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

Pierre Boulanger[§]
Advanced Man Machine Interface Laboratory (AMMi)
University of Alberta
Edmonton, Canada

Abstract

The recent advances in the fields such as modelling bio-mechanics of living tissues, haptic technologies, computational capacity, and virtual environments have created conditions necessary in order to develop effective surgical training and learning methods. Simulation environments for surgical training have no limitations on the number of times a procedure can be executed, and most importantly have no risk to patients. Moreover, these simulations allow for quantitative evaluation of a surgeons performance, leading to the ability to create performance standards in order to determine a surgeons current surgical expertise.

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.

Index Terms: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics Processors; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Physically based modeling

[*]e-mail: cdiazleo@eafit.edu.co
[†]e-mail: sneliuk@ualberta.ca
[‡]e-mail: htrefftz@eafit.edu.co
[§]e-mail: pierreb@ualberta.ca

1 Introduction

The current model of learning and training in laparoscopic surgery is based on new surgeons observing and gradually participating in laparoscopic procedures, taking active roles depending on their experience [1]. Unlike open surgery, Minimally Invasive Surgery (MIS) requires the development of different skills due to the lack of haptic and indirect visual feedback [2]. This makes it necessary to develop new training methods for MIS. In this context, virtual reality has arisen in the last decade as a tool for the development of surgical training environments.

Nevertheless, these systems must face a trade-off between the realism and the real-time performance of the surgical simulator. When considering workload and computation for a surgery simulator the largest computational task is that of simulation of the deformation of the anatomical structures due to collision with surgical instruments. There has been several techniques proposed to improve the performance in terms of simulation of the deformation phase. Some authors have chosen to implement simplistic physical models [3], others have relied on expensive configurations of multi-core hardware and complex physical models [4]. However, these approaches have drawbacks, such as high computational cost, slow interactivity, and low realism with respect to the deformation phenomenon [5].

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 [6]. Similarly, the recent release of the CUDA framework [7], from Nvidia, has provided more accessible programming, and implementation, of GPGPU-based calculations on the GPU. 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, fulfilling the requirements of a surgical simulator.

Currently, there are some implementations of the *mass-spring* model on the GPU using OpenGL superbuffers, using different types of data structures [8] and applying different approaches of the algorithm to solve the model [9]. 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 [14] was explored the use of CUDA for simulating deformable objects using the mass-spring model. Basically, in this paper was considered two types of data structures, one is called implicit and another is called explicit. Explicit data structure is very similar to what we propose in this article. Moreover, the implicit data structure uses a three-dimensional grid to store the data of neighborliness between the particles to facilitate the use of shared memory. Our approach differs from the proposed in [14] 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.

In general, in this paper serial and parallel CPU–based approach were implemented to compute the real-time deformation of the anatomical structures with an acceptable resolution using a *mass-spring* model. Then, the CUDA framework was used for the implementation of the algorithm on the GPU, analyzing the obtained speed-up depending on the application of different types of available memory. And finally, an experimental test was carried out to compare the performance achieved by each of the proposed algorithms, serial, CPU based parallel, and GPU approaches.

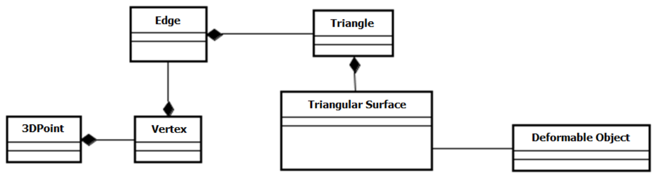

Figure 1: Data structure of the deformable objects in the Simulator.

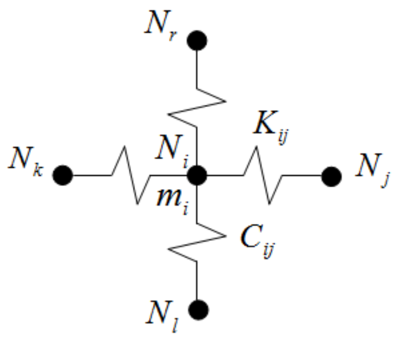

Figure 2: Representation of the *Mass-Spring* Model.

2 Materials and Methods

2.1 Data Structure of the Physical Model in the Surgical Simulator

The surgical simulator developed in the Virtual Reality Laboratory of EAFIT University [10] has a specific data structure to load the tridimensional models and apply the calculation of the deformation. Figure 1 shows the data structure of the deformable object representation in the surgical simulator.

In general, a deformable object is composed by a set of points connected together to form triangles and edges, and these are grouped into a single geometrical surface of the object. A vertex is composed of two points, a point that contains the information of the previous positions (previous frame) and information that contains the current position (current frame), these are used to calculate the deformation. The vertices are connected by an edge, and are arranged such that three vertices joined by three edges form a triangle, so that two adjacent triangles share an edge and two vertices.

2.2 Overview of the *Mass-Spring* Model and Serial Approach

In order to determine the interactions among objects in a virtual surgery environment, an algorithm that takes into account deformable objects is necessary. To accomplish this objective we implemented a physical model based on the *mass-spring* method. The next explanation is based on the work published by Brown et al. [11]. This model represents the deformable object by a tridimensional mesh M, of n nodes $N_i (i = 1,...,n)$ connected by links L_{ij} such as i, j, $\in (1,...,n)$ and i is different to j. Each node is mapped to a specific point of the object, so that the displacements of the nodes characterize the deformation of the object. The nodes and links on the object′s surface can be triangles or tetrahedra, in our case, the surface is triangulated. The physical model is represented in Figure 2.

The mechanical properties (viscoelasticity) of the object are described by constants stored in the nodes and links of M. A mass m_i

and a damping coefficient C_i are associated with each node N_i, and a stiffness K_{ij} is associated with each link L_{ij}. The internal force between two nodes N_i and N_j is shown in equation 1.

$$\vec{F_{ij}} = -K_{ij}(\Delta_{ij}\vec{u_{ij}}) \tag{1}$$

where,

$$\Delta_{ij} = l_{ij} - rl_{ij} \tag{2}$$

Δ_{ij} is the current length of the link minus its resting link, and $\vec{u_{ij}}$ is the unit vector pointing from N_i toward N_j. The stiffness K_{ij} may be constant or a function of Δ_{ij}. In either case, F_{ij} is a function of the vectors coordinates x_i and x_j of N_i and N_j. This representation can describe objects that are nonlinear, non-homogeneous, and anisotropic. At any time t, the deformation and motion of M is described by a system of n differential equations, each expressing the motion of a node N_i:

$$m_i\vec{a_i} + c_i\vec{v_i} + \sum_{j\in\sigma(i)} \vec{F_{ij}}(\vec{x_i}, \vec{x_j}) = m_i g + \vec{F}_i^{ext} \tag{3}$$

where $\vec{x_i}$ is the coordinate vector of N_i; $\vec{v_i}$ and $\vec{a_i}$ are its velocity and acceleration vectors, respectively, $m_i g$ is the gravitational force, and $\vec{F}_i^{ext}$ is the total external force applied to N_i. $\sigma(i)$ denotes the set of indexes of the nodes adjacent to N_i in M.

It is possible to reduce the complexity of Equation 3 considering the approach proposed on [11]. In this approach we assume that the velocity of the nodes is small enough so that the mesh achieves static equilibrium at each instant, that is, every frame of the simulation. This is a reasonable assumption for soft objects with relatively high damping parameters, which is the case for most human tissues.

We can then neglect the dynamic inertial and damping forces. In this way the shape of the mesh M is described by the following expression:

$$\sum_{j\in\sigma_{ij}} F_{ij}(x_i, x_j) = m_i g + F_i^{ext} \tag{4}$$

This system can be solved using an iterative solver method like Runge-Kutta or Newton, however we implemented a penalty solver algorithm most commonly used in surgical simulation, alike to Newton's method. The following pseudocode describes the implemented method.

Algorithm to solve the cuasi-static model

```
Get the current position of each node
    While $Cont < \delta$ then
        For each $i \in I$
        $f_i \leftarrow \sum_{j\in\sigma_{ij}} F_{ij} - m_i g$
        $x_i \leftarrow x_i + \alpha f_i$
        $Cont = Cont + 1$
        End For
    End While
```

Cont is the current iteration of the solution system, δ is the minimal iteration number to guarantee the convergence of the method, in order to reach the equilibrium state. I denotes the set nodes of the mesh and α is a convergence factor of the solution method.

Analyzing the complexity of the physical model algorithm we can define that the convergence of the method and the real-time performance depends on the number of nodes composing the mesh and the number of links to each node. Generally a node is linked to six another nodes, and the performace of the algorithm directly depends of the number of nodes. This means that if the mesh has a large number of nodes, the real time performance is hard to guarantee.

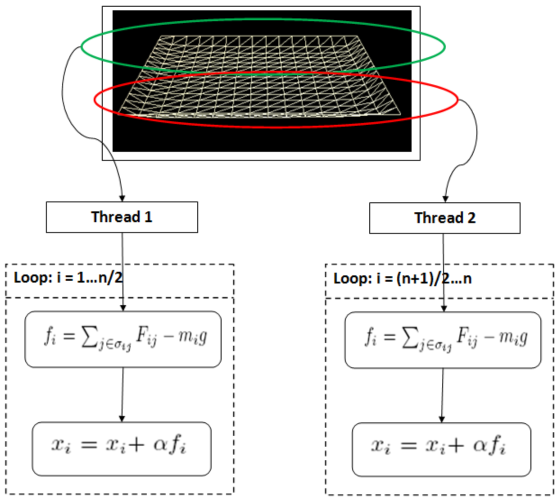

Figure 3: Graphical representation of the parallel algorithm implemented. In this case there are two threads, therefore the domain problem (nodes) is divided in two subsets.

2.2.1 Parallel CPU–based Approximation of the *Mass-Spring* Model

In recent years, multi–core technologies have become very common and inexpensive. This situation has motivated the parallelization of the costly algorithms that require real-time performance, such as a physical models in the surgical simulation applications. For this reason, in this section we explore the parallelization of the algorithm described using a mesh with a large number of nodes.

The first task, when parallelizing a sequential algorithm, is to define the decomposition of the problem. The decomposition of the problem can be made in different levels, depending of the specific application. In general there is data decomposition, task decomposition, data and task decomposition (pipeline) and finally mixed solutions.

In our case, the algorithm presented in the last section 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. Figure 3, shows the graphical representation of the parallelization applied to the algorithm. Communication is needed to share information between nodes, because the algorithm is based on the summation of forces.

In order to implement the parallel algorithm we used the *threading building blocks* (*TBB*) [12] software library. This library offers the opportunity to avoid thread management. This will result in code that is easier to create, easier to maintain, and more elegant. Specifically, in the project it was necessary to parallelize the *for* loop in order to sweep the nodes of the mesh and compute the next

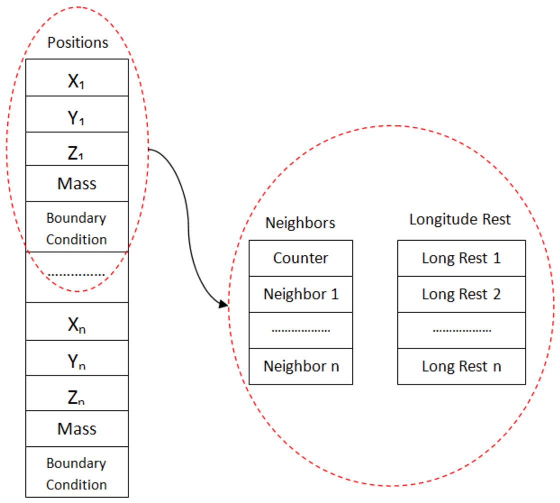

Figure 4: Data structure used to store the information of the mesh which deformation will be calculated on the GPU.

position of each node. To accomplish this goal we used the template *parallelfor* included in *TBB*. This template allows for the parallel execution of a fixed number of independent loop iterations.

2.3 GPU Approximation 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.

2.3.1 Global Memory Implementation

In order to implement a GPU approach of the *mass-spring* model it is necessary to build a special data structure to copy the information of the mesh from the host memory to the device memory on the GPU. The data structure consists on three 1D arrays which are linked by the position of each point in the array. The first array called *Positions* contains the x, y and z 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. Figure 4, shows the data structure described above.

Once the data structure, in which information of the mesh will be stored, is defined, one must setup the decomposition of the problem to parallelize the computation of the deformation on the GPU. As with the parallel algorithm, described in the previous section, the algorithm implemented on the GPU will make a spatial decomposition of the problem, so that each thread in GPU computes the new position of a point in the mesh. In Figure 5, one can observe the spatial decomposition realized to implement the algorithm in the GPU. In this example the mesh is composed of 12 points. In order for each thread to process a point, four blocks with four threads is used. Consequently, the number of blocks used depends on the number of points in the mesh.

For meshes with a large number of points, this approach can offer a great performance improvement, due to the high parallelization achieved in the calculation. However, 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, still far faster than CPU/Hot memory.

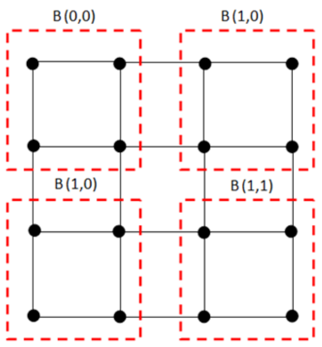

Figure 5: Example of the spatial decomposition of the deformation calculation of the GPU. In this example a grid 2 by 2 and 2 by 2 blocks were used.

To solve this problem, three additional approaches were considered: The first one makes use of the property of coalescence memory that holds the global memory, the second uses shared memory on the GPU to store the information of the processed points in each block. This minimizes the number of accesses made to global memory in order to read the position of the neighbors, an operation that is required to update each point of the mesh and the third simply combines the use of shared memory and the property of coalescence memory. These approaches are described next in the following section.

2.3.2 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 [13]. In this way if it is possible to exploit this property of the global memory, it may somewhat reduce the impact produced by the long latency of reading and writing of the global memory.

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. Figure 6 shows the new data structure.

The only difference between the data structure shown in figure 4 and in Figure 6, is the order in which data is stored. The schema presented in the Figure 6 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 and in this way to favor the coalescence memory of the global memory.

2.3.3 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 [13]. The idea of this approach is to copy the positions of points from the

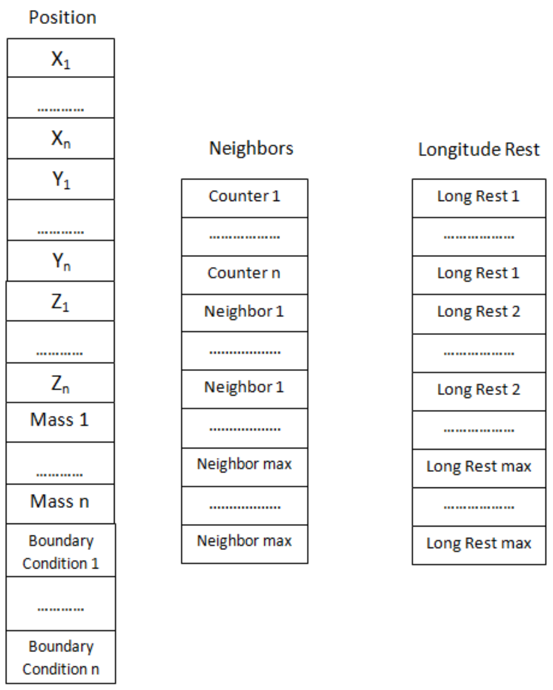

Figure 6: Data structure implemented to exploit the property of coalescence memory of the global memory of the GPU.

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.

It was then necessary to make some modifications to the algorithm and the data structure. Similarly it was assumed that the maximum number of neighbors of any point on the mesh is 8.

Changes to the data structure are basically focused on how the neighbors of each point are stored. In the data structure implemented initially, for each point of the mesh, the number of neighbors and the indexes of each neighbor 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. The reason to propose this new configuration of the data structure is to easily map the position of the points stored in the global memory to the shared memory of each block. Figure 7 presents an example showing the basic difference between the data structures used in the global memory implementation and the shared memory implementation.

Regarding the changes of the algorithm for the computation of deformation, an extra piece of code was added to determine the block and thread id. This was needed in order to determine what memory was needed to be copied from global to shared device memory. In general each thread of the block reads a point of the mesh and is copied to shared memory (squares inside red square in Figure 8), but for the calculation it is necessary to have access to the coordinates of the points around the block (squares outside red square in Figure 8), some threads of the block must copy these additional positions.

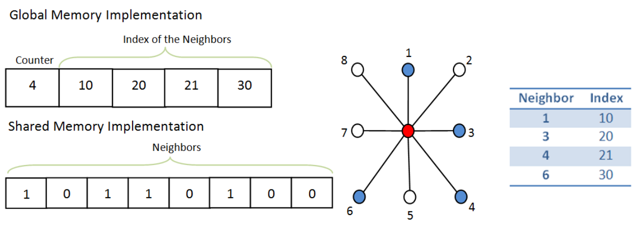

Figure 7: Comparison of the data structures used in the global memory and shared memory implementations. In graphic connections between the red point and its neighbors is represented using colors, where color white means that there is not a connection and the color blue means that there is a connection.

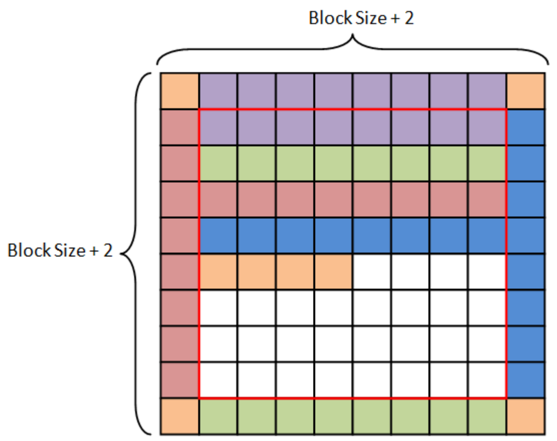

Figure 8: Scheme followed by the algorithm to copy data from the global memory to the shared memory in order to calculate update position of each point.

Figure 8 shows how a block organizes the copy of data from the global memory to the shared memory; the colors represent the way in which threads of each block copy the information of the additional points from the global memory to shared memory.

2.3.4 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, i.e. shared and coalescence memory implementations, and we used the same algorithm implemented in the previous approach to copy the position of the points in the mesh from global memory to shared memory.

3 Experimental Setup and Results

In order to compare the sequential, CPU-based parallel and GPU algorithms, an experimental test was developed. In the test several performance variables were measured, such as accuracy, 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 ($2.2GHz$) and 2 Gb of RAM memory. For the implementation in

Table 1: Charateristics of the meshes used in the experimental test

Name	Triangles	Edges	Points
Mesh 1	450	704	256
Mesh 2	24642	37184	12544
Mesh 3	61250	92224	30976
Mesh 4	85698	128960	43264

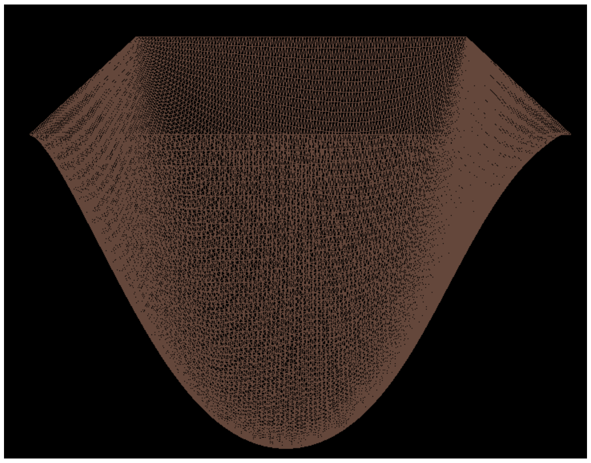
Figure 9: Simulation scenario used in the experimental test.

the GPU, the tests were conducted using a fixed block size for the different resolution meshes. The experimental test was carried out in a machine with a Nvidia GeForce 8800 GT GPU.

The simulation was be composed of a triangular mesh. We used four meshes with different sizes, Table 1 shows the characteristics of the meshes. We believe using different size meshes is important because often researchers have problem specific meshes, and we wanted to show the scope of improvement on different sized meshes.

In order to conduct the experimental test, a perturbation on the physical model of the mesh, produced by the gravitational force, was applied . To avoid the fall of the mesh to the ground of the environment, the boundary nodes of the mesh were fixed to the initial position during the simulation. Figure 9 shows the visualization of the simulation setup used .

Figure 10 and 11 presents the results of the execution time obtained for each of the approaches described before versus the number of points are possessed by each of the meshes evaluated.

Aditionally, figure 12 and 13 shows the speed-up achieved for each of the approaches and meshes evaluated during the experimental tests.

In addition to performing the experimental tests described previously we wanted to compare different implementations. Therefore, an experimental test was also carried out, in order to explore whether a relationship exists between the execution time of the algorithm implemented on the GPU and the block size used for spatial decomposition. The results of these experimental test can be observed in Table 2.

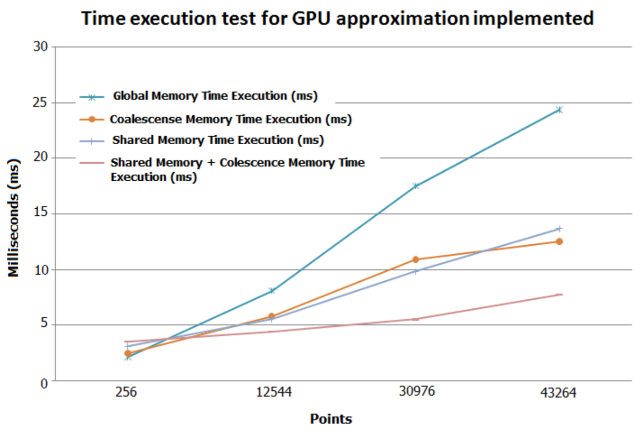

Figure 10: Time execution for each of the meshes evaluated for the GPU-based approaches.

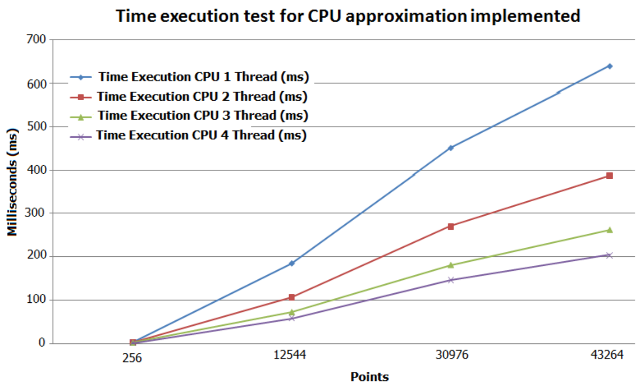

Figure 11: Time execution for each of the meshes evaluated and parallel approaches implemented on the CPU.

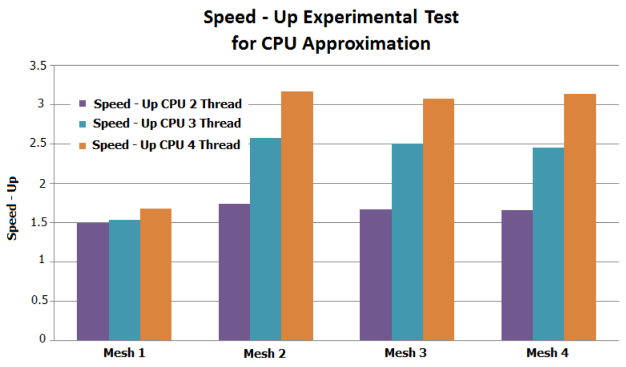

Figure 12: Speed-up for CPU multi-threaded approaches compared to single thread, using meshes of different sizes.

Finally, a test was carried out to evaluate the accuracy of the GPU algorithm. The error was calculated from the difference between the position of each one of the nodes in the sequential algorithm and the GPU algorithm. Figure 14 shows the evolution of the error calculated between the GPU algorithm and the sequential algorithm while the simulation iterates, showing a relatively stable error rate, that most importantly, does not effect the visual simula-

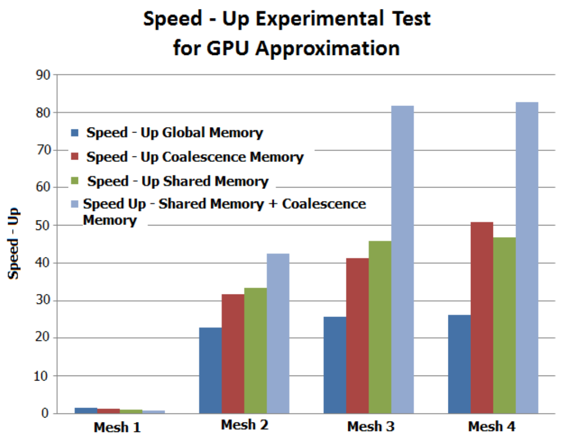

Figure 13: Speed-up for GPU approaches, using meshes of different sizes, measured against single-core CPU implementation.

Table 2: Relation between the block size and the time execution of the GPU implementation

Block Size	Number of Threads by Block	Time Execution Mesh 1 (ms)	Time Execution Mesh 2 (ms)	Time Execution Mesh 3 (ms)
2	4	17.1	39.28	53.75
4	16	8.64	18.59	25.26
8	64	8.34	17.6	24.5
16	256	8.096	17.52	24.37

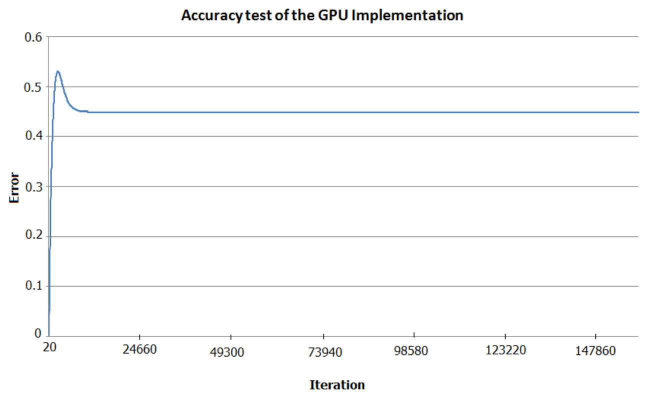

Figure 14: Accuracy test to measure the difference between the results of the sequential algorithm and GPU algorithm, specifically squared error is measured.

tion. This error is a result of the ordering of the iterations, the GPU versions of the algorithm are no less accurate because of the non-increasing and bounded error.

4 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 10 and 11, allow one to conclude that the time execution of the CPU-based parallel algorithm decreases when compared to sequential time execution. Furthermore, the time execution of the CPU-based parallel algorithm 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, such as *mesh*1, 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, basically not enough points in the mesh, therefore less GPU-cores are used, consequently the highly parallel GPU is not used effectively.

This same result is visible by analyzing Figure 12 and 13 where the speed-up of the CPU-based parallel and GPU algorithms can be observed. 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. The coalesced memory approach is very flexible since it can represent arbitrary geometry, and is the simplest strategy to be implemented.

Moreover, from Figure 10 and 11 it is possible to conclude that, if it is necessary to simulate the deformation of meshes with up to $\approx 43K$ points for real time applications, the best option is to apply the approaches that use the GPU, since these can provide for a computation time less than $16ms$, and obtain an approximate update frequency of $60Hz$. This frequency guarantees the interactivity of the simulation in a real-time surgical simulator. Whereas, the best CPU algorithm was 15x slower.

In the experimental test carried out to evaluate how block size, affects the time execution of the algorithm in the GPU, we can see that the larger the block size, the shorter the time execution of the algorithm, and therefore, better performance. This can be explained considering that the CUDA compiler, and thread scheduler, schedule the instructions as optimally as possible to avoid register memory bank conflicts. They achieve best results when the number of threads per block is a multiple of 64, i.e. in the experimental test with block size of 8 and 16. Aditionally if the block size is larger, CUDA incurs a lower cost of scheduling to arrange the tasks of each of the threads as there are fewer blocks. However, after 64 threads are used we do not notice a large speedup. This is mainly because the mesh size was to small, if a larger mesh is used a more substantial speedup is seen.

With respect to the accuracy test of the GPU algorithm, it is possible to be observe in Figure 14 that the error reached with this method is not considerably large, if it is considered that the error depends on the number of points in the mesh, and that in this case the mesh is evaluated with $10K$ points. Furthermore, in the solver algorithm used to calculate the deformation, one can observe that the result is highly dependent of the order in which each point is calculated, mainly due to the new position of the point depends on the current position of its neighbors and therefore, is not the same calculation, because it depends on if the neighborhood positions have already been calculated, or not. In this case, we can conclude that for use in a surgical simulation, the error is not sufficiently large to affect the visual perception of the surgeon during the simulation and is relatively stable .

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. However, as mentioned above a problem that persists when a GPU approach using shared memory, is the need for a structured mesh (very common). Additionally, the approaches implemented on the GPU allow the real-time computation of the deformation in meshes with high degree of resolution, which logically results in a better realism of the visualization of the anatomical structures in the simulator.

5 Future Work

In future research we will have three purposes (i) improve the realism of physical model implemented to simulate the deformation. For this purpose we will implement a physical algorithm to guarantee the volume conservation of the deformable object. For example, using *mass-spring* model with a tetrahedral mesh or implementing special algorithms used in surgical simulation, the volume distribution method.(ii) implement an algorithm to improve the performance of the simulation. The algorithm presented before have some problems of performance, because it calculates the spring force several times for each point. We propose to evaluate an edge-centric approach to resolve this performance problem. This approach calculates the spring force once for point, avoiding redundant calculations, and should increase update frequency of the simulation. (iii) Incorporate the implementations realized to the complete surgical simulation system and carry out a experimental test to evaluate the performance of the algorithm using real surgical environments.

References

[1] D. Woodrum, P. Andreatta, R. Yellamanchilli, L. Feryus, P. Gauger and R. Minter. Construct validity of the LapSim laparoscopic surgical simulator. *American Journal of Surgery*, Volume 191, pp. 28 32, 2006.

[2] R. Aggarwal, T. Grantcharov, K. Moorthy, J. Hance, A. Darzi. A competency-based virtual reality training curriculum for the acquisition of laparoscopic psychomotor skill. *The American Journal of Surgery*, Volume 191, pp. 128133, 2006.

[3] A. Nealen, M. Muller, R. Keiser, E. Boxerman and M. Carlson. Physically based Deformable Models in Computer Graphics. *Proceedings of Computer Graphics Forum*, p.p. 809 836, 2005.

[4] G. Szekely, Ch. Brechbuehler, R. Hutter, A. Rhomberg, N. Ironmonger and P. Schmid. Modeling of Soft Tissue Deformation for Laparoscopic Surgery Simulation. *Medical Image Analysis*, Volume 4, Issue 1, p.p. 57 66, 2000.

[5] U. Meier, O. Lopez, C. Moserrat, M. Juan, and J. Alcaniz. Real-time Deformable Models for Surgery Simulation: A Survey. *Computer Methods and Programs in Biomedicine*, Volume 77, Issue3, p.p. 183 197, 2005.

[6] T. Sorensen and J. Mosegaard. An Introduction to GPU Accelerated Surgical Simulation. *Biomedical Simulation*, Volume 4073, pp. 93 104, 2006.

[7] NVIDIA. *CUDA Programming Guide v. 1.1*.

[8] J. Georgii, F. Echtler R. Westermann. Interactive Simulation of Deformable Bodies on GPUs. *Proceedings of Simulation and Visualisation 2005*, pp. 247 258, 2005.

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

[10] C. Diaz, D. Posada, H. Trefftz and B. Bernal. Development of a Surgical Simulator to Training Laparoscopic procedures. *International Journal of Education and Information Technologies*, Issue 1, Volume 2, pp. 95 103, 2008.

[11] J. Brown, S. Sorkin, J.C. Latombe, K. Montgomery and M. Stephanides. Algorithmic tools for real-time microsurgery simulation. *Medical Image Analysis*, Issue 6, Volume 3 pp. 289 300, 2002.

[12] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism*. OReilly, 2007.

[13] 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.

[14] 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.