BRAC UNIVERSITY

PARALLEL COMPUTING USING GPU FOR EFFICIENT TRAFFIC SIMULATION

In partial fulfillment of the requirements of the Degree of BACHELOR OF COMPUTER SCIENCE in BRAC UNIVERSITY

Student's name: SADAT SAKIF AHMED (12241010)

Advisor: PROFESSOR MOHAMMAD ZAHIDUR RAHMAN, Ph.D

Dhaka, Bangladesh

2013

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Parallel Computing can be made possible using the multiple cores of the Graphics Processing Unit (GPU) thanks to the modern programmable GPU models. This allows the use of parallel computing techniques to improve upon the computation time of large scale traffic simulations. This paper proposes the use of a multi-processor algorithm for creating efficient traffic simulation software.

The method in consideration achieves this by separating the r oad network into regions which are individually computed as a threaded block inside the GPU and merged together using the Central Processing Unit to provide the final data of the simulation. A significant improvement in the computation time is observed when the proposed parallelization techniques are applied to the simulator.

ABBREVIATIONS

CUDA - Compute Unified Device Architecture

GPU – Graphics Processing Unit

CPU – Central Processing Unit

SRIPS – Short Range Interactive Particle System

PC – Personal Computer

CHAPTER 1

INTRODUCTION

This chapter introduces the brief description of the research's background and rationale; next, research problems are specified, next the purpose of the research is addressed followed by the tools used and the reason behind the utilization of those tools over other similar tools.

1.1 Background of Research

Computerized traffic simulation is a very cost-effective solution to urban road traffic planning. This is a very commonly adopted method of planning out road networks in any urban settlement. Simulations such as these often deal with a massive amount of data which requires the use of multi-core architecture as a backbone for real-time or accelerated simulations. So far most of these simulators utilize distributed computing solutions in order to achieve the desired results. The solution is effective but not cost efficient. This paper presents a methodology to use the massively parallelization capabilities of modern programmable GPU in order to achieve this effect without the need for a large computer network.

1.2 Purpose of Research

The purpose of this paper is to propose a process which enables scalable parallel simulations to become possible without the need for massive numbers of units in a distributing computing environment and as such, the process described in this paper is a scalable mechanism which allows the performance to improve based on the number of cores available. In essence, the performance boost is achieved by modeling the vehicles as particles

which have a certain degree of autonomy and assigning individual cores to the particles. This effectively speeds up the execution time by a factor of the cores involved. This mechanism is very similar to a domain-distributed computing, but is utilized with the cores available in the Graphics Processing Unit (GPU). With the advent of the new GPU models, it is very affordable to obtain a GPU with an average of 300-400 cores which is the equivalent of using several hundred processors in a distributed computing network. On an advantageous side, the GPU cores are specifically designed for efficient floating point calculation which aids in such simulations.

The simulation allows for real-time adjustments in the constraints and implements a simple machine learning mechanism in order to simulate human behavior among the vehicles. The primary bottleneck in this simulation is the road linkage calculation as it is then that the GPU cores need to wait on the CPU for cached data in order to compute the congestion in the road network.

1.3 Tools Utilized

This simulation software utilizes Nvidia Compute Unified Device Architecture (CUDA) GPU Computing Platform in order to achieve the communication with the graphics device. The CUDA platform is a proprietary platform of Nvidia Corporations and is only supported by Nvidia Graphics Processors. The other alternative to this platform is the OpenCL (Open Computing Library) which is open-source and free to use and is supported by most every graphics processors with programmable pipelines. However, the CUDA platform provides some more benefits such as better support and more efficient communication between supported devices compared to OpenCL as well as better documentation and more efficient libraries.

The simulation program used for testing the proposed method has been coded entirely from scratch using the CUDA platform and Microsoft Visual C++ and has been benchmarked using CUDA Platform's benchmarking software.

This method improves upon the time factor of the simulation significantly with respect to the CPU only implementation and allows for distributed computing using multiple CPU as well as multiple GPU giving this a very large scale implementation possibility.

The simulation takes in account the behavior of road traffic at specific time of the day and also takes in the preference of the general populace to traverse any specific road at the expense of increased congestion as well as traffic signal delays and allows the user to hold off traffic at certain points of the road network in order to assess the impact of the blockage on the rest of the network.

CHAPTER 2

BACKGROUND

This chapter provides some background information regarding the GPU and CUDA and how the two are interlinked. This chapter begins with an idea of how GPU programming works followed by some details on how CUDA works specifically and concludes with a mention of a major issue in CUDA.

2.1 GPU Programming

GPUs are massively parallel multithreaded devices capable of executing a large number of active threads concurrently. A GPU consists of multiple streaming multiprocessors each containing a multiple scalar processor core. For example, NVIDIA's GeForce GT X 560 contains 336 CUDA Cores which can handle up to 36,000 active threads in parallel.

In addition, the GPU has several types of memory, most notably the main device memory (global memory) and the on-chip memory shared between all threads in a block. The CUDA language library facilitates the use of GPUs for general purpose programming by providing a minimal set of extensions to the C programming language. From the perspective of the CUDA programmer, the GPU is treated as a coprocessor to the main CPU. A function that executes on the GPU, called a kernel, consists of multiple threads each executing the same code, but on different data, in a manner referred to as "single instruction, multiple data" (SIMD). Further, threads can be grouped into thread blocks, an abstraction that takes advantage of the fact that threads executing on the same multiprocessor can share data via the on-chip shared memory, allowing a limited degree of cooperation between threads in the

same block. Finally, since GPU architecture is inherently different from a traditional CPU, code 5 optimization for the GPU involves different approaches. [4, 5]

2.2 Description of CUDA

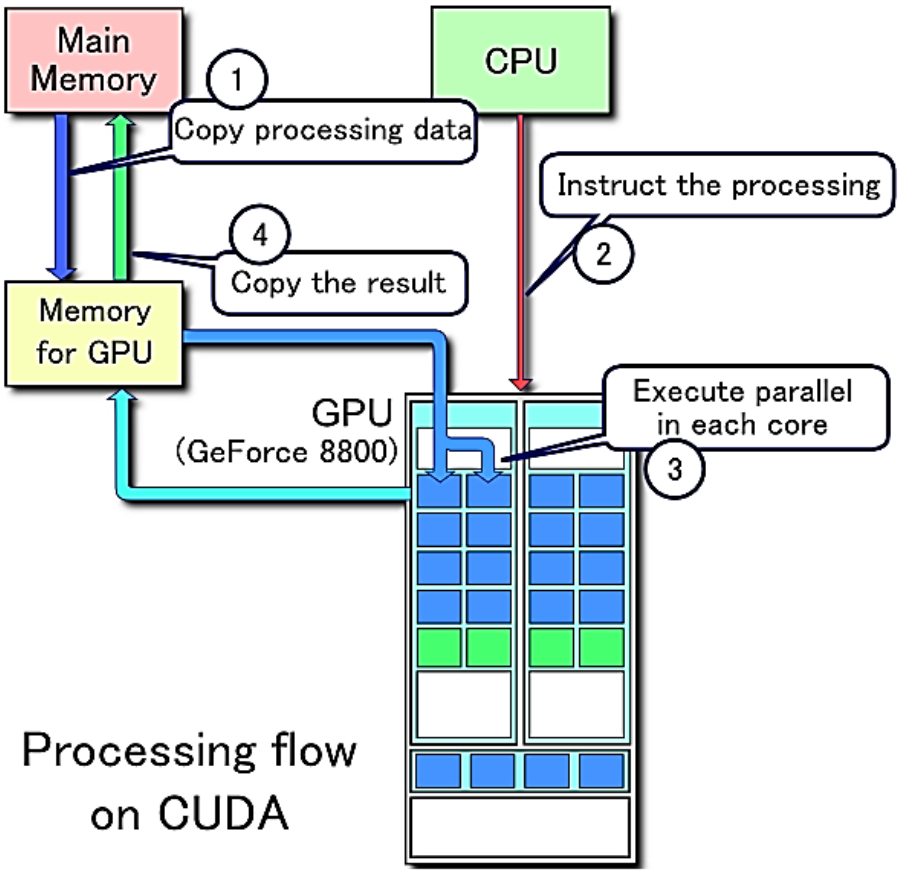

Figure 1: Processing Flow on CUDA

The CUDA platform models the processes to run as thread blocks which together form an emulation of a processor. These thread blocks can be as big as necessary to fit the entire process in them and the number of process blocks available depends on the number of cores available. This is a great advantage over single thread based parallelization modules since this provides the GPU with a means of accommodating larger processes than a single GPU core can handle. This mechanism requires all data for every instance of the process to

be copied to the GPU memory which is accessed via the process blocks for that particular instance. [4, 5]

The above mentioned approach of CUDA to parallelization in GPU has one major disadvantage which is the lack of inter-communication between the processes. In order to provide inter-communication, the GPU processes need to upload the data into the main memory which the CPU accesses and performs the necessary steps to merge them.

2.3 Issues with GPU Precision

As noted in the CUDA Programming Guide [4, 5], CUDA implements single precision floating-point operations e.g., division and square root operations, in ways that are not IEEE-compliant. Their error, in ULP(Units in the Last Place) is nonzero. While addition and multiplication are IEEE-compliant, combinations of multiplication and addition are treated in a nonstandard way that leads to incorrect rounding and truncation.

CHAPTER 3

SIMULATION MODELING

This chapter describes how the objects in the simulation have been modeled for efficient parallelization; beginning with the description of how the vehicle has been modeled and following through with the basic machine learning implemented in the vehicle. Finally, the model of the road network is described.

3.1 Vehicle Modeling

The vehicle is modeled as a particle moving through a rectangular tube which only allows multiple particles to move side-by-side. This approach has been influenced by a method proposed by Knowles, P for GPU based particle simulation using spatial Short Range Interacting Particle System (SRIPS).[2]

```
for i = 0 to N − 1 do
    move(particle[i]);
    for all j in neighbor(particle[i]) do
        check collision between particle i and j;
    end for
end for
```

Algorithm **1**: SRIPS using a spatial data structure

The spatial SRIPS algorithm is an Θ(n) algorithm since the inner loop is always fixed to a constant number of other particles. In the case of this simulation this value is at most 4 as

there is no possibility of any other particles on any other side. The particles are clumped together based on their position and the surrounding particles are detected from a list of neighboring particles which may or may not change on every road intersection. The vehicle shows general commute tendency depending on the time of day and attempts to show the traffic movement on a macro scale. The collision is maintained through the use of a list of vehicles in the vicinity of every vehicle and the data of the surrounding vehicles is passed in to the thread block along with the data of each vehicle. This leads to the data of the surrounding vehicles to be computed multiple times which is then merged using a Gaussian distribution in the CPU for averaging out the positional data in order to provide an approximation of the resulting position of the particle. This method does not provide an very accurate result for the position of the vehicles, but the margin of error in the position is rather minimal and does not affect the result in such a level as to create a large error. The error is calculated thus

```
for i = 0 to N − 1 do
    c = count(instances[i])
    for all pos in instances[i] do
        difference += maxPos(instances[i]) – minPos(instances[i]);
    end for
    difference /= c;
end for
```

Algorithm 2: Calculating margin of error in position recalculation

Using this mechanism is the most effective measure of obtaining parallelization of the algorithm and provides a workable margin of error which ranges from 15% – 25% in the overall simulation.

3.2 Vehicle Learning Mechanism

The vehicle model concept makes use of the modern computing capabilities in order to provide an intelligent, human-like behavior among the vehicles and attempts to make the simulation more accurate. In order to achieve this, the model implements a basic set of rules to define a fuzzy logic which defines the behavior of the driver. In this case the driver's behavior is simulated based on a set of values which define the perception and decision of the driver. The behavior simulation is applied to nearby vehicles with a diffusion gradient in order to simulate crowd mentality and reduce the computational pressure instead of having to compute the crowd mentality by simulating through each and every vehicle.[1]

3.3 Road Network Modeling

The road network is modeled as a simple n-ary graph with the nodes being the crossroads and the edges the routes. Each individual edge contains data regarding the current density of vehicles in the route and maintains the general driving preference of the population through that route. The driving preference is utilized in order to simulate the behavior of general population as we see in nature that even though there are multiple routes which lead to the same destination, the preference of the routes depends on the general perception of the road being the quicker path to the destination as decided through the greedy nature of humans. This behavior is the primary reason for uneven traffic congestion in roads. This

behavior is modified in the simulation depending on the amount of congestion that drivers find in the given road compared to other roads which share the same destination.

Computing the road congestion is the biggest performance hit as the calculation is dependent on the number of particles entering and leaving the edge. This essentially requires the data in the GPU to be downloaded into the main memory and then accessed through the CPU in order to calculate the number of vehicles in the respective edge. This is performed using a spatial sorting algorithm which works as follows

```
for e = 0 to E − 1 do
   for all particle in edge[e] do
      removeFrom(currentEdge, particle);
      addTo(destEdge, particle);
   end for
end for
```

Algorithm 3: Restructuring Edge Lists

Maintaining multiple lists for individual edges allows the calculation in a different edge to continue in the GPU while rearrangement is being performed in an edge. This is essential to maintaining the parallelization of the traffic congestion algorithm. This approach also enables scalability as multiple instances of the simulation can work on simulating different regions of the road network on different CPU and GPU without being dependent on the entire data.

There is one difficulty which requires a more elaborate approach to solve and that is when a vehicle is moving from one edge in one instance of the program into another edge in a

different instance of the simulation. This problem has been solved by maintaining a global list of edges and the instance id of the respective instance of the simulation where the edge lies. This requires a bit more memory, but that is a small price to pay when compared with the alternative of having to copy entire data from all instances and searching through all of the data in order to reach the desired edge and then having to recopy the modified data into all instances for updating. With the previous approach all that is necessary is to look through the list of edges and update the necessary edge in the associated instance of the simulation in order to maintain the road network. This approach saves valuable time as each lookup is only Θ(n) whereas the alternative would have been Θ(n*i*e + n^2). The tendency of any given vehicle to traverse any specific edge in the road network depends on the threshold of the preference which is adjusted based on the number of times the vehicle traversed the route over its congestion threshold. The congestion threshold is a measure of how much congestion is tolerable by the vehicle before its preference for that particular road starts to drop. This shows a general tendency for traffic to become distributed evenly over time.

There is inherently no complete path finding algorithm used in this simulation as the mechanism of path finding algorithms does not allow for parallelization. However, this problem has been approximated rather splendidly using greedy path following mechanisms which tweak the preference threshold in order to make the vehicle traverse the required path.

CHAPTER 4

FINDINGS

This chapter describes the findings of the implementation of the research and provides reasoning behind the findings. Firstly, a brief description of the implementation platform followed by a detailed performance analysis.

4.1 Implementation Platform

Computer Parts	Specifications
GPU	Nvidia GeForce GTX 560
Dedicated Graphics Memory	1024 MB GDDR5
GPU Memory Interface	256-bit
CUDA Cores	336
GPU Processor Clock	1620MHz
GPU Memory Data Rate	4008 MHz
CPU	Intel Core 2 Duo 2.50GHz
RAM	4.00 GB
Operating System	Windows 8 (X64)

Table 1: *Implementation Device Specification*

The above table shows the specification of the implementation device for the simulation software. The software was run in this one single PC with a maximum road network of 5000 nodes and 1,000,000 vehicles without any noticeable delay when utilizing the parallel computing capabilities of the GPU. When run without using the GPU the maximum node count remained unchanged, however the number of vehicles had to be

reduced to 100,000 to observe the same level of delay in the computation. Below is a graph of Frame Update Time in sec vs. number of vehicles with and without using the GPU.

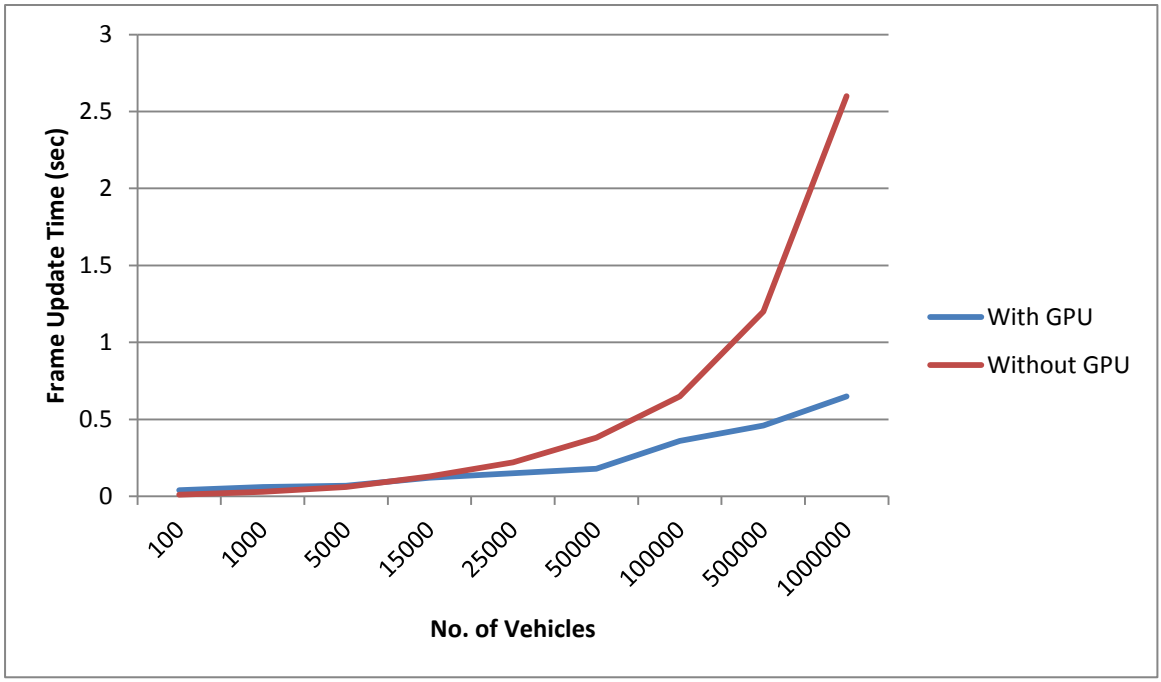

Figure 2: *Comparison of frame update time against number of vehicles*

Note that the computation time increase is rather irregular in the simulation using the GPU as well while the CPU only simulation is following a rather standard $\Theta(n^2)$ curve pattern. This is because the computation time increase in GPU is only observed when all 336 cores are overwhelmed with the number of threads entering and requires multiple iterations to compute all threads. This increase is rather sudden as it only occurs when the number of iterations increases as opposed to the $\Theta(n^2)$ of the CPU.

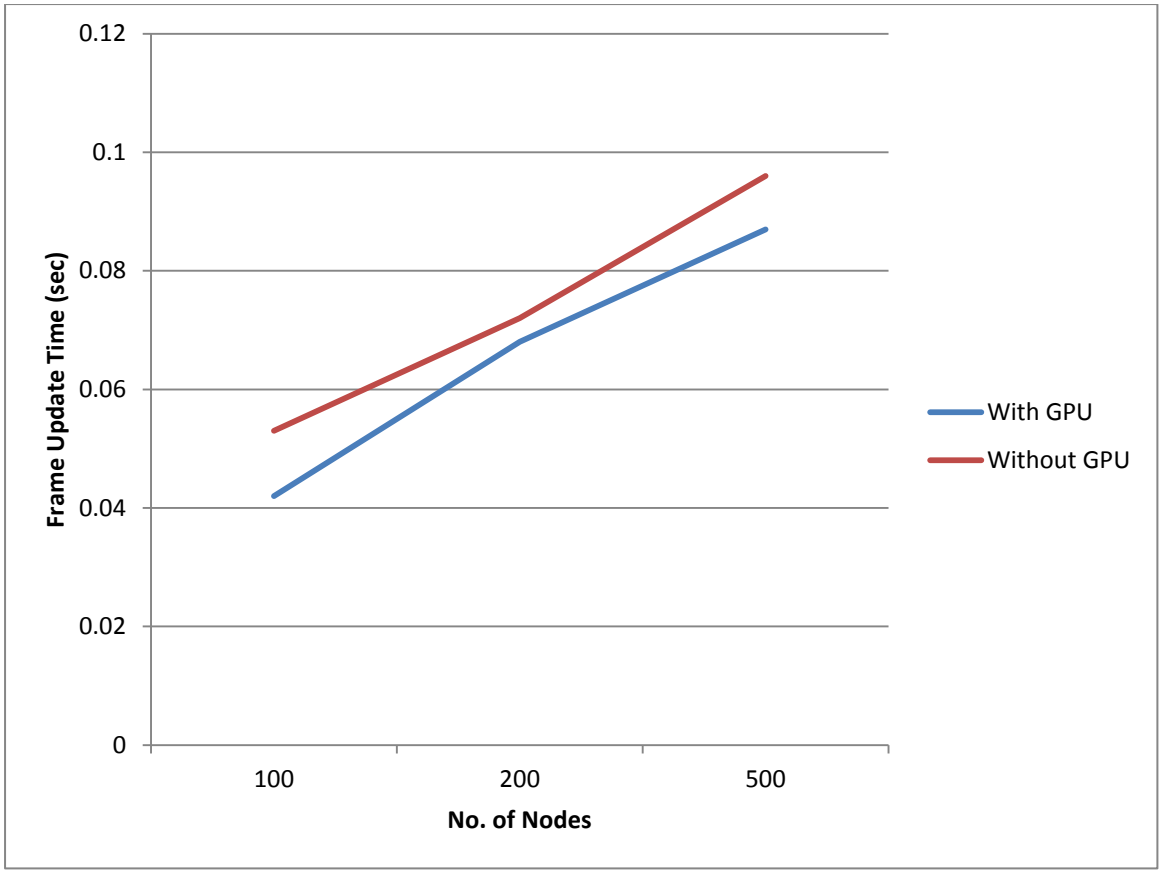

Figure 3: *Comparison of frame update time against number of Nodes*

In this performance analysis, the number of vehicles were left at a constant of 5,000 while the number of nodes were increased, the change in performance of both the simulations were observed to be approximately O(n) since the computation at the nodes are bounded by the number of CPU as there is no scope of parallelization in this context. So it is clearly evident that unless parallelism is observed the performance gain from utilizing the GPU is minimal.

4.2 Detailed Performance Analysis

The performance of the simulation depends on two major factors. The number of vehicles and the number of road network nodes. The number of vehicles places stress on the GPU and even though the vehicle movement and collision algorithm is $\Theta(n/k)$ where k is the number of cores involved, the number of nodes has no impact on the GPU instead the number of nodes will place the load entirely on the CPU and the algorithm for finding out the road density is

Θ(n*m) where m is the number of nodes involved. This creates a bottleneck when the number of nodes is increased in a single CPU. Since the approach specified in this paper enables distributed computing, it is necessary to increase the number of CPU in the network in order to improve the performance in the scenario of a large number of nodes. However, since the number of vehicles does not place any load on the CPU and as it has been observed from Figure 1, the increase in delay in case of GPU aided computing is much lower than that of the standard CPU based computing, it has been observed that utilizing the parallelization capability of the GPU does indeed improve the performance of the entire simulation process without the need of too many computers in the distributed network.

CHAPTER 5

CONCLUSION AND FUTURE WORK

This chapter reflects upon the findings of the research and provides an insight on what can be done to improve upon this research in the future.

5.1 Conclusion

In this thesis it has been shown that using parallelization with the aid of GPU is in fact a very practical solution in battling the need for high performance computing with lower cost solutions, to this end, it has been shown in the performance analysis that only increasing the number of cores in a GPU improves the computation time of distinct set of elements while doing nothing for the performance of elements which are dependent on other data for continuing the process. This shows that even though computation time can be improved with the use of GPU we still cannot eliminate the need for distributed computing when parallelization is of the utmost importance.

This thesis also shows how a traffic simulator can be made to work in a distributed computing environment while also utilizing the parallelization capabilities of the GPU.

5.2 Future Work

The simulator used in this research is more of a make-shift benchmarking tool than an actual simulator. Since the purpose of this paper was to prove that using the parallelization capabilities of the GPU enhances performance of processing large distinct instances of data, it was beyond the scope of this thesis to create a fully functional traffic simulator. The framework developed during this thesis can be further enhanced with added modules in order

to develop this simple benchmarking tool into a practical traffic simulation tool which has the potential to be used as a tool for urban development planning. So, in light of such possibility of improvement of the current software, future work involves improving the current simulator with more functionality and more parameters in order to enable it to perform much more functionalities. The simulator used in this thesis is also prone to error thanks to the number of approximation algorithms used, it is also necessary to improve upon the margin of error and provide more accurate results, so future work includes the development of the simulator as well as optimizing the approximation algorithms used in order to come up with a fully functional road traffic simulator.

REFERENCES

[1] Talal Al-Shihabi and Ronald R. Mourant. (2001). A framework for modeling human-like driving behaviors for autonomous vehicles in driving simulators. In *Proceedings of the fifth international conference on Autonomous agents* (AGENTS '01). 286-291.

[2] Knowles, P. (2009). GPGPU based particle system simulation. *School of Computer Science and Information Technology RMIT University Melbourne, Australia*, *12*(04), 55-58.

[3] Wikipedia. (2012). CUDA. Retrieved from *http://en.wikipedia.org/wiki/CUDA*

[4] H. Nguyen. GPU Gems 3. 2008.

[5] NVIDIA. NVIDIA CUDA - Programming Language. 2008.