Network Simulator Tools and GPU Parallel Systems

Leonid Djinevski, Sonja Filiposka, and Dimitar Trajanov

Abstract—**In this paper we discuss the possibilities for parallel implementations of network simulators. Specifically we investigate the options for porting parts of the simulator on GPU in order to utilize its resources and obtain faster simulations. We discuss few issues which are unsuitable for the GPU architecture, and we propose a possible work around for each of them. We introduce a design of parallel module that interconnects with a network simulator, while maintaining transparency in aspect of the simulation modeler.**

Index Terms—**Network Simulator Tools, HPC, GPGPU, CUDA, OpenCL.**

I. Introduction

NETWORK simulators are tools used by researchers in order to test new scenarios and protocols in a controlled and reproducible environment, allowing the user to represent various topologies, simulate network traffic using different protocols, visualize the network and measure the performances. Although network simulators are very useful, most of the widely used network simulators do not scale [1]. Simulation of medium to large networks results in a long simulation time which is not practical for investigating protocols.

With the development of parallel systems, significant processing power is becoming available. The single instruction, multiple data (SIMD) models of parallel systems, more particular the Graphics Processing Units (GPUs) have provided a massive acceleration. Additionally, the low cost of these units have brought a huge performance in the insides of regular personal computers (PCs). The first attempts for utilizing the GPU hardware for general purpose computing proved to be a very complicated process [2]. However, with development of the Compute Unified Device Architecture (CUDA) programming model in 2007 [3], and also with the publishing of the standard Open Computing Language (OpenCL) late 2008 [4], general purpose computing on graphics hardware has significantly improved. Therefore, many general purpose applications have been ported for the GPU architecture.

Network simulators have traditionally been developed for execution on sequential computers. Developing a parallel implementation for a network simulator is not straight forward. There are many architectural issues that have to be taken in to account and they might prevent the complete utilizing of the GPU resources.

In this paper we review few of the most widely used network simulators. We also discuss the possibilities for parallel implementations of network simulators. Specifically we investigate the options for porting parts of the simulator on GPU in order to utilize its resources and obtain faster simulations. Additionally, we identify modules which carry the biggest workload as well as possible, issues that make the network simulators unsuitable for the GPU architecture, and we propose resolutions to work around these issues.

This rest of this paper is organized as follows: We review implementations of network simulator tools in Section 2, followed by a short overview of the GPU computing in Section 3. In Section 4 we identify which modules of the network simulator contain intensive workloads. Also in this Section we propose a framework which will utilize the GPU resources. In Section 5 we analyze performance, and we conclude and propose future work in Section 6.

II. Related Work

There are two types of approaches for developing a parallel network simulator. One can create the parallel simulator from scratch, where all the simulation software is custom designed for a particular parallel simulation engine. For this approach a significant amount of time and effort are necessary to create a useable system. This is so, because new models must be developed, and therefore validated for accuracy.

An example of this approach is the Global Mobile Information System Simulator (GloMoSim), which is a scalable simulation library designed at UCLA Computing Laboratory to support studies of large-scale network models, using parallel and/or distributed execution on a diverse set of parallel computers [5]. GloMoSim beside sequential adopts parallel simulation model using libraries and layered API. The libraries are developed using PARSEC [6], which is a parallel C based programming language which uses message based

Manuscript received 1 May 2012. Accepted for publication 30 May 2012. Some results of this paper were presented at the 4th Small Systems Simulation Symposium, Niš, Serbia, February 12-14, 2012.

Leonid Djinevski, Sonja Filiposka and Dimitar Trajanov are with the E-TNC Research Group, Faculty of Computer Science and Engineering, Ss. Cyril and Methodius University, Rugjer Boshkovikj 16, 1000 Skopje, Macedonia, E-mail: {leonid.djinevski, sonja.filiposka, dimitar.trajanov}@finki.ukim.mk.

DOI: 10.7251/ELS1216081D

approach.

Another example is the Scalable Simulation Framework (SSFNet) which claims that is a standard for parallel discrete event network simulation [6, 7]. SSFNET's commercial Java implementation is becoming popular in the research community, but SSFNet for C++ (DaSSF) does not seem to receive nearly as much attention, probably due to the lack of network protocol models. It is a high performance network simulator designed to transparently utilize parallel processor resources, and therefore scales to a very large collection of simulated entities and problem sizes.

The second approach for developing parallel/distributed simulation involves interconnecting with existing simulators. These federated simulations may include multiple copies of the same simulator (modeling different portions of the network), or entirely different simulators. Few parallel implementations of this approach are presented in the following.

The NS-2 Simulator [8] is widely used in the networking research community and has found large acceptance as a tool to experiment new ideas, protocols and distributed algorithms. It is a discrete event driven sequential network simulator, developed at UC Berkeley by numbers of different researchers and institutions. NS-2 is suitable for simulating and analyzing either wired or wireless network sand is used mostly for small scale simulations. NS-2 is written in C++ and OTcl. The users define the network topology structure, the nodes, protocols and transmitting times in an OTcl script. The open source model of NS-2 encourages many researchers from institutions and universities to participate and contribute to improve and extend the project. NS-2 plays an important role especially in the research community of mobile ad hoc networks, being a sort of reference simulator [9]. Adding new network objects, protocols and agents requires creation of new classes in C++ and then linking them with the corresponding OTcl objects.

A parallel simulation extension for the traditionally widely used NS-2 simulator has been created at the Georgia Institute of Technology (PADS Research Group), but it is not in wide use. The Parallel/Distributed NS (*PDNS*) [10] was designed to solve the NS-2 problems with large scale networks by running the simulator on a network of workstations connected either via a Myrinet network, or a standard Ethernet network using the TCP/IP protocol stack. In that way the overall execution time of the simulation should be at least as fast at the original single workstation simulation, allowing simulating large scale networks.

Georgia Tech Network Simulator (GTNetS) is a network simulation environment which uses C++ as a programming language [11]. GTNetS is designed for studying the behavior of moderate to large scale networks. The simulation environment is structured as an actual network with distinct separation of protocol stack layers.

OMNeT++ is a network simulation library and framework, primary used for simulation of communication networks, but because of its flexible architecture can be used to simulate complex IT systems too. OMNeT++ offers an Eclipse based IDE and the programming language used is C++ [12, 13].

In this paper we introduce a different approach for parallelizing network simulators that is based on federation simulations. In order to fully utilize the available hardware we investigate the possibility to port the computing intensive network simulator modules to the GPU and thus obtain faster simulation time.

III. GPGPU, CUDA, AND OPENCL

In this section we summarize some key fact of the GPU architecture so we can provide and discuss information about parallel module implementation of a network simulator. The origin of General-Purpose computing on Graphics Processing Units (GPGPU) comes from graphics applications, so in similar fashion, CUDA or OpenCL applications can be accelerated by data-parallel computation [14] of millions of threads. A thread in this context means an instance of a kernel, which is a program that is running on the GPU. This way, the GPU device can be visualized as a SIMD parallel machine. Therefore, understanding of the graphics pipeline to execute programs is not needed. In a nutshell, CUDA or OpenCL provide convenient memory hierarchy, allowing maximizing the performance, by optimizing the data access. The memory hierarchy of a GPU device is presented in Fig 1.

The GPU device has off-chip memory, so called global memory. Since this memory is separated from the GPU, a single fetching of data takes at least 500 cycles. This is the slowest memory on the device, and therefore the most expensive performance wise.

The next level in the memory hierarchy is the local memory, which is shared by a number of threads organized in work groups. This memory is very small 16 – 48KB, and it can be accessed almost as fast as register memory denoted in Fig. 1 as private memory which is exclusive to a single thread. Therefore, a program will compute correctly if there is no data dependence between threads in different work groups.

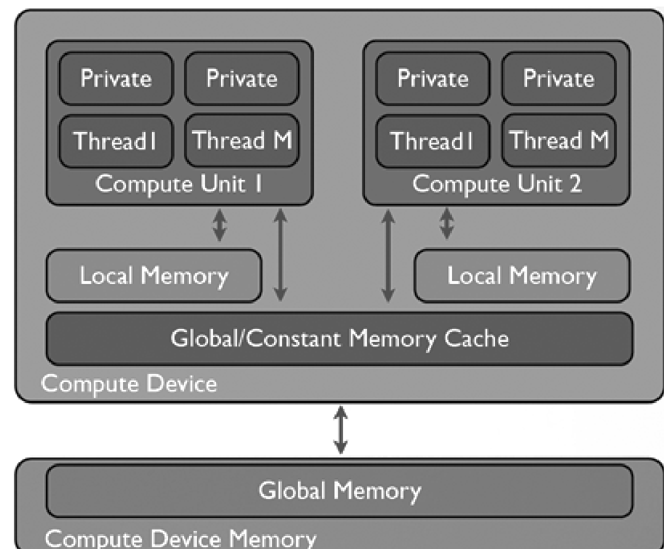

Fig. 1. GPU device memory hierarchy.

Exception is that within the same work group thread can have dependence because they can exchange data using the local memory.

IV. Network Simulator Modules

Network simulator algorithms are usually not so straight forward for mapping on the GPU, therefore we need to identify the workload of each module. The modules with the biggest workload are candidates for parallelization. Since, the GPU is a SIMD, in order to utilize the architecture, we look for segments of the algorithm code which are repeated regularly. Usually, these code segments are for loops or loops for which control flow can be predicted.

Once we identify which modules to parallelize, few issues have to be taken in to account. If the code segment works with small amount of data, the GPU device parallelism cannot be expressed. Another major issue is the control flow divergence. If the code segment contains much branching, the parallel code gets serialized, thus minimal or no performance increase is achieved. Nevertheless, in order to tweak the algorithm, few methods can be used to decrease the divergence. However, the worst divergence situation is presented in Listing I.

Listing I.
Unavoidable Divergence

```
if (condition 1)
    do this block of operations
else if (condition 2)
    do that block of operations
else if (condition 3)
    do some block of operations
else
    do any block of operations
```

In this case the divergence can cause up to 75% efficiency reduction, because the block of operation requires hundreds of instructions, thus making the algorithm unsuitable for SIMD parallel execution.

A. *Program Transformations*

In order to exploit more parallelism from the resources at hand, the program has to be transformed. The structure of the computations and their schedule need to be changes, so the program transformations will result with equivalent program which will have better performance.

Since data access is the most expensive part of the program execution, sometimes the program can be transformed so the data is not loaded from memory and calculated on the GPU device. In addition, another important factor is to have enough data to process in order to utilize the parallel resources. Therefore, it is prudent to introduce more calculation even if there are not needed at the moment, since in the following moments a requested calculation could already been obtained.

V. Performance Analysis

In order to obtain relevant results, we propose using a GPU device from the high-end segment. An example of a high-end GPU device is the Nvidia Tesla C2070 GPU, which is the flag holder device for Nvidia at the moment of writing this paper.

Regarding parallelism, the Amdahl Law is plotted in Fig. 2, where the x-axis is the number of processors p, and the y-axis is the achieved speedup.

There are three segments that can be noticed on the plot. The segment I represents a relation between the speedup and the number of processors, where by increasing the number of processors. In the second segment, a saturation is achieved, so the speedup stays constant with the increasing the number of processors. The segment III, indicates that increasing of the number of processors, can lead to decreasing of the speedup, which is a consequence of much more communication between the processors and much less computing achieved.

Since for a given GPU device, the number of cores is constant, the plotted curve will depend of the amount of data that is being computed as it is presented in Fig. 3.

The curve 1 is the same curve as plotted in Fig 2. Curves 2 and 3 present the speedup for larger data quantities. Hence, we can conclude that for larger data quantities, the curve achieves saturation much slower.

Therefore, the network simulator parallel module, should scale well over different sizes of networks, in such a way that the simulation scenarios of interest are in the linear segment I, and possibly, if unavoidable in the saturation segment II.

The parallel module should achieve maximal speedup of at

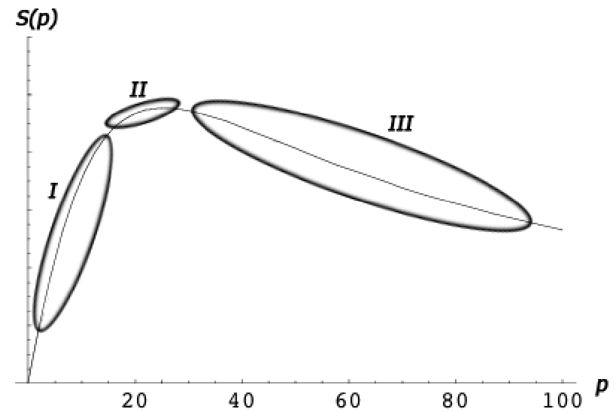

Fig. 2. Parallel speedup.

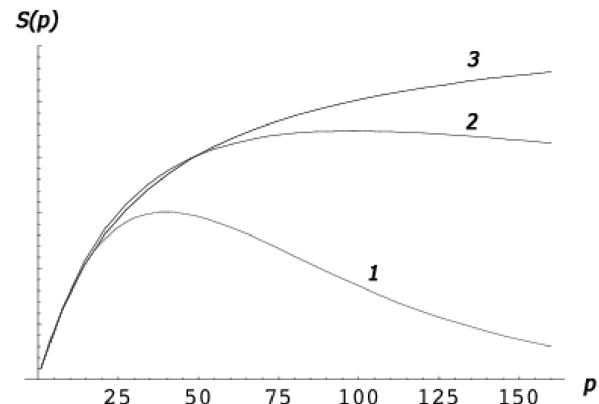

Fig. 3. Parallel speedups for different data amounts.

least x25 on a high-end TESLA C2070 GPU for the overall execution of the network simulator. This is a reasonable performance increase that is consistent with many real-life applications ported to the GPU platform, thus providing another example of achieved acceleration by utilizing the computational power of modern programmable GPU devices.

VI. CONCLUSION

Specific modules of the network simulators demand high computational resources. Therefore, we propose a parallel module for the network simulator in order to utilize the computational performance of GPU devices. Usually the network simulator algorithms run in single precision, so the GPU devices are suitable, although the fact that the GPUs support double precision which is still significantly slower.

In our future work, we intend to develop an implementation of a parallel module for one of the few most widely used network simulators. Also, we would like to evaluate how the GPU implementation of the network simulator extension can perform in specific case network topologies. In addition, we would like to search for the best suitable data structures that can provide further optimization. Beside the stand alone machine setup, we would like to test our parallel module on a multi-GPU setup. Additionally we would like to combine MPI and OpenCL, in order to investigate how parallel module will perform on a cluster of computers, where each computer has a multi-GPU setup.

REFERENCES

[1] Weingartner, E., vom Lehn, H., Wehrle, K., "A Performance Comparison of Recent Network Simulators" in Conf. Rec. 2009. ICC '09. IEEE Int. Conf. Communications, pp. 1-5.

[2] Harris, M.J., “General Purpose Computation on GPUs”, retrieved June 2011 from http://www.gpgpu.org/.

[3] NVIDIA CUDA, retrieved February 2010 from http://developer.nvidia.com/object/cuda.html/.

[4] The OpenCL Specification, Version 1.0, document Revision 43, 2009, retrieved February 2010 from http://www.khronos.org/opencl/.

[5] Zeng, X., Bagrodia, R., Gerla, M., “GloMoSim: A Library forParallel Simulation of Large-Scale Wireless Networks”, in Proc.12th Workshop on Parallel and Distributed Simulation, Banff, Alta.Canada, 1998, p. 154-161.

[6] Parallel Simulation Environment for Complex Systems (PARSEC), retrieved June 2010 from http://pcl.cs.ucla.edu/projects/parsec/.

[7] Cowie, J.H., Nicol D.M., and Ogielski A.T., “Modeling the GlobalInternet”, Computing in Science and Engineering, 1999.

[8] NS-2 Simulator, retrieved June 2010 from: http://nsnam.isi.edu/nsnam/index.php.

[9] Di Caro, G. A., “Analysis of simulation environments for mobile adhoc networks”, Technical Report No. IDSIA-24-03, IDSIA / USISUPSI,BISON project, Switzerland, 2003.

[10] Riley, G., Fujimoto, R.M., Ammar, M., “A Generic Framework for Parallelization of Network Simulations”, in Proc. 7th Int.Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 1999, p. 128-135.

[11] Riley, G.F., “The Georgia Tech Network Simulator”, in Proc. of the Workshop on Models, Methods, and Tools for Reproducible Network Research (MoMe Tools), 2003.

[12] Varga, A., “The OMNeT++ discrete event simulation system”, Proc. of the European Simulation Multiconference (ESM '2001), Prague, Czech Republic, 2001.

[13] Sekercioglu, Y. A., Varga, A., Egan, G. K., “Parallel Simulation Made Easy With Omnet++”, in Proc. of the European Simulation Symposium (ESS2003), Oct. 2003, Delft, The Netherlands.

[14] Grama, A., Gupta, A., Karypis, G., Kumar, V., Introduction to Parallel Computing, 2nd Edition, Addison-Wesley, Reading, MA, 2003.