Simulating Soft Tissues using a GPU approach of the *Mass-Spring Model*

Christian Andres Diaz Leon
Virtual Reality Laboratory
EAFIT University
Medellin, Colombia

Steven Eliuk
Advanced Man Machine Interface Laboratory (AMMi)
University of Alberta
Edmonton, Canada

Helmuth Trefftz Gomez
Virtual Reality Laboratory
EAFIT University
Medellin, Colombia

ABSTRACT

The recent advances in the fields such as modeling bio-mechanics of living tissues, haptic technologies, computational capacity, and graphics realism have created conditions necessary in order to develop effective surgical training and learning methods using virtual environments. However, virtual simulators need to meet two requirements in order to be useful in a training environment: good interactivity (real-time FPS) and high realism. The most expensive computational task in a surgical simulator is that of the physical model. The physical model is the component responsible to simulate the deformation of the anatomical structures and the most important factor in order to obtain realism. In this paper we present a novel approach to virtual surgery. The novelty comes in two forms: specifically a highly realistic *mass-spring* model, and a GPU based technique, and analysis, that provides a nearly 80x speedup over serial execution and 20x speedup over CPU based parallel execution.

1 INTRODUCTION

When considering workload and computation for a surgery simulator the largest computational task is that of simulation of the deformation of the anatomical structures. In recent years, the advent of programmable graphical processing unit (GPU), has allowed for the use of the computational power for general purposes programming on the GPU (GPGPU), such as calculating the deformation of anatomical structures in the surgical simulator. Due to the above reasoning, and the high degree of parallelization possible within the calculation of the *mass-spring* model, these methods are a perfect candidate to be implemented on the GPU.

Currently, there are implementations of the *mass-spring* model on the GPU [3]. However, these methods lack the ability to use specialized shared-memory on the GPU, available through CUDA, where a substantial speedup can be obtained. Recently, in [2] was explored the use of CUDA for simulating deformable objects using the mass-spring model. Our approach differs from the proposed in [2] particularly in two respects. The first is the deployment of a hybrid approach that uses shared and coalescence memory in order to increase the speed-up. In the second, we present a comparative study between the CPU-based with the GPU-based aproaches, analyzing variables such as computational error, runtime and speed-up.

2 PARALLEL CPU–BASED APROXIMATION OF THE *Mass-Spring* MODEL

The algorithm used to calculate the deformation applies the same operation to each node of the mesh. This operation consist of the calculation of the F_i (Internal Force of the node) to each node and the updating of the node position using the previous position, F_i is calculated as well as the convergence factor α. Taking this into account, the problem domain can be divided in a data level, where the grain size can be each node. The algorithm can be parallelized making the computation of F_i and x_{i+1} in a parallel way to each node of the mesh.

3 GPU APROXIMATION OF THE *Mass-Spring* MODEL

Next we describe the approaches implemented in the GPU to calculate the deformation using four different memory setups to store the data structure of the mesh.

- *Global Memory Implementation*: The data structure used in the GPU implementation consists on three 1D arrays which are linked by the position of each point in the array. The first array called *Positions* contains the geometric coordinates, the mass and the boundary condition of the each point. The second array called *Neighbors* has the neighbors of each point in the mesh and the third array called *Length Rest* contains the length rest of the each link in the mesh. In this implementation the data structure is stored in the global memory of the GPU. The descomposition of the problem is carried out in a similar way like the parallel algorithm, described in the previous section, that is each thread in GPU computes the new position of a point in the mesh. In this implementation, the use of global memory to store the data structure may limit the performance by this approach due to high latency of reading and writing to global memory on the GPU.

- *Coalescence Memory Implementation*: By performing a simple modification of the data structure described before, it is possible to improve the performance of the algorithm implemented on the GPU. To this end, it is necessary to apply the concept of coalescence memory. Coalesced memory refers to property that the global memory of the GPU has been arranged in a way to allow memory access to the same DRAM page when multiple threads simultaneously access contiguous elements of memory [1]. For that reason, and in order to exploit this property of the global memory, the data structure described before was slightly modified, simply by organizing all information that will be accessed at the same time for each thread in a consecutive way in the memory. This schema ensures that the coordinates x, y and z, the masses, boundary conditions, neighbors of each point and the rest length of each spring are consecutively stored in memory.

- *Shared Memory Implementation*: Other option for improving the performance of the algorithm is to use the shared memory of the GPU, which has writing and reading latency that is less than that of the global memory [1]. The idea of this approach is to copy the positions of points from the global memory to shared memory. Exploiting the characteristics of a neighborhood, and in this way to minimizes the accesses made to the global memory. However, this is only applicable if the information contained in the array is structured, i.e. if the neighbors of a specific point within the array, are also neighbors in the geometry of the mesh. Changes to the data structure are basically focused on how the neighbors of each point are stored. In this case the index is the position of a point in the array of points. In the new data structure each

point has maximum eight neighbors, and to determine if there is a connection with each of these neighbors, values of 0 and 1 are used, where 1 refers to a connection and 0 otherwise. Regarding the changes of the algorithm, each thread of the block reads a point of the mesh and is copied to shared memory, but for the calculation it is necessary to have access to the coordinates of the points around the block some threads of the block must copy these addition all positions.

- *Shared Memory + Coalescence Memory Implementation*: Finally, the last implementation carried out, took advantage of the benefits in terms of performance offered by shared memory and the property of coalescence memory. For this purpose we combined the data structures used in each approach.

4 Experimental Setup and Results

In order to compare the sequential, CPU-based parallel and GPU algorithms, an experimental test was developed. In the test performance variables were measured, such as time execution and speed-up. In the case of CPU-based parallel implementation, several trials were conducted changing the number of threads. The experimental test was carried out in a machine with Intel processor Quad core (2.2GHz), 2 Gb of RAM memory and Nvidia GeForce 8800 GT GPU. Figure 1 presents the results of the execution time obtained for each of the approaches described before versus the number of points that possess each of the meshes evaluated.

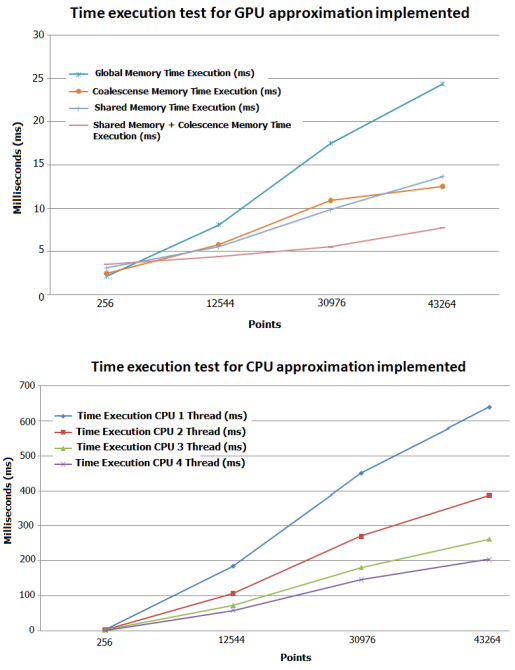

Figure 1: Time execution for each one of meshes evaluated and approaches implemented, GPU and CPU

Aditionally, figure 2 shows the speed-up achieved for each of the approaches and meshes evaluated during the experimental tests.

5 Conclusions

In this paper we explore various GPU implementations of the *mass-spring* model using the CUDA framework. The purpose was to compare the improvements offered by multi–core and GPU technologies and to explore to what extent they fulfill the current requirements of surgical simulators. In this way the results shown in

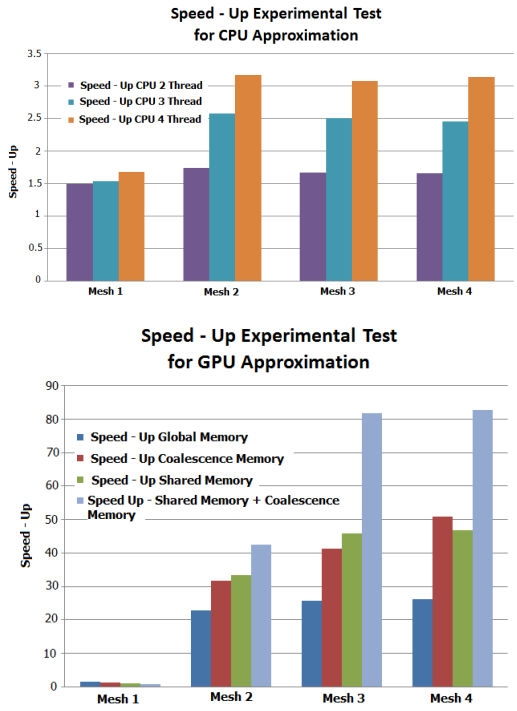

Figure 2: Speed-up for each one of meshes evaluated and approaches implemented, GPU and CPU

Figure 1, allow one to conclude that the time execution of the CPU-based parallel algorithm decreases approximately owns an equal relation to the sequential time divided by the number of threads, as it is established by the theoretical predictions. However, the number of CPUs required to reach the performance level of one GPU is unrealistic considering the ratio of flops/dollar. Likewise, it is clear that the approach made in the GPU requires a shortest time execution with respect to time spent by the implementations on the CPU (sequential and parallel). However, for very small meshes, the performance is similar to the one achieved by the approaches implemented in the CPU because in this case the kernel does not utilize the parallel powers of the GPU fully. This same result is visible by analyzing Figure 2. In this case, the higher speed-up was obtained with the implementation on the GPU that combines the use of shared memory and the property of coalescence memory. However, the methods that use shared memory, need a structured mesh to be implemented, this limits the implementation of such methods to only those with a structured mesh. Finally, regarding the approaches explored, in order to fulfill the performance requirements of a surgical simulator, only those implemented on the GPU, especially the approaches that make use of shared memory and property of memory coalescence of global memory.

References

[1] W. Hwu, C. Rodrigues, S. Ryoo and J. Stratton. Compute Unified Device Architecture Application Suitability. *Computing in Science and Engineering*, Volume 11, Issue 3, pp. 16 26, 2009.

[2] A. Rasmusson, J. Mosegaard and T. Sangild. Exploring Parallel Algorithms for Volumetric Mass-Spring-Damper Models in CUDA. *Lecture Notes in Computer Science*, Volume 5104, pp. 49-58, 2008.

[3] J. Georgii and R. Westermann. Mass-Spring Systems on the GPU. *Simulation Modelling Practice and Theory*, Volume 13, Issue 8, pp. 693 702, 2005.