A Parallel Mass-Spring Model for Soft Tissue Simulation with Haptic Rendering in CUDA

C.E. Etheredge
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
c.e.etheredge@student.utwente.nl

ABSTRACT

Virtual surgery simulators are an emerging training method for medical specialists. A number of techniques have previously been proposed for simulating tissue for these simulators, one of which is the mass-spring model.
This paper aims to improve upon the perceived realism of virtual simulator systems by redesigning the underlying physics algorithms so that accurate simulations with full and smooth haptic interaction can be performed.
We present a novel mass-spring model algorithm designed to accurately simulate soft human organ tissue, both through visual (graphic) and haptic perception (touch). The algorithm is implemented as a massively parallel algorithm running on the *NVIDIA® CUDA®* computing architecture, allowing it to simulate human tissue at a rate of at least 1000 Hz, the minimal rate at which we can ensure smooth haptic interaction.
The mass-spring model is integrated as part of *VICTAR®*, or *Virtual Competence Training Area*, an extendable software framework designed for virtual surgical training[1]. VICTAR® features haptic device handling and a scripting engine that allows for easy prototyping of certain medical scenarios and also serves as the primary method for benchmarking. The results shows that the presented implementation is capable of sustaining rates above the requirements and is therefore fast enough to ensure smooth haptic interaction.

Keywords

Surgical simulation, Mass-spring models, Soft-body dynamics, Haptic feedback, Parallel computing, GPGPU, CUDA

1. INTRODUCTION

Virtual surgery systems must be capable of creating a virtual environment that accurately represents the human anatomy and physiology. When working with virtual medical training scenarios, the simulation must be accurate enough to be used for extensive practice of various treatment techniques by medical students. The system accepts user input in the form of a haptic device that represents a virtual surgical tool within the system that the user can move and rotate. Output of the system is provided visually, by means of a graphics renderer that outputs to a monitor, as well as haptically through the same haptic device. By sending the calculated set of forces to the device, the user will feel resistance, e.g. as the virtual tool collides with the model in the simulation. The accuracy of visible deformation and haptic perception of the tissue (sense of touch) in response to the interaction of the virtual surgical tool is especially important. It is the main factor in the simulator's perceived realism and faces close scrutiny from experts working in the field.

Soft tissue can be simulated by a handful of methods. Our research focuses on the mass-spring model, where a body is modeled as a set of point masses interconnected by many elastic springs that follow general physical laws. The concept behind the model is fairly straightforward, and the algorithm required to simulate the model generally does not require any complex operations. In practice however, common serial implementations are far from being fast enough due to the high number of masses and springs required for accurate and precise simulated models.

The inherent problem of virtual surgery systems with high perceived realism is the required performance of the simulations. At one end, the model used in the simulation must be of high resolution so that the user can clearly distinguish visual and haptic features and can perform operations as defined in a medical training scenario. At the other end, the simulation must run at a fast enough rate to allow smooth interaction with the haptic devices. These haptic devices contain actuators that apply a set of forces to the user's hand and generally require forces to be sent at a real-time rate of 1000 Hz. As a consequence, any simulation that wants to achieve smooth haptic interaction must run at a rate that is at least equal to this haptic device rate.

Our research attempts to solve the performance problem by redesigning the mass-spring model algorithm. Because of the simplicity of the mass-spring model, it is possible to rewrite the model algorithm to take advantage of existing parallel hardware, so that many masses within the model can be processed in parallel. Existing parallel hardware can be found in modern consumer and workstation grade graphics cards (also referred to as GPUs) such as the *NVIDIA® GeForce®* and *Tesla®* series. These cards consist of a very complex and large hardware architecture optimized for parallel execution of programs. Development is made possible by frameworks such as *NVIDIA CUDA* that run on top of these architectures and contain the necessary tools to implement and execute parallel code while communicating with the computer hosting the GPU

[1]The VICTAR® platform is a development of Vrest Medical, Institutenweg 38, NL - 7521 PK Enschede.

(also referred to as the CPU), further explained in section 3.1. The GPU is essentially a computer within a computer with very high parallel computational power and memory bandwidth at low cost.

The idea of parallelizing the mass-spring algorithm is not new. Section 2 describes previous work with successful parallel implementations of the mass-spring model. Though, these works merely represent the initial steps of the mass-spring model in the parallel world and do not explicitly state any requirements for haptic interaction. The only required system output is the visual representation, which is allowed to run at a rate as low as the monitor's refresh rate (60 Hz). As a result, performance is not high enough to allow the algorithms to be used in real-world virtual surgery systems.

Our research is novel as it provides a parallel mass-spring algorithm that is explicitly designed to run at real-time haptic rates with the goal of becoming an integral part of the existing virtual surgery system of VICTAR®. In practice, the algorithm runs at a haptic frame rate (1000 Hz and above) instead of a graphics rate (60 Hz), implying a required speedup of at least 15x so that smooth realistic haptic interaction with the model is possible.

However, there are many pitfalls and hurdles that are in the way of a correctly functioning parallel algorithm. Threading issues such as race conditions, unnecessary synchronization, limited memory bandwidth, resource usage and other bottlenecks have to be solved in order for the algorithm to function correctly. Many of these bottlenecks can cause stalls and other delays in the GPU hardware architecture that significantly decrease the efficiency at which the parallel algorithm is executed. This is further explained in section 4, in which our research goes into detail on how to rewrite the mass-spring algorithm in parallel.

Our research then focuses on the data structures and shows how these can be structured appropriately for CUDA and the GPU hardware architecture. We then design the integration of our algorithm within VICTAR® to allow visual rendering and haptic interaction with the model by using an existing graphics renderer and haptic device handler. Each of these are contained in different subsystems or host threads, running at different rates. A certain degree of data transfer and communication is necessary between these subsystems in order for data to, for example, flow from the haptic devices to the mass-spring algorithm and vice versa. The system is completed by a buffering strategy that is used to allow safe data transfers while minimizing performance overhead and latency between all functional parts of VICTAR®.

Finally, the results in section 6 show that we have succeeded in creating a mass-spring algorithm that meets the earlier stated performance requirements while operating as an integral part of VICTAR®. Our implementation is benchmarked using two complex human data sets that show the algorithm performing at sustained rates exceeding those of the haptic device as can be seen in table 2. Smooth haptic interaction is therefore possible, allowing it to be used for virtual surgery and medical training.

2. RELATED WORK

For the past two decades, physical modeling of deformable objects has been an area of extensive research within the field of computer graphics and medical technology. In 1987, [16] was the first to incorporate physical properties into a graphical object, creating *elastically deformable models* that were capable of responding to external forces and constraints. The underlying mechanics of these models were initially used with the *finite element method (FEM)*, such as in [2] and [3] for animating muscle deformations in anatomically based characters.

While FEM is considered to be very accurate, it comes with high inherent numerical complexity. Therefore, research has advanced towards simplifying the model to find a more balanced tradeoff between accuracy and performance. This led to the adoption of the simpler *mass-spring model*, an approximated model where the object is subdivided into discrete masses that are interconnected by springs. Springs have attributes based upon the physical properties of the object. While the model remains an approximation, it was found to have a fair accuracy trade-off and has since been widely used in various areas such as virtual surgery [8], rigid cloth simulation [14], muscle deformation [12] and others.

Recently, research into more realistic virtual surgery simulation has been growing. Additional degrees of realism are added by increasing the complexity of the objects as well as allowing the user to have haptic interaction with the model. Though these two factors play an important role in improving perceived realism, they also impose severe limits on the computation time and require significant efforts in improving the performance of the underlying physical model [7].

In this context, extensive work is being done in implementing the physical models on emerging hardware architectures such as Graphical Processing Units (GPUs) instead of conventional software approaches. This shift towards new cutting-edge architectures, made possible by frameworks such as OpenCL and CUDA, has been an important subject of research in the last few years and is also known as General-Purpose GPU (GPGPU) computing.

Specifically, CUDA provides the technology to implement parallel algorithms by using consumer grade graphics cards (GPUs) that are capable of massive concurrent execution of instructions, resulting in potentially high performance gains.

Previously, mass-spring models have been implemented on the GPU using different techniques. In [10][9] [11], parallel GPGPU-based approaches are presented with significant speedups when compared to reference CPU implementations. In more recent work [15] [4], different CUDA implementations of the mass-spring model for surgery simulation are investigated and evaluated, showing potential speedups compared to earlier GPU-based approaches.

3. BACKGROUND

3.1 GPGPU and CUDA architecture

3.1.1 Architectural overview

The CUDA framework exposes the parallel processing power of the GPU in the C programming language. The GPU is considered as a highly parallel general-purpose processor with the ability of running numerous fine-grain threads. Conventional GPGPU programming can be seen as a form of stream processing, where identical operations are performed on each element of the input set in parallel. Once implemented in CUDA, an operation can be invoked by a single line of code that instructs the device to perform the operation in parallel with a specific organization of threads.

NVIDIA released the first CUDA-capable GPU in 2006 and is currently shipping its third-generation CUDA capable architecture. This includes the *NVIDIA Tesla C2050*, one of the high-end GPUs targeted at the high performance computing market, supporting 448 streaming multiprocessors (SMs) and 3 GB of GDDR5 memory. Each SM supports on the order of a thousand parallel threads and is equipped with a large register file, giving each thread its own dedicated set of registers. A high-end GPU with many SMs can thus sustain tens of thousands of threads simultaneously. Each SM also contains high-bandwidth, low-latency on-chip shared memory, while at the same time providing its threads with direct read/write access to off-chip DRAM [5][13][6]. The key challenge lies in implementing our parallel algorithm as efficiently as possible by utilizing maximum computational power while avoiding operations that cause unnecessary memory latency and synchronization.

3.2 Mass-spring model

The basic fundamentals of the algorithm in question are easy to understand. A mass-spring model consists of a set of mass points M_i with $i = [0, N]$ where N represents the number of total masses. The model is interconnected with springs S_{ij} connecting any two arbitary mass points M_i and M_j.

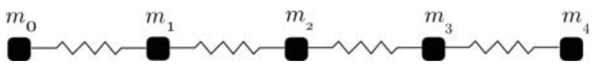


Figure 1. A one-dimensional case of a mass-spring system.

Figure 1 represents the simple one-dimensional case of a mass-spring model as detailed in [1]. This particular case contains masses M_0, M_1, M_2, M_3, M_4 and springs S_{01}, S_{12}, S_{23}, S_{34}. In the mass-spring model, forces that are exerted on the connected masses are summed according to Newton's second law of motion:

$$m_i a_i = \sum_{j \in S_i} \vec{F_{ij}} \tag{1}$$

where S_i is the set of connected masses for m_i, a_i is the acceleration of m_i, and $\vec{F_{ij}}$ is the force exerted by the spring connecting the two masses m_i and m_j that can be obtained from Hooke's law:

$$\vec{F_{ij}} = k_{ij}(l_{ij} - l^0_{ij})\frac{l_{ij}}{|l_{ij}|} \tag{2}$$

where k_{ij} is the spring stiffness, l_{ij} is the current spring length or distance between masses m_i and m_j, and l^0_{ij} is the spring rest length in equilibrium.

To obtain a set of useful results, equation 1 is numerically integrated using *Verlet integration*. Verlet integration is one of many numerical methods to integrate Newton's equations of motion and is generally used to calculate the trajectories of masses. Similar lower and higher order methods are Euler, Backward Euler, Leapfrog and Runge-Kutta [11]. The main advantage of the Verlet integration method is that only the two last positions of the mass M_i are needed to calculate the updated position (saving performance) while the error of the outcome stays reasonably low (reasonable accuracy):

$$x_i(t + \Delta t) = 2x_i(t) - x_i(t - \Delta t) + \frac{1}{2} f_i \Delta t^2 \tag{3}$$

Where t is the time, Δt is the fixed time step and f_i is the total force or acceleration of the mass. Other factors such as spring damping and gravity can be easily added by extending f_i beyond the acceleration in equation 1:

$$f_i = a_i + -d_i \vec{v_i} + \vec{g} \tag{4}$$

Where d_i is the damping factor of the mass and $\vec{g}$ is the gravity vector.

3.3 Integration with VICTAR®

Our algorithm is implemented and tested using VICTAR®. Figure 2 shows a diagram of VICTAR® including all the relevant threads and subsystems running at different update rates. Various synchronization and threading strategies need to be employed in order to integrate the physics model properly. For instance, as the graphics subsystem is displaying the model, there needs to be a certain degree of synchronization with the algorithm running in a CUDA thread. The haptic devices also require interaction with the CUDA thread for manipulation of the physics model.

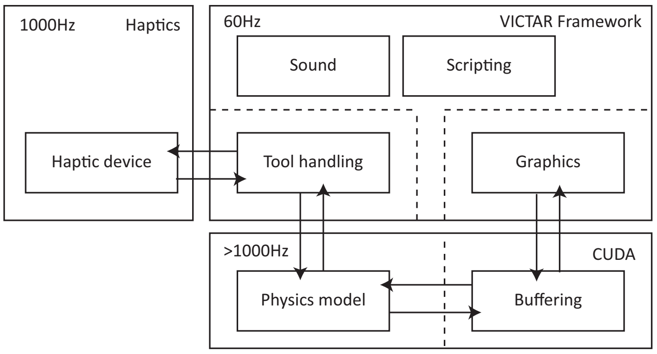


Figure 2. Our physics model and its interaction with various threads and subsystems within VICTAR®.

4. MASS-SPRING ALGORITHM

At the heart of our system lies the algorithm that calculates all elements within the mass-spring model by using the physics equations in section 3.2. Each time the algorithm is invoked on the model, the result is an updated model containing the masses and springs with their positions and other variables adjusted according to any applied (tool) forces within a predefined time step of Δt.

4.1 Model layout

The basic layout for our mass-spring model in the three dimensional case is represented by a uniform grid consisting of masses connected by springs as can be seen in figure 3.

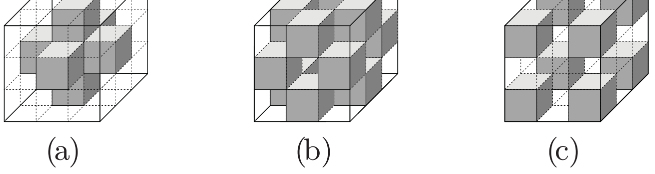


Figure 3. Three-dimensional uniform grid layout connecting the center mass to at most 26 neighbouring masses (in grey) at Euclidean distance (a) 1, (b) $\sqrt{2}$ and (c) $\sqrt{3}$.

As in [15], the uniform grid is chosen purely for simplicity and structural integrity of the model. Every mass within the grid is connected to all of its neighbouring masses, forming springs, all of which are within a 3x3x3 grid surrounding the mass. The rest length r_{ij} of each spring is defined as the initial Euclidian (or ordinary) distance between a mass M_i and any of its neighbouring masses M_j forming the spring. Because of the grid layout, this distance is either 1, $\sqrt{2}$ or $\sqrt{3}$. For every mass we can define at most $3*3*3-1=26$ neighbouring masses (and thus springs), while e.g. masses at the outer surface of the grid will generally have fewer neighbours.

Keeping this model in mind, we can now move forward to the implementation of the mass-spring algorithm.

4.2 Naive serial implementation

The most straightforward and intuitive implementation of the mass-spring algorithm simply iterates over every spring S_{ij} (connecting arbitrary masses M_i and M_j where $0 \leq i,j < N$) within the model and applies equations 2 and 3 as can be seen in algorithm 1.

Algorithm 1 Straightforward serial implementation

```
 1: for all masses $M_i$ do
 2:     $\vec{F_i} \leftarrow 0$ {initialize total force}
 3: end for
 4: for all springs $S_{ij}$ do
 5:     get mass positions $\vec{x_j}(t)$ and $\vec{x_i}(t)$
 6:     determine spring length $\{l_{ij} \leftarrow \vec{x_j}(t) - \vec{x_i}(t)\}$
 7:     calculate spring force $\{\vec{F_{ij}} \leftarrow k_{ij}(l_{ij} - l^0_{ij})\frac{l_{ij}}{|l_{ij}|}\}$
 8:     calculate force on $M_i$ $\{\vec{f_i} = \vec{a_i} = \vec{F_{ij}} * \frac{1}{2}\}$
 9:     calculate force on $M_j$ $\{\vec{f_j} = \vec{a_j} = -\vec{F_{ij}} * \frac{1}{2}\}$
10:     update accumulated forces $\vec{F_i}$, $\vec{F_j}$ by adding $\vec{f_i}$, $\vec{f_j}$
11: end for
12: for all masses $M_i$ do
13:     integrate $\{\vec{x_i}'(t+\Delta t) \leftarrow 2\vec{x_i}(t) - \vec{x_i}''(t-\Delta t) + \frac{1}{2}\vec{F_i}\Delta t^2\}$
14:     set mass position $\vec{x_i}'(t+\Delta t)$
15: end for
```

For each mass M_i the total force exerted by the connected springs is accumulated into $\vec{F_i}$ and finally integrated using Verlet integration to determine the new mass position x_i.

4.3 Parallel implementation

The first trivial step towards parallelizing algorithm 1 is to process the existing for-loops in parallel instead of doing sequential iterations. However, when each of the iterations is processed by a different thread in parallel, the accumulation of forces $\vec{F_i}$ and $\vec{F_j}$ will in practice lead to race conditions due to different threads trying to read and write these values at the same time. This poses a serious threat to the stability of the model.

The problem could be mitigated by using synchronization and thus making the accumulation atomic so that only one thread at a time can modify its value. This would however degrade performance due to excessive locking. Instead, it is also possible to rewrite the algorithm and avoid any synchronization at all.

Algorithm 2 has been written in such a way that it is now iterating over every mass M_i and subsequently each of its neighbouring masses M_j. Each combination of a mass and its neighbour represents a spring S_{ij} so that all springs in the model are implicitly iterated. The for-loop can be iterated by many threads in parallel. Race conditions are now eliminated because the only data that is updated ($\vec{x_i}'(t+\Delta t)$) is unique and never accessed by any other thread within the same time step.

Algorithm 2 Initial parallel implementation

```
 1: for all neighbouring masses $M_i$ do
 2:     get mass position $\vec{x_i}(t)$
 3:     $\vec{F_i} \leftarrow 0$ {initialize total force}
 4:     for all neighbouring masses $M_j$ do
 5:         get mass position $\vec{x_j}(t)$
 6:         determine spring length $\{l_{ij} \leftarrow \vec{x_j}(t) - \vec{x_i}(t)\}$
 7:         calculate spring force $\{\vec{F_{ij}} \leftarrow k_{ij}(l_{ij} - l^0_{ij})\frac{l_{ij}}{|l_{ij}|}\}$
 8:         accumulate total force $\vec{F_i}$ $\{\vec{F_i} \leftarrow \vec{F_i} + \vec{F_{ij}} * \frac{1}{2}\}$
 9:     end for
10:     integrate $\{\vec{x_i}'(t+\Delta t) \leftarrow 2\vec{x_i}(t) - \vec{x_i}''(t-\Delta t) + \frac{1}{2}\vec{F_i}\Delta t^2\}$
11:     set mass position $\vec{x_i}'(t+\Delta t)$
12: end for
```

Note that the algorithm has a number of implicit memory accesses, e.g. the list of neighbouring masses at (4) and variables k_{ij} and l^0_{ij} at (7). For the sake of readability, these have been kept implicit and will be further expanded in the next section.

4.4 Extending the implementation

Extending the initial parallel algorithm to target platform CUDA poses a few difficulties. For one, as outlined in section 3.1.1, care should be taken to ensure efficient data structuring so that potential bottlenecks are pinpointed and memory access is minimized.

Algorithm 3 Extended CUDA implementation

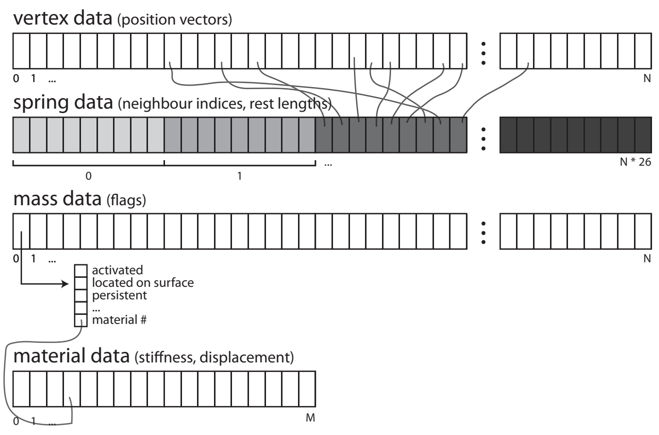
```
 1: for all masses $M_i$ do
 2:     get mass position $\vec{x_i}(t)$
 3:     get mass properties $p_i$
 4:     synchronize threads
 5:     if $p_i.active$ then
 6:         $\vec{F_i} \leftarrow 0$ {initialize total force}
 7:         for $k = 0 \rightarrow 26$ do
 8:             get neighbour mass at $k$ $\{M_j \leftarrow N_i(k)\}$
 9:             if $M_j$ exists then
10:                 get mass properties $p_j$
11:                 if $p_j.active$ then
12:                     get mass position $\vec{x_j}(t)$
13:                     get spring properties $k_{ij}$ and $l^0_{ij}$
14:                     determine spring length $\{l_{ij} \leftarrow \vec{x_j}(t) - \vec{x_i}(t)\}$
15:                     calculate spring force $\{\vec{F_{ij}} \leftarrow k_{ij}(l_{ij} - l^0_{ij})\frac{l_{ij}}{|l_{ij}|}\}$
16:                     accumulate total force $\vec{F_i}$ $\{\vec{F_i} \leftarrow \vec{F_i} + \vec{F_{ij}} * \frac{1}{2}\}$
17:                 end if
18:             end if
19:         end for
20:         get mass position $\vec{x_i}''(t-\Delta t)$
21:         synchronize threads
22:         perform collision handling
23:         integrate $\{\vec{x_i}'(t+\Delta t) \leftarrow 2\vec{x_i}(t) - \vec{x_i}''(t-\Delta t) + \frac{1}{2}\vec{F_i}\Delta t^2\}$
24:     else
25:         keep old position $\{\vec{x_i}'(t+\Delta t) \leftarrow \vec{x_i}(t)\}$
26:     end if
27:     set mass position $\vec{x_i}'(t+\Delta t)$
28: end for
```

Algorithm 3 shows the extended algorithm with all memory access transactions explicitly stated, containing predictable coalesced transactions as well as unpredictable random transactions. Every iteration runs in a separate thread and requires multiple memory access transactions in order to retrieve data. This data is retrieved with different access characteristics: at (2) $\vec{x_i}(t)$ will always be retrieved by thread M_i (i is always known beforehand, as is also the case for (3), (20) and (27)) while the retrieval of p_j and $\vec{x_j}(t)$ at (10) and (12) is effectively a random memory access because j can be any arbitrary mass connected to M_i. Memory transactions (2), (3), (20) and (27) rely on i which is unique to each parallel thread. i is already known in advance and these transactions can therefore be predicted and will be optimized for data coalescing by CUDA. With little added synchronization as in (4) and (21), these transactions - performed by threads in parallel - will be combined into single transactions, saving considerable memory bandwidth.

At (8) function $N_i(k)$ retrieves the index j for the neighbouring mass M_j represented by a particular $0 \leq k < 26$ for M_i. N_i thus represents a data structure containing at most 26 indices for every mass M_i. The algorithm also contains the necessary code to perform collision handling for haptic interaction, which is left simplified in this listing and will be elaborated in section 5.1.

4.5 Data structures

Data in memory that is accessed by predictable transactions should be structured in such a way that allows for coalescing according to a set of GPU-dependent alignment rules. As a general rule for CUDA, data elements should always be aligned to 32-bit, 64-bit or 128-bit words so that multiple parallel transactions at sequential addresses can be combined into single transactions.



vertex data (position vectors)
0 1 ...
N
spring data (neighbour indices, rest lengths)
0
1
...
N * 26
mass data (flags)
0 1 ...
N
activated
located on surface
persistent
...
material #
material data (stiffness, displacement)
0 1 ...
M

Figure 4. Layout overview of (aligned) data structures, from top to bottom: *vertex data*, *spring data*, *mass data* and *material data*.

Because memory space and bandwidth must be minimized at all times, we have added several additional optimizations. As the number of springs (at most $N * 26$) quickly increases with more complex models, we have chosen to classify every spring S_{ij} or mass M_i by a material m. Every spring S_{ij} is assigned a material m that globally defines spring properties such as the stiffness k_{ij}, which is now no longer unique to every spring rather than to the group of springs defined by m. By keeping the number of materials limited to L (e.g. $L = 16$), we only require to store L instead of $N * 26$ spring properties, saving considerable memory usage and bandwidth.

Our data structures are organized as simple one-dimensional arrays of (aligned) data. Figure 4 gives an overview of how these structures are laid out in memory. The *vertex data* contains positions in $\vec{x}$ as accessed by algorithm 3 at (2), (12), (20) and (27). The *spring data* contains the indices for $N_i(k)$ at (8) as well as unique spring properties such as l^0_{ij} at (13). The *mass data* contains mass properties such as flags p at (10) and the earlier discussed material m. Finally, the *material data* contains global spring properties such as k_{ij} for every material m applied to the group of springs as defined by m.

4.6 Additional optimizations

Algorithm 3 already contains the basics for some important performance optimizations. Note that at (5), the algorithm checks whether M_i is currently marked as active. If M_i is not an active mass, all computations can be skipped early and the position $\vec{x_i}$ remains unchanged. By dynamically changing $p_i.active$ for all masses in the model, computational power is saved that would otherwise be wasted on masses that barely move. For example, a mass M_i can be marked inactive if the total spring force $|\vec{F_i}|$ falls below a certain small threshold ϵ (e.g. $\epsilon = 0.001$). If M_i is barely affected by any spring forces, it is likely that this will continue for the next iterations, so it makes sense to deactivate the mass until $|\vec{F_i}|$ exceeds the threshold ϵ, e.g. in the case when the mass is displaced by a virtual tool and is subsequently reactivated. We have optionally implemented this optimization strategy as can be read in section 6.

5. SYSTEM DESIGN

Given the extended algorithm implementation in the previous section, we now have a means to calculate the positions within the mass-spring model. At this point, it is important to integrate the algorithm into a system that handles user input (through *haptics*) and output (through *haptics* and *graphics*) so that the mass-spring model can actually be used in practice.

As explained in section 3.3, user input is performed by one or two *haptic devices* resembling a virtual tool that can be used to manipulate the mass-spring model. Moving the haptic device will result in identical movements of the appropriate virtual tool in the system. This, together with the force feedback output capabilities of the device, where the force of any masses acting on the virtual tool will result in a force being applied to the user's hand, will provide the user with a full sense of haptic interaction with the model.

Another form of system output is of course the visual representation of the model as generated by the system's graphics subsystem. VICTAR® implements this subsystem on *Microsoft® DirectX® 11* that also provides shader capabilities of *Shader Model 4.0*. A specialized multi-pass shader rendering technique is used to render the visual representation of the mass-spring model. This however falls beyond the scope of this research.

In figure 2, three subsystems can be clearly distinguished, each running in their own thread on the host (CPU) at different rates. The *CUDA* subsystem represents the CUDA host thread that executes the mass-spring algorithm on the GPU at a dynamic rate, whereas the *haptics* subsystem represents a host thread that interfaces with the haptic device driver at a constant rate of 1000 Hz. Finally, the *framework* contains the renderer that generates the visual representation of the model as well as other auxiliary functions in the framework such as scripting and sound functionality.

Figure 5 provides a diagram of the complete data flow for the system and also describes whether the data is residing on the CPU or GPU.

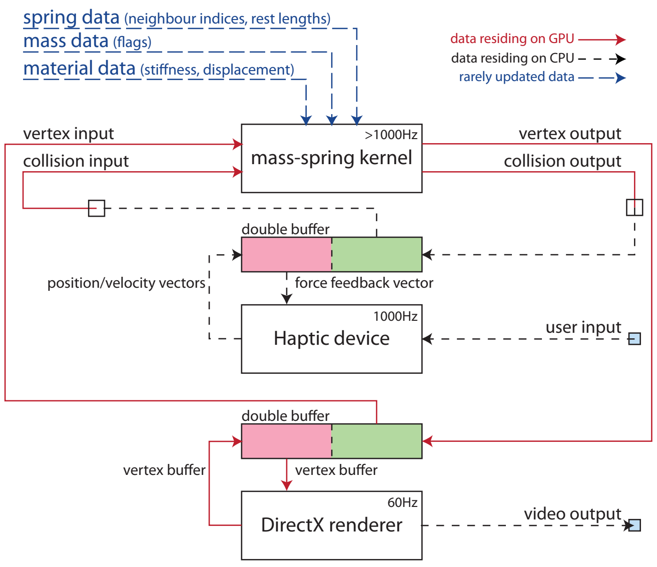


Figure 5. Detailed overview of data flow within the system as implemented by VICTAR® consisting of various subsystems (threads) as defined by the white boxes. Squares represent the memory copies required to move the data from CPU to GPU or vice versa. Double buffering, where applicable, is used to prevent threading issues.

5.1 Haptic synchronization

The first important form of user interaction is done with haptic devices. We are using the *SensAble® PHANTOM Omni®* haptic devices, seen in figure 6, that provide two-way haptic interaction: the device has sensors in six degrees of freedom to register the device's current position (as well as velocity and rotation) while built-in actuators provide a way to apply force feedback to the device in three degrees of freedom. The device is controlled by the *Haptic Device API (HDAPI)* that schedules a haptic thread to run at a constant rate of 1000 Hz in which the sensors can be read out and an actuator force can be applied. The high haptics rate is necessary to prevent jerky oscillating movements of the actuators, where 1000 Hz is generally fast enough to provide a force that feels smooth and consistent at the user's hand.

The sensor data representing the position of the virtual tool is read as a *float vector buffer* (in the haptic thread) that must be passed into the mass-spring algorithm (in the CUDA host thread) with as little latency as possible. At the same time, a *float vector buffer* containing a force must be written (in the haptic thread) after being calculated by the mass-spring algorithm (in the CUDA host thread), again with as little as possible latency.

The fastest way to accomplish this would be to let both threads access these two vectors at the same location in memory. This could however result in either of the threads reading a value that is currently being written to by the other thread, yielding corrupt half-written data. Another option would be to apply synchronization through locking, but this would cause either of the threads to wait and stall, potentially degrading performance.

To prevent any issues, we employ a *double buffering* strategy with two identical co-existing buffers where each buffer is being exclusively used for either reading or writing. In the case of a single-reader single-writer situation, as with our system, buffer A is used for reading while B is used for writing. At any point in time when there are no threads reading or writing at all, buffers A and B will be flipped so that the reader is able to read the updated data in B while the writer can overwrite outdated data in A.

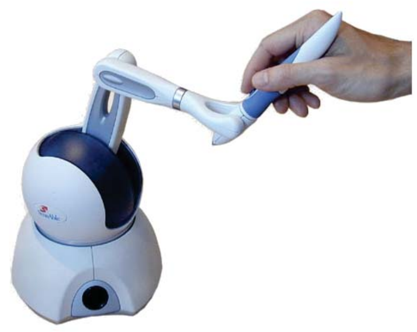


Figure 6. *SensAble® PHANTOM Omni®* haptic device with sensors in six degrees of freedom and actuators in three degrees of freedom, used for haptic user interaction.

Both vectors can now be synchronized as quickly as possible and without threading issues. Note that within the CUDA host thread, the vectors still require a copy from CPU to GPU, but at little cost. The mass-spring algorithm will use the tool's position vector to perform collision detection and displacement of the model as the virtual tool touches the model, and generates a force by summing the spring forces $\vec{F_{ij}}$ acting on the masses that are currently touched by the virtual tool. The user is therefore able to move the tool while feeling the model's masses colliding and giving resistance.

5.2 Graphics synchronization

At this point, the system can run fine with just haptic interaction between the user and the model. The next step however is to add visual output, which is accomplished by close cooperation of the CUDA subsystem and the DirectX renderer. The renderer takes a *vertex data buffer* as input and presents this buffer to the GPU through DirectX's API so that it is ultimately displayed on the monitor. Specialized shaders are used to make sense of the vertex data and turn it into a visual representation of the model. The renderer typically runs in sync with the monitor's vertical refresh rate, which we assume to be at a constant 60 Hz.

The shaders accept a vertex data structure that is identical to the structure used by the mass-spring algorithm in CUDA. The data structure is simply an aligned one-dimensional array of *float4 vectors* of size N. The renderer can therefore directly read any buffer that has been updated by the mass-spring algorithm. As both CUDA and DirectX will only access data that is residing on the GPU, the vertex data never has to leave the GPU, saving costly memory copies between GPU and CPU.

Note that the renderer and mass-spring algorithm run in different threads at vastly different rates. We need to ensure that, at no point in time, the renderer can ever read a vertex data buffer that is currently being written to by the algorithm. The renderer could theoretically render half-written corrupted vertex data due to the algorithm still

Table 1. Profiling results of the mass-spring algorithm in CUDA obtained by performing several test runs on arbitrary data sets.

Measurement unit or property	Profiler output
Kernel block size	256
Kernel occupancy	66.67%
Shared memory usage	0 KB
Register count	26

updating the buffer, or the renderer could end up locking the buffer (at 60 Hz) causing a severe performance degradation of the algorithm as CUDA will be waiting to get a lock. To prevent these threading issues we thus apply the same double buffering strategy as described in section 5.1 for vertex data.

6. RESULTS

We have implemented the parallel mass-spring model as described in the previous sections as part of VICTAR®'s virtual surgery system. The configuration of our target platform on which we will be testing the model and measuring results is a workstation with a *Intel® Core® i7-860* CPU with 6GB of RAM and a *NVIDIA Tesla C2050* graphics card supporting *NVIDIA CUDA 4.0*, the latest at the time of writing.

The *CUDA Visual Profiler* allows developers to peek at the internals of the mass-spring kernel in CUDA. Timing information such execution times and overhead on GPU and CPU as well as architectural features such as the register count, kernel occupancy and memory throughput can be measured and give insight about the current and maximum potential efficiency of the kernel. [13] and [6] give an in-depth overview of how these units can be used to profile the performance of a kernel and guide future optimizations. Table 1 contains the results of a performance profile run on the mass-spring algorithm as implemented in CUDA.

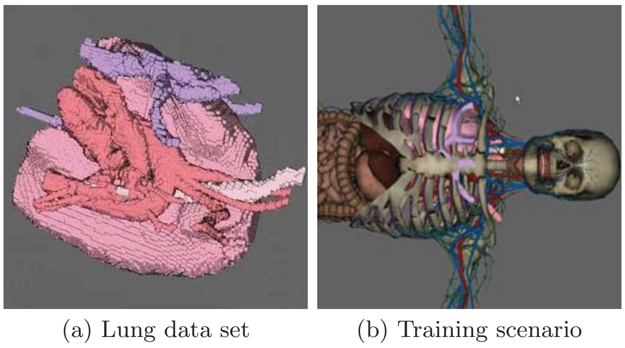

(a) Lung data set (b) Training scenario

Figure 7. Visual representation of the *lung* data set as a model and the VICTAR® lung surgery training scenario where the model is used.

By analyzing the data in 1, we can identify potential performance limitations that exist in the current implementation of the algorithm. Analysis shows that the algorithm kernel, due to its complexity, puts a heavy load on the GPU's registers. As explained in [13], the kernel occupancy shows the load or busyness of the GPU while executing the kernel. Each multiprocessor (SM) inside the GPU only has a fixed number of registers and cache memory resources available. As the register count increases, less threads can be scheduled to run concurrently and the GPU load decreases. In our case, the GPU is only loaded for 66%.

Table 2. Profiled performance results of the mass-spring algorithm in CUDA operating on two different data sets.

Model	N	Mem. usage	GPU time	time/mass
lung	34108	11.9 MB	0.254 ms	7.4 ps
arm	82105	28.5 MB	0.456 ms	5.5 ps

Note that in order to measure fair and concrete performance results, the optimizations for dynamically activated masses introduced in 4.6 have been disabled so all masses within the model are always assumed to be active. The algorithm is tested with two distinct soft tissue surgery data sets: *lung*, which is a highly complex data set containing a complete lung organ with veins, arteries, trachea, lymph nodes and a tumor obtained from three-dimensional MRI data of a patient, and *arm* which is another complex data set containing a left arm with bones, veins, arteries and muscles. The *lung* data set, illustrated in 7(a), is part of one of the VICTAR® surgery training scenarios that can be seen in 7(b). Results are illustrated in table 2.

Table 2 shows similar results for the two different data sets. For the *arm* model, the algorithm performance of *0.456 ms* alone represents a physics rate of around *2192 Hz* as measured by the profiler. When we add the overhead of synchronization with other subsystems and miscellaneous CUDA control instructions, the physics rate is sustained at around *1830 Hz* in practice, putting it well above the requirement of 1000 Hz for a data set representing a model with $N = 82105$ active masses and an implicit worst case of $N * 26 = 2134730$ springs.

This means that the system is capable of processing approximately 150250 active masses at a rate of around 1000 Hz. However, if we enable the dynamic activation optimization described in 4.6, we can sustain rates of 1000 Hz for far larger data sets. We deliberately left these results out because these data sets will initially only contain inactive masses and are thus unsuitable for consistent and fair testing. Theoretically, when excluding the overhead of iterating over inactive masses, the complexity of the data sets could be increased by multiple orders of magnitude while sustaining a 1000 Hz and up most of the time, unless of course in the unlikely situation where all of the masses become activated at once.

7. CONCLUSION

The main goal of this research was to design and implement a parallel mass-spring model with a performance requirement of running at a rate of 1000 Hz or higher. This requirement was set to allow for smooth and consistent haptic (and visual) interaction with the user, where two known soft tissue data sets were available to test this requirement.

We have designed and implemented an algorithm as a CUDA kernel that satisfies the above performance requirement. The kernel was made into an integral part of VICTAR® by working together with subsystems such as the graphics renderer and haptic devices in a multi-threaded environment. Smart buffering strategies were used to ensure that data flow between the various subsystems was kept safe from threading issues while avoiding synchronization and stalling of threads.

Finally, the combination of this novel parallel mass-spring algorithm and VICTAR® as a whole represents a software solution capable of virtual simulation of human soft tissue with haptic user interaction, serving as the base for new commercial virtual surgery training platforms.

8. FUTURE WORK

Possible future work includes investigating the scalability of the mass-spring algorithm in different ways. If we consider future GPU architectures with improved resources such as computational power and increased memory bandwidth, it remains to be seen whether the algorithm will scale linearly with these resources. At the same time, the system could perhaps also be adjusted for setups with workstations containing multiple graphics cards. The algorithm would have to be distributed over multiple available GPUs and scale accordingly.

9. ACKNOWLEDGEMENTS

This research would not have been possible without help from various important people. We would like to thank M. Weber and M. Stoelinga of the University of Twente for their supervision and Vrest Medical for providing all the necessary facilities. Special thanks goes to E.E. Kunst and A.J.B. Sanders of Vrest Medical for their continued supervision and for making this research possible.

10. REFERENCES

[1] A. Ahmad, S. Adly, O. Terraz, and D. Ghazanfarpour. Stability analysis of filtered mass-spring systems. 2007.

[2] J. E. Chadwick, D. R. Haumann, and R. E. Parent. Layered construction for deformable animated characters. In *Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '89, pages 243–252, New York, NY, USA, 1989. ACM.

[3] D. T. Chen and D. Zeltzer. Pump it up: computer animation of a biomechanically based model of muscle using the finite element method. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '92, pages 89–98, New York, NY, USA, 1992. ACM.

[4] T. S. M. C. d. Farias, M. W. S. Almeida, a. M. X. N. Teixeira, Jo V. Teichrieb, and J. Kelner. A high performance massively parallel approach for real time deformable body physics simulation. In *SBAC-PAD '08: Proceedings of the 2008 20th International Symposium on Computer Architecture and High Performance Computing*, pages 45–52, Washington, DC, USA, 2008. IEEE Computer Society.

[5] M. Garland and D. B. Kirk. Understanding throughput-oriented architectures. *Commun. ACM*, 53:58–66, November 2010.

[6] Y. Jiao, H. Lin, P. Balaji, and W. Feng. Power and performance characterization of computational kernels on the gpu. In *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, GREENCOM-CPSCOM '10, pages 221–228, Washington, DC, USA, 2010. IEEE Computer Society.

[7] J. Kim, C. Choi, S. De, and M. A. Srinivasan. Virtual surgery simulation for medical training using multi-resolution organ models. *The International Journal of Medical Robotics and Computer Assisted Surgery*, 3(2):149–158, 2007.

[8] U. Kühnapfel, B. Neisius, H. G. Krumm, C. Kuhn, and M. HÃijbner. CAD-Based Simulation And Modelling For Endoscopic Surgery, 1993.

[9] J. Mosegaard, P. Herborg, and T. S. Sørensen. A GPU accelerated spring mass system for surgical simulation. *Studies in health technology and informatics*, 111:342–348, 2005.

[10] J. Mosegaard and T. S. Sørensen. GPU Accelerated Surgical Simulators for Complex Morphology. In *VR '05: Proceedings of the 2005 IEEE Conference 2005 on Virtual Reality*, pages 147–154, 323, Washington, DC, USA, 2005. IEEE Computer Society.

[11] M. Müller, B. Heidelberger, M. Hennix, and J. Ratcliff. Position based dynamics. *J. Vis. Comun. Image Represent.*, 18(2):109–118, 2007.

[12] L. P. Nedel and D. Thalmann. Real Time Muscle Deformations using Mass-Spring Systems. In *Proceedings of the Computer Graphics International 1998*, CGI '98, pages 156–, Washington, DC, USA, 1998. IEEE Computer Society.

[13] NVIDIA. *NVIDIA CUDA Programming Guide 4.0*. 2011.

[14] X. Provot. Deformation constraints in a mass-spring model to describe rigid cloth behavior. In *IN GRAPHICS INTERFACE*, pages 147–154, 1995.

[15] A. Rasmusson, J. Mosegaard, and T. S. Sørensen. Exploring Parallel Algorithms for Volumetric Mass-Spring-Damper Models in CUDA. pages 49–58. 2008.

[16] D. Terzopoulos, J. Platt, A. Barr, and K. Fleischer. Elastically deformable models. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '87, pages 205–214, New York, NY, USA, 1987. ACM.