. over 100GB/sec for off-the-shelf CPUs, fine branching granularity, support for lots of different processes or threads, and high performance on a single thread of execution. In contrast, a GPU has many more arithmetic units and support for extremely high data parallel and instruction parallel execution.

The evaluation process of high-fidelity network modeling involves both task-parallel and data-parallel computing. As observed in a typical wireless network simulation, the simulation engine, consisting of event scheduler, queue manager, statistics collector etc., mainly deals with scheduling and management of processes or threads, which are well suited to task parallel computation offered by traditional multi-CPU architectures. In addition, we have identified three sources of data parallelism, which can make best use of GPUs. First, analytical approximation of network (traffic) dynamics may be modeled with a set of linear algebra operations, which can effectively exploit the data parallelism of a GPU. Second, detailed simulation of physical layer involves many compute-intensive operations, e.g. FFT/IFFT, adaptive beamforming etc. Many RF radios/devices themselves are source of data parallelism, e.g. antenna arrays and sub-carriers in OFDM system. Third,

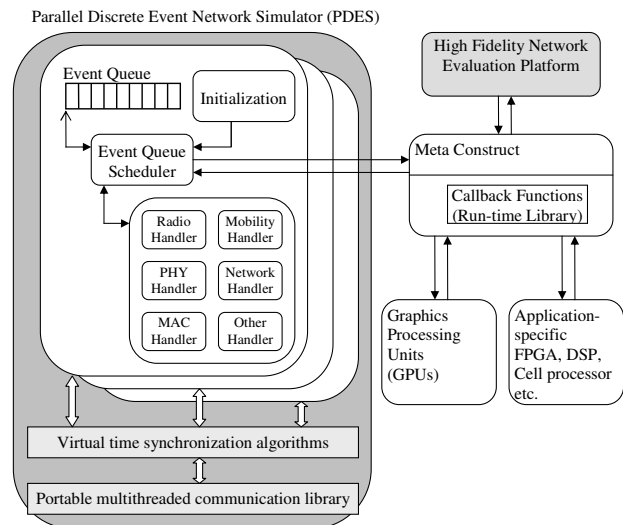


Figure 4. Framework for hardware acceleration

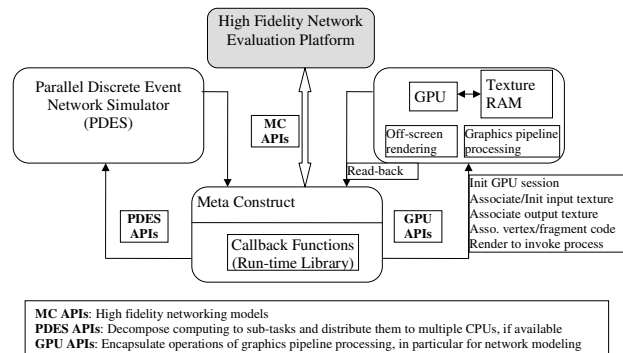


Figure 5. Interfaces between main components in GPU-accelerated evaluation platform

aggregation of simulation events, typically in PHY layer, can contribute to high degree of data parallelism, e.g. simultaneous packet transmissions.

Based on the above observation, the hardware accelerated simulation platform is structured for simultaneous use of CPUs and other co-processors, e.g. GPUs, as shown in Figure 4. The high fidelity network evaluation platform is built on top of Meta Construct (MC), where two types of information are registered, manually or automatically. The first category is related to the available computing sources, e.g. number of CPUs, availability of GPUs, functions provided by application-specific FPGAs, relevant characteristics of computation overhead etc. The second category is related to the computing profiling of high fidelity network models, e.g. availability of data parallelism. The MC interfaces with DES engine and other available co-processors, e.g. GPUs etc, and is responsible for distributing to the DES engine and other co-processors both task-parallel and data-parallel computing tasks identified in high fidelity network models. The detailed interaction is illustrated in Figure 5.

The DES engine is extensible to all parallel computing architectures, e.g. CMP, SMP and grid computing. The current evaluation platform is designed and implemented only for shared memory computing architecture. In particular, Parsec [1] is the parallel simulation language used in the evaluation platform. The network simulation can be partitioned, either horizontally or vertically, into multiple top-level entities, which are synchronized by using any existing virtual time synchronization protocols. These protocols are implemented on top of portable multi-threaded communication library, e.g. MPC, MPI etc.

The GPU APIs encapsulate the operations of GPUs, including the initialization of GPU session, association of input/output parameters with the texture memory in GPUs, graphics pipeline processing and off-screen rendering. So the internal data representation and hardware-dependent operations on GPUs remain transparent to average network researchers while APIs to matrix/vector based numerical operations and other primitive simulation models in GPUs are exposed and can be used as the building blocks for developing GPU-accelerated simulation modules. Further, a runtime library for high fidelity network models in GPUs is provided together with the evaluation platform. The key distinctions from existing GPU studies for general computing include the computational efficiency and the run-time library for operations commonly used in network modeling, which will be elaborated in Section 3.

The PDES APIs encapsulate the operations of task decomposition and distribution between multiple CPUs. In our evaluation platform, the event manager in the DES engine is enhanced to identify set of events, which can be aggregated into a single data-parallel operation. Subsequently, the MC uses the PDES APIs to interact with the enhanced event manager to re-delegate the evaluation of those events to a data-parallel co-processor. To determine whether the computing aggregation and migration is necessary in terms of performance gain, the MC considers not only the availability and computing characteristics of co-processors but also the performance profiling of the corresponding computation in the evaluation platform host. The intuition behind this mechanism is, for example, that operations of several small matrixes can be aggregated into operations of a big one, which would be more suitable for data-parallel co-processors, e.g. GPUs. In the implementation of our evaluation platform, the decision-making can be performed either based on performance threshold values that are manually configured or based on the values that are automatically tuned for the evaluation platform host.

3 Case studies and implementation

Good candidate for GPU acceleration should have two features: highly data-parallel and arithmetic-intensive. As

discussed above, there are many sources of data parallelism that can be exploited in network simulations. In the paper, in order to demonstrate that use of GPUs can speed up network simulations, we present two case studies that respectively focus on different layers of the network protocol stack. The first is multi-paradigm simulation of mixed wired and wireless networks, which uses an analytical representation to model the wired component. This study shows how a fluid-flow-based TCP model can be accelerated and hence usefully integrated into the mixed network model. The second study is a high-fidelity PHY layer discrete event model for wireless network using a detailed antenna model. By evaluating simulation results and execution time of the models, we observe GPU-implemented version has achieved significant speedup without observable accuracy degradation in simulation results.

3.1 Fluid-flow-based TCP model

The fluid-flow-based TCP model proposed by Misra et al [12] is accurate in predicting the traffic dynamics at active queue management routers for wired backbone networks with high volumes of traffic. In summary, interactions of a set of TCP flows and active queue management routers are modeled with jump process driven Stochastic differential equations, which can be transformed into a set of ordinary differential equations (ODEs). The CPU-based implementation uses a ODE solver, ODE45, provided in Matlab. Our GPU-based version maps all data structures in CPU to on-board memory in GPU and implements the algorithm based on the GPU stream programming model.

This model has been integrated into a multi-paradigm modeling environment: MAYA [20], designed to evaluate mixed wired and wireless networks. J. Zhou et al demonstrate the mixture of QualNet and fluid flow model does not affect the validity of network modeling results. However, the integration relies on having accurate representations of the time varying state of the routers (characterized by the instantaneous queue lengths) that are used at the boundary between the wired and wireless networks. This in turn, requires that the ODE solvers be recomputed periodically and the execution speed of the mixed model is significantly affected by the execution speed of the ODE solvers. By implementing the solver in the GPU, we can significantly increase the size of the networks that can be evaluated.

In this case study, the TCP module will be singled out for performance evaluation and comparison between CPU-implemented version and GPU-accelerated one. The insignificant differences in TCP dynamics predicted by the two versions further lead to the conclusion that the mixture of QualNet and fluid flow model in the MAYA framework does not affect the validity of network modeling results, as observed and reported before.

3.2 Adaptive antenna model

In high-fidelity wireless network simulation, a significant portion of computation cost can be attributed to execution of detailed physical models, e.g. detailed OFDM model as shown in [18]. The adaptive antenna algorithm [9, 2] for OFDM system is another computation-intensive algorithms. However, we observe that number of elements in the antenna array and number of sub-carriers in the OFDM system constitute high degree of data-parallelism, which can be best exploited by GPUs.

The antenna adaptation algorithm recursively updates the weights of the beamformers in the direction minimizing mean squared error (MSE), whose criterion is the error signals between the pilot symbols and the corresponding received signal. In practice, the MSE can be calculated either in the time domain or in the frequency domain. In our implementation, we choose recursive least squares (RLS) estimation [6] as the filtering algorithm. To make efficient use of GPUs, we carefully design and implement the data layout and operations of arrays of complex numbers in a GPU.

3.3 Implementation details

To implement simulation models in GPUs and facilitate development of future GPU-based models, we have to address several technical issues related to commercial graphics cards; the three primary ones are highlighted below.

3.3.1 Representation and operations of arrays of complex numbers in GPU

Arrays of real numbers in CPU-based algorithms are represented in GPUs as 2D *textures* of floating point data values. Each element in a texture, or *texel*, corresponds to a *pixel*. The fragment shader is executed for every pixel. In commercial GPUs, e.g. nVidia GeForce 7900GTX, processor instructions are optimized for 4-tuples of floating point values. That's say each texel is associated with four color channels, i.e. red, blue, green and alpha. So to ensure efficient use of GPUs' computational capacity, every four elements of an array in CPU shall be packed into one texel in GPU's texture memory. Take a 16-element array for example. It may be represented as a texture of width 2 and height 2. Figure 6 illustrates an example of fragment shader written in GLSL, an implementation of multiplication.

For arrays of complex numbers, there are two types of representation. First, the real parts of all numbers in the array are mapped to a texture while the imaginary parts to another. Second, each complex number in the array is mapped to two components of a texel so the entire array will be represented in one texture. The first representation is inefficient for operations on GPUs. When we multiply two

```
uniform sampler2D g_sVector;
uniform sampler2D g_sVector2;

void main() {
    gl_FragColor = texture2D(g_sVector, gl_TexCoord[0].xy)
    * texture2D(g_sVector2, gl_TexCoord[0].xy);
}
```

Figure 6. Fragment shader for multiplication of real numbers written in GLSL

```
uniform sampler2D g_sVector;
uniform sampler2D g_sVector2;

void main() {
    vec4 valA = texture2D(g_sVector, gl_TexCoord[0].xy);
    vec4 valB = texture2D(g_sVector2, gl_TexCoord[0].xy);
    vec4 temp1 = valA * valB;
    vec4 temp2 = valA * valB.yxwz;
    vec4 temp3 = vec4((temp1.xz - temp1.yw), (temp2.xz +
temp2.yw));
    gl_FragColor = temp3.zxyw;
}";
```

Figure 7. Fragment shader for multiplication of complex numbers written in GLSL

arrays, the former representation requires 4 input textures and 2 rendering textures while the later only needs half of the amount. All commercial GPUs have constraint on the number of input and rendering textures, e.g. 8 in nVidia GeForce 7900GTX. So the second representation is chosen for the implementation. The implementation of multiplication is illustrated as an example in Figure 7, in which x, y, z and w refer to the four components of a texel.

Based on the data structure proposed above, we define and implement a C++ class *clComplexVector*, which encapsulates the data layout and all algebraic operations in GPU. The code fragment, as shown below, illustrates the member functions initializing and reading-back array of complex numbers in GPU. A matrix carrying complex numbers is represented as a set of diagonal vectors. Its operations are encapsulated in a separate C++ class *clComplexMatrix*.

```
class clComplexVector {
    void setData(float fData_RealPart[],
                float fData_ImgPart[]);
    void getData(float fData_RealPart[],
                float fData_ImgPart[]);
    ...
}
```

3.3.2 Outputs of GPU-based function calls

Data transfer between CPU and GPU constitutes high communications cost. To minimize the cost, we avoid reading back intermediate results whenever possible. Additionally, intermediate results can be stored in GPU's texture memory and used for subsequent passes of computation.

For GPU-based function calls with two or more outputs, we use multiple rendering targets (MRTs), a technique

available in most off-the-shelf GPUs to write multiple textures in a single pass. However, there is constraint on the number of rendering textures, e.g. 8 in nVidia GeForce 7900GTX. To bypass this constraint, we employ two techniques in our implementation. First, we combine multiple outputs and render them to one texture, especially when they will be used together in subsequent passes. Second, we break a compound operation with many outputs into a sequence of operations each with less number of outputs. In practice, due to the efficient representation of arrays of complex numbers, we rarely have to break up a compound operation to meet the constraint.

The process of rendering textures in GPUs is encapsulated in C++ class implementation so MRTs remain transparent in the software abstraction provided by our APIs.

3.3.3 Abstraction or customization: implementation of fragment shader

We have seen several recent work, e.g. [11] [16] etc., aimed for automating the mapping of data structures from CPUs to GPUs and encapsulating common linear algebra operations in GPUs by APIs. However, over-simplified use of the existing APIs could result in poor utilization of GPUs' computational power. A compound operation involving multiple textures may be broken into several simple linear algebra operations in GPUs. So corresponding standard APIs are called sequentially and in GPUs several passes are needed for the compound operation. As summarized above, the GPUs have a fix-staged pipeline including geometry processing, rasterization etc. And for general-purpose computing, most calculations are implemented in the stage of rasterization. So the more passes, the more overhead. Worse, more passes means more intermediate states rendered.

In our implementation, all compound operations are customized with individual fragment shader as long as they could. Also it is technically practical since there is no longer strict constraint on length of fragment shader in most off-the-shelf GPUs. As a result, we observe huge performance gain immediately benefiting from the extensive customization. Take as an example the multiplication of a vector u and a matrix A , i.e. $u * A$. The matrix A is represented as a set of diagonal vectors. As shown in Equation 1, each iteration involves two operations: one multiplication and one addition. When we implement the two operations in one fragment shader, we achieve more than 50% saving in execution time.

$$u * A = \sum_{i=1}^n \text{right_shift}(X_i, n - i + 1), \text{ where} \quad (1)$$

$$X_i = u * A_i, \text{ where}$$

$$A_i = [A_{1i}, A_{2((i+1)\%n)}, \dots, A_{n((i+n)\%n)}]$$

4 Results

In this section, we will evaluate the performance of the GPU-based simulation models from two perspectives: accuracy of simulation results and speedup in execution time.

The hardware platform used for the results is a Dell Dimension desktop configured with Intel (dual core) 3GHz Pentium 4 CPU and 1GB DDR2 memory. The simulation models are implemented in the nVidia GeForce 7900GTX graphics card with 512MB texture memory. The vertex and fragment program (shader) are programmed with OpenGL and GLSL.

In both cases, we observe that GPU-based simulation models predict accurate results and achieve significant acceleration. In some large scenarios, the GPU-based implementations deliver more than 10 times speedup, compared with CPU-based counterparts.

4.1 Evaluating fluid-flow-based TCP models

The scenario used in our evaluation is characterized by number of TCP flows and number of queues (AQM routers). The smallest scenario is configured with four TCP flows and four queues while the largest is with 1024 flows and 1024 queues. The average number of queues that a TCP flow passes through ranges from 2 in small scenarios to 8 in large scenarios. The network topology used in all the experiments are generated randomly.

In Figure 8 and 9, we present the dynamics of TCP window sizes and average queue lengths predicted by GPU-based simulation and CPU-based simulation for a scenario with 4 flows and 4 queues. Visually and numerically, we observe insignificant differences in simulation results between the two versions of implementation. In order to evaluate the impact of the distinction in TCP traffic dynamics predicted by the two versions, for each 10ms, we collect one sample value on the difference. Figure 10 plots the distribution of the difference in predicted TCP window size for flow 1 and average queue length for queue 1 respectively. With the insignificant difference predicted by the two versions of analytical modules, the mixed modeling paradigm still predicts same network dynamics. Good match is also observed in medium and large sized scenarios. Plots on the dynamics of data loss rate, instantaneous queue lengths are omitted here due to space limit.

In Figure 11, we compare the execution time of the two implementations. The left diagram illustrates the execution time of a single iteration of the ODE solver. The GPU-based implementation outperforms the CPU version in scenarios with 256 flows and 256 queues or more. The right diagram plots the overall simulation time, which shows that the GPU-based one costs less time than the CPU-based one

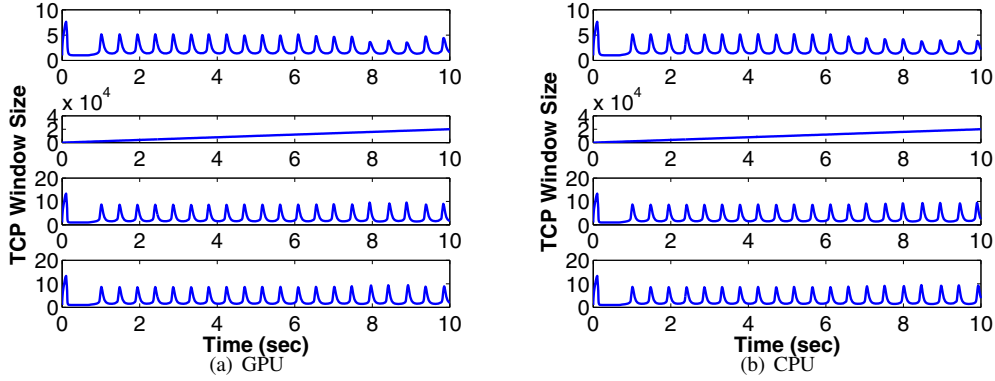


Figure 8. Prediction in dynamics of TCP window size (flow 1 - flow 4)

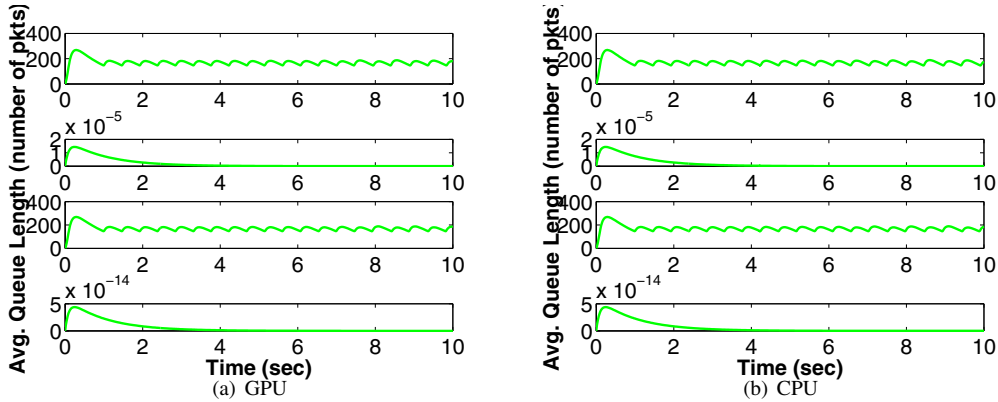


Figure 9. Prediction in dynamics of average queue length (queue 1 - queue 4)

in scenarios with more than 256 flows and 256 queues. The discrepancy is explained by larger number of iterations in GPU-based ODE solver. Our in-depth performance analysis shows lack of support for double-precision floating points in GPU could result in slower convergence in ODE solver.

4.2 Evaluating adaptive antenna systems for OFDM

In this sub-section, we evaluate the effect of an adaptive beamforming algorithm. For the single transmission link, we use QPSK modulation scheme, multiplicative channel distortion, and frequency-domain equalizer.

In Figure 12, we present the signal constellation diagram with and without the adaptive-beamforming based equalizer. The receiver is configured with an antenna array of size 16 and only one subcarrier is used in the transmission. Both the GPU-based simulation and the CPU-based version have correctly produced the matching prediction.

In Figure 13(a), we compare the execution time of simulations. The receiver is configured with varying sizes of antenna arrays and one subcarrier is used for transmission. We observe that the GPU-based simulation runs faster than the CPU-based one when the antenna array size exceeds 256.

In realistic OFDM systems, it is quite common to have a larger number of sub-carriers. And as proposed in [2], individual beamformers are used for different sub-carriers so that each beamformer can adjust its own weights to adapt to different interference patterns that may be experienced in different bands. With this setup, QPSK-modulated OFDM symbols are transmitted using a number of subcarriers, which is varied from 4 up to 512 in our experiments. The signals are applied channel distortion and then equalized by individual beam-formers in the receiver. In Figure 13(b), we compare the simulation time, which is normalized over 1600 OFDM symbols. The square-marked line corresponds to CPU-based simulation configured with an antenna array of size 4. And the other three dashed lines correspond to GPU-based simulations with antenna arrays of sizes 4, 16 and 64 respectively. It is observed that the execution time of CPU-based implementation stays flat while that of the GPU-based version linearly decreases with respect to the number of sub-carriers. The decrease is due to the parallel processing for all sub-carriers in the GPU.

The GPU-based version effectively exploits the parallelism in scenarios with large number of sub-carriers, which have even bigger implications for network-level simulations since concurrent data transmission can be widely observed

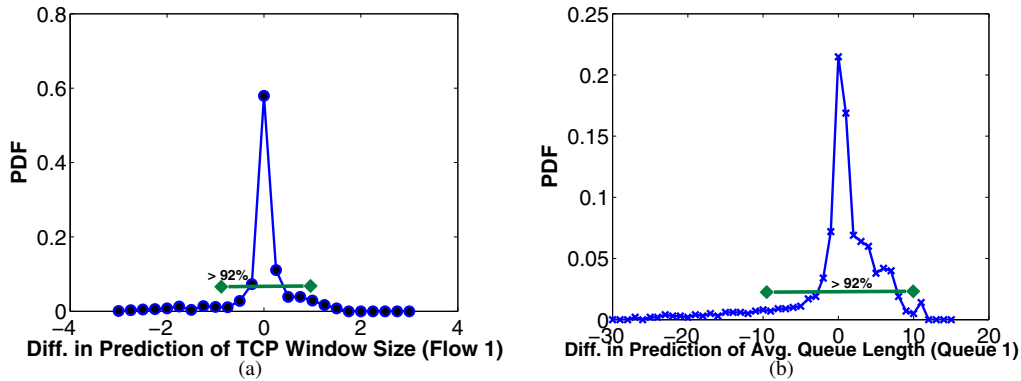


Figure 10. Difference in prediction of traffic dynamics (sample rate: 10ms)

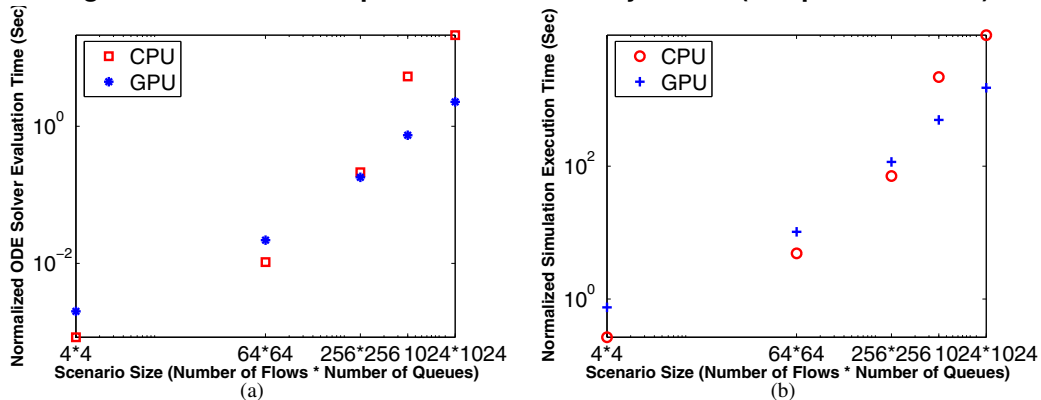


Figure 11. Performance comparison between GPU accelerated simulation and CPU based simulation of fluid-flow models

in wireless MANET scenarios. It will be further elaborated how aggregation of PHY layer events may lead to best use of GPUs' parallel processing in the end of Section 5.

5 Discussion

As revealed in our investigation, GPUs have their own limitations for network simulations. Not all simulation models can adequately exploit the parallel processing capabilities of GPUs. The process of encoding and decoding in communication system is a case in point. We implement Viterbi coding scheme using GPUs but the performance evaluation shows no clear speedup. In summary, the GPU based processing for simulation is not effective when they involve the following types of computations: First, the processes constitute largely sequential operations. Second, the processes require bit-wise operations, which are not naturally supported in commercial GPUs. Other specialized DSP platform, e.g. FPGA-based processor, may help for such applications. For example, researchers have used FPGA-based DSP platform to emulate the radio propagation with detailed channel model [7]. In a general hardware-accelerated simulation platform, execution of dif-

ferent types of models may be delegated to different processors, e.g. GPU, cell processor, FPGA-based DSP etc.

Further, GPU-based acceleration is usually observed in medium to large scenarios. The overhead due to slow data transfer between GPU and CPU and the fixed-stage graphics hardware pipeline offsets performance gain in small scenario. So we suggest that GPU based acceleration may not be suitable for real-time evaluation. Instead, our evaluation platform is more aimed for scaling network simulations or increasing fidelity of modeling physical layer phenomena.

With our design for the evaluation platform, events in a network scenario can be aggregated to exploit the parallel processing capability of GPUs. As shown in Section 4.2, the processing cost of a packet is amortized among sub-carriers. Consequently, the average processing time of a fixed-sized packet decreases with an increase in the number of sub-carriers. In wireless MANET scenarios, simulation of concurrent packet transmissions can be aggregated in a similar way. For example, if there are two packet transmissions in two uncorrelated links and both require detailed antenna models for evaluation accuracy in transmission errors, operations of several small matrixes implemented for the antenna model can be packed together and converted

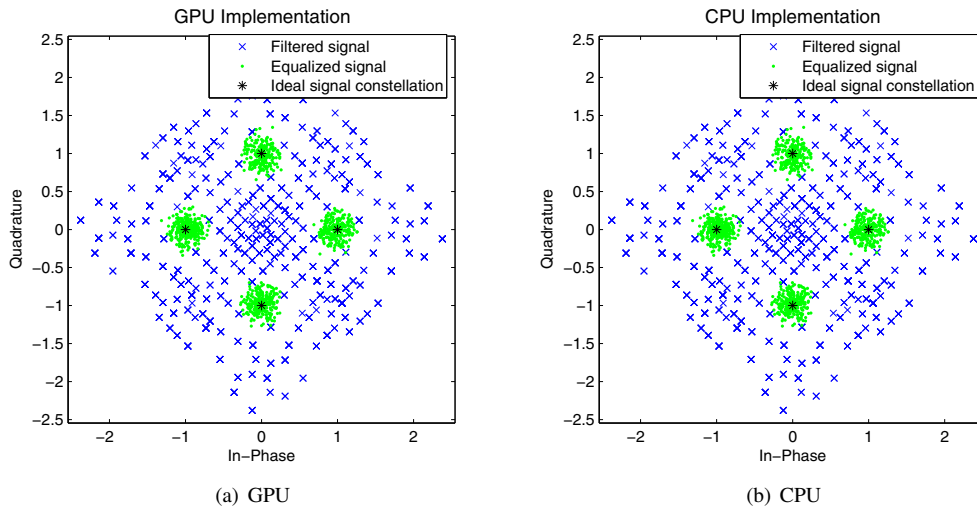


Figure 12. Signal constellation diagram w/ and w/o adaptive-beamforming based equalizer

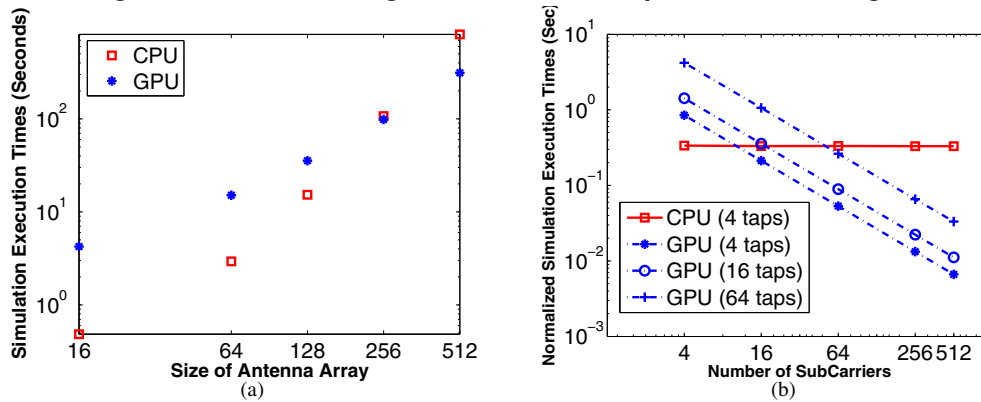


Figure 13. Performance comparison between GPU accelerated simulation and CPU based simulation of adaptive antenna systems

to operations of fewer larger matrixes. Consequently, the average processing time of a packet in the GPU-based implementation will be significantly reduced.

6 Related work

High-fidelity simulation-based performance evaluation of wireless networks demands detailed simulation models [15]. The increasing details in simulation result in more computation and longer evaluation cycles. Judd et al [7] propose an FPGA-based DSP engine to emulate RF signal propagation. With the incorporation of external hardware simulation or emulation engine, existing wireless simulators could achieve improved accuracy without prohibitive increase in computation cost. The distinction between our work and others' is alternative computing hardware and architecture are studied in a general evaluation framework.

Multi-paradigm evaluation framework, which combines

the benefits of physical testbed, simulation and emulation, is extensively studied in MAYA [20] and TWINE [19] by J. Zhou et al. Our evaluation platform is focused on exploiting innovation in computing hardware and architecture to improve computational efficiency.

Owens et al [13] conduct extensive surveys on the use of graphics hardware for general-purpose computations. More can be found in <http://gpgpu.org>. In particular, Perumalla et al. [14] first use GPGPUs for discrete event simulation (DES). Fan et al. [3] build a GPU-based cluster for scientific computing and achieve much better performance than solutions based on a CPU cluster. In complementary to existing work, we explore how to harness the computing horsepower of GPUs for high fidelity networking modeling.

To harness the horsepower of GPUs, needed are experiences with graphics APIs, e.g. OpenGL, DirectX etc., and good understanding on the graphics hardware pipeline in GPUs. Solutions are proposed to simplify programming of

GPUs. Krüger et al [11, 10] propose a GPU framework for solving systems of linear equations, which provides abstract representations of high-level data structures, e.g. vectors, and some linear system operations. Tarditi et al [16] provide a high-level data-parallel programming model as a library. Although we have used the preceding efforts, our work is focused on identifying components in network simulation that can be accelerated by using GPUs and devising a general network simulation architecture incorporating use of GPUs and alternative computing hardware architectures. More, the efficiency of using GPUs is explicitly evaluated and improved for high fidelity network modeling.

7 Conclusions and future work

In this work, we explore use of commercial graphic cards (GPUs) for network simulations. First, we proposed a general simulation architecture using multiple CPUs, GPUs, and potentially extensible to other computing processors, to speed up execution of network simulations. Software abstractions are designed to transparently integrate GPU-based simulation modules with existing CPU-based ones and to facilitate use and development of GPU-based models. Second, we develop and implement two GPU-based simulation models to evaluate the benefits of using GPUs for protocol and system performance evaluation. The case study clearly demonstrates that GPU-implemented version can effectively speed up execution of simulation without observable degradation in accuracy of performance predictions. In some large scenarios, we achieve more than 10 times speedup. On the other hand, in network simulations there are also detailed models involving largely sequential operations or requiring extensive bit-wise operations, which are shown not suitable for GPUs-based acceleration. In conclusion, our study suggests that high fidelity network simulations can be accelerated by parallel use of CPU and GPU units available in off-the-shelf hardware. Guidelines proposed above must be followed to ensure best use of them in our high fidelity network evaluation platform.

As ongoing implementation effort, we are integrating GPU-implemented modules into existing simulation-based network evaluation platform using the architecture proposed above. And aggregation of events in PHY-layer simulation modules is specifically supported in order to exploit more data-parallelism. We're also investigating inclusion of FPGAs and cell processors for network simulations.

References

[1] R. Bagrodia, R. Meyer, M. Takai, Y. Chen, X. Zeng, J. Martin, B. Park, and H. Song. Parsec: A Parallel Simulation Environment for Complex Systems. *Computer*, 31(10):77–85, October 1998.

[2] B.-L. P. Cheung. Simulation of adaptive array algorithms for OFDM and adaptive vector OFDM systems. Master's thesis, Virginia Polytechnic Institute and State University, 2002.

[3] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU Cluster for High Performance Computing. In *ACM / IEEE Supercomputing Conference*, 2004.

[4] R. Fernando, M. Harris, M. Wloka, and C. Zeller. Programming Graphics Hardware. In *EUROGRAPHICS (Tutorial)*, 2004.

[5] R. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, 1990.

[6] S. Haykin. *Adaptive filtering theory*. Prentice Hall, 2002.

[7] G. Judd and P. Steenkiste. Using Emulation to Understand and Improve Wireless Networks and Applications. In *Proc. Usenix and ACM NSDI*, 2005.

[8] B. Khailany, W. J. Dally, S. Rixner, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, and A. Chang. Imagine: Media Processing with Streams. *IEEE Micro*, Mar/April 2001.

[9] C. K. Kim, K. Lee, and Y. S. Cho. Adaptive beamforming algorithm for OFDM systems with antenna arrays. *IEEE Transactions on Consumer Electronics*, 46(4):1052–1058, November 2000.

[10] J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics (TOG)*, 22(3):908–916, 2003.

[11] J. Krüger and R. Westermann. A GPU Framework for Solving Systems of Linear Equations. In *GPU Gems 2*, chapter 44, pages 703–718. Addison-Wesley, 2005.

[12] V. Misra, W.-B. Gong, and D. Towsley. Fluid-based Analysis of a Network AQM Routers Supporting TCP Flows with an Application to RED. In *Proc. ACM Sigcomm*, 2000.

[13] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. In *State of the Art Reports, Proc. EUROGRAPHICS*, 2005.

[14] K. S. Perumalla. Discrete-event Execution Alternatives on General Purpose Graphical Processing Units (GPGPUs). In *Proc. PADS'06*.

[15] M. Takai, J. Martin, and R. Bagrodia. Effects of wireless physical layer modeling in mobile ad hoc networks. In *Proc. ACM MobiHoc*, 2001.

[16] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: simplified programming of graphics processing units for general-purpose uses via data parallelism. Technical Report MSR-TR-2005-184, Microsoft Research, 2005.

[17] T. D. Vancourt. *LAMP: Tools for Creating Application-specific FPGA Coprocessors*. PhD thesis, Boston Univ., 2006.

[18] G. Yeung, M. Takai, R. Bagrodia, A. Mehrnia, and B. Daneshrad. Detailed OFDM Modeling in Network Simulation of Mobile Ad Hoc Networks. In *Proc. PADS'04*.

[19] J. Zhou, Z. Ji, and R. Bagrodia. TWINE: A Hybrid Emulation Testbed for Wireless Networks and Applications. In *IEEE INFOCOM*, 2006.

[20] J. Zhou, Z. Ji, M. Takai, and R. Bagrodia. MAYA: Integrating Hybrid Network Modeling to the Physical World. *ACM Transactions on Modeling and Computer Simulation*, 14(2):149–169, April 2004.