Harnessing the GPU for Real-Time Haptic Tissue Simulation

C. E. Etheredge
University of Twente

E. E. Kunst
Vrest Medical

A. J. B. Sanders
Vrest Medical

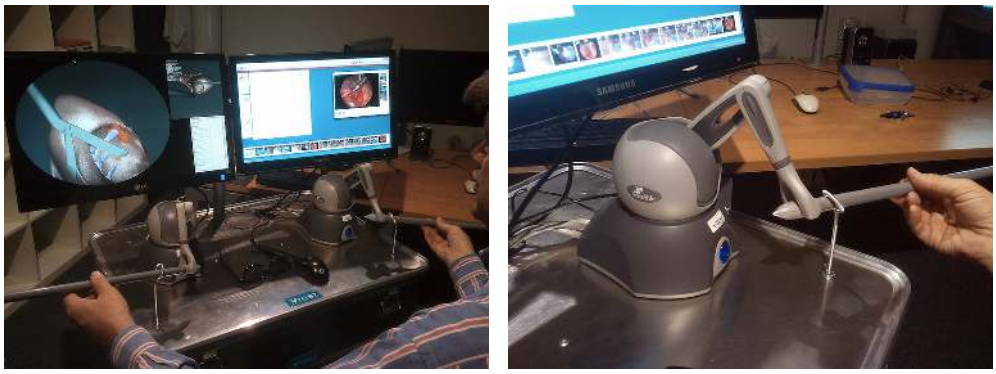

Figure 1. Left: Typical VICTAR setup in action, running our implementation with two haptic devices providing model interaction. Right: SensAble PHANTOM Omni haptic device with a custom tool extension.

Abstract

Virtual surgery simulators are emerging as a training method for medical specialists and are expected to provide a virtual environment that is realistic and responsive enough to be able to physically simulate a wide variety of medical scenarios. Haptic interaction with the environment requires an underlying physical model that is dynamic, deformable, and computable in real-time at very high frame rates.

By harnessing the GPU, we are able to simulate an environment with soft volumetric tissue that supports real-time deformation and two-way haptic interaction. In particular, we present a parallel algorithm that uses a volumetric mass-spring model to simulate this environment, implemented using NVIDIA CUDA. Our algorithm is implemented and used as an integral part of *Virtual Competence Training Area* (VICTAR), an extendable virtual surgery simulation software framework by Vrest Medical. We show that our method is capable of simulating a model with over 100 K masses at 1000 Hz on NVIDIA Tesla C2050. We also discuss the scalability and potential future applications of the algorithm.

1. Introduction

Virtual surgery simulators are expected to create a virtual environment containing a representation of human anatomy (e.g., a patient) and supporting an extensive degree of user interaction. For a medical training scenario that is simulated, the environment must be accurate enough in order to be used for extensive practice by medical specialists.

In our opinion, the ideal virtual surgery simulator accepts user input in the form of a tactile sensor, such as a haptic feedback device, which represents a controllable virtual surgical tool within the simulator. It has virtual tools available that range from simple probes to scalpels and that are able to interact and physically deform the virtual environment. Output of the system is provided visually and through the haptic feedback device by means of forces that correspond to the calculated forces acting on the virtual tool. The user can feel varying degrees of physical resistance, depending on the type of tissue or tool, as the virtual tool collides with the environment. Haptic perception of the virtual environment and corresponding visual cues in response to the interaction of the tool are important. They serve as the main factor in the simulator's perceived realism and are expected to face close scrutiny from field experts.

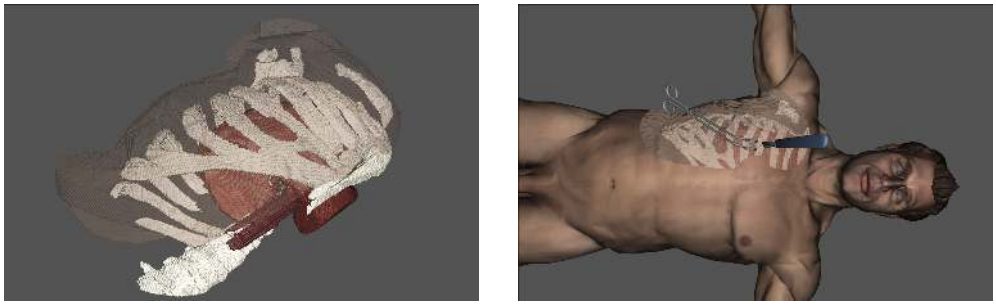

Figure 2. Practical examples of a volumetric lung model as part of a surgery training scenario in VICTAR.

In order to translate the above idea of the ideal simulator into practice, the virtual environment has to be represented by an underlying dynamic physical model that supports this kind of interaction, while its computation must be fast enough to support real-time I/O with haptic feedback devices. By using a volumetric model that defines the surface, as well as the internal structure of any simulated object, realistic interaction can be achieved.

Physical models with these characteristics can be found in the field of soft body dynamics, and there exists a variety of different models with accompanying algorithms. Our research focuses on the volumetric mass-spring model, where the volumes of objects are modeled as a set of point masses and interconnected by many elastic springs that follow general physical laws. The theory behind the model is

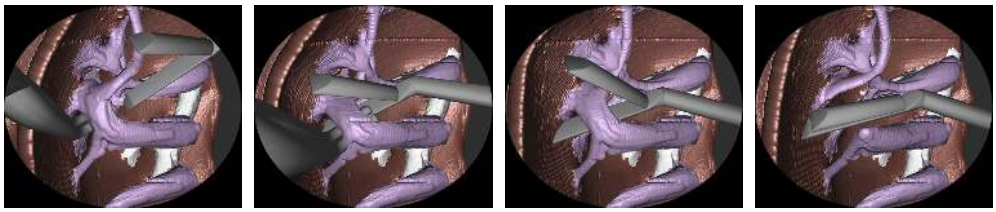

Figure 3. Example of the four steps involved in endoscopy training with VICTAR where an artery is pulled and subsequently cut. The model shown is rendered by means of sphere particles with deferred bilateral filtering.

explained in Section 3. It essentially uses a straightforward algorithm that does not require any complex mathematical operations. In practice, however, the serial implementation is far from being fast enough due to the large amount of masses and springs involved in volumetric models.

On the one hand, 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 the medical training scenario. One such example is given in Figure 3. On the other hand, the simulation must run at a fast-enough rate to support real-time I/O with haptic feedback devices and allow smooth haptic interaction. These haptic devices contain actuators that apply a set of forces to the user's hand and generally require a force to be streamed real-time at a minimum rate of 1000 Hz, with lower data streaming rates resulting in uncontrollable oscillation and irregular actuation. As a consequence, the simulation must be capable of processing haptic input, calculating a new state of the physical model given the input, and returning corresponding haptic output all within a maximum time span of 1 ms (1000 Hz).

In summary, this paper provides a parallel CUDA algorithm that computes a volumetric mass-spring model at a real-time haptic interaction rate and describes how it is implemented as part of the VICTAR surgery simulation framework, which is currently being developed by Vrest Medical. This is an extendable software framework designed for virtual surgical training, featuring haptic device handling and a scripting engine that allows for easy prototyping of various simulation scenarios. The work described in this paper is made in cooperation with Vrest Medical and is ultimately used to provide physics interaction as part of the aforementioned product.

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, Terzopoulos et al.'s paper [1987] was the first to incorporate physical properties into a graphical object, creating volumetric and elastically deformable models 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 papers by Chadwick et al. [1989] and Chen et al. [1992] for animating muscle deformations in anatomically based characters.

FEM however comes at a high computational expense. For a more balanced trade-off between accuracy and (interactive) performance, we adopted the mass-spring model, which is an approximated model where the environment or object is subdivided into discrete masses that are interconnected by springs with attributes based upon the physical properties of the object. Though the model may be more limited in accuracy, it has shown to have a relatively good performance and is therefore already widely used in various areas, such as virtual surgery [Kühnapfel et al. 1993], rigid cloth simulation [Provot 1996], muscle deformation [Nedel and Thalmann 1998], and others.

Research into more realistic virtual surgery simulation is ongoing, and improvements have been made resulting in an increasing complexity of the environment as well as enabling better haptic interaction with the model. These two factors play an important role in improving the perceived realism, but they also impose severe limits on the computation time and require significant efforts in improving the performance of the underlying physical model [Kim et al. 2007].

Previously, deformable objects were successfully implemented on the GPU using different techniques. Research by Shi et al. [2008] presents a simulation model for 2D cloth with haptic rendering implemented in CUDA that is considerably faster than a reference CPU implementation. Although the research provides insight into FEM-based simulation models, the model is only 2D and not volumetric, and haptic rendering is still performed on the CPU at a rate far lower than the ideal rate, and therefore only has limited practical use.

In papers by Mosegaard et al. [2005][2005] and Müller et al. [2007], parallel GPGPU-based approaches for mass-spring models are presented with significant speedups when compared to reference CPU implementations. In more recent work by Rasmusson et al. [2008] and Farias et al. [2008], different CUDA implementations of the mass-spring model for surgery simulation are proposed and evaluated, showing potential speedups compared to earlier GPU-based approaches. Unfortunately, none of the papers implement support for two-way haptic interaction with the mass-spring model.

The lack of two-way haptic interaction with the simulated model is a common problem. Being able to properly feel the surface of a model through a haptic device requires a constant high-frequency (e.g., 1000 Hz) output of forces from the simulation model into the haptic device. As in the previous papers, the problem of providing a proper high-frequency force output is often intentionally left open, because it typically implies that the model must be simulated with very high performance.

However, research by Courtecuisse [2010] and Zhang [2010] goes into further detail on the use of two-way haptic interaction. The two papers implement similar FEM-based models with complex topology in order to simulate soft-tissue deformation and (haptic) needle insertion. As mentioned before, these FEM models have the potential to provide very accurate and stable model simulation, but at the cost of significant performance decrease. In both cases, the high haptic rate of 1000 Hz is acknowledged, but the final simulation performance that is presented remains far below this rate, at 25 Hz and 40 Hz. These output forces are then upsampled to the haptic rate using a clever multi-rate technique or interpolation filter. As a consequence, the high-frequency characteristics of the force output (and thus haptic interaction) are lost. Practical use is therefore limited, since the high-frequency characteristic allows the user to perceive (feel) clear and detailed distinctions between smooth and non-smooth or highly frictional model surfaces, as well as oscillation in the model (such as a heartbeat), which are essential for virtual surgery.

In a nutshell, this research instead focuses on achieving model simulation performance that allows for true high-frequency force output, making practical use of two-way haptic interaction and virtual surgery possible.

3. Background

3.1. Mass-spring Model

The basic fundamentals of the mass-spring algorithm are not very difficult. A mass-spring model consists of a set of mass points M_i with $i = [0, N-1]$ where N represents the number of total masses. The model is interconnected with springs S_{ij} connecting any two arbitrary mass points M_i and M_j.

Figure 4 represents the simple one-dimensional case of a mass-spring model [Ahmad et al. 2007]. 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:

$$m_i \vec{a}_i = \vec{F}_i, \tag{1}$$

where m_i is the mass, $\vec{a}_i$ is the acceleration, and $\vec{F}_i$ is the total force on mass M_i.

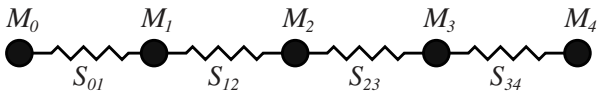

Figure 4. A one-dimensional case of a mass-spring system.

The total spring force $\vec{F}_i$ is determined by summing all forces exerted by springs:

$$\vec{F}_i = \sum_{j \in S_i} \vec{F}_{ij}, \tag{2}$$

where S_i is the set of connected masses for mass M_i. The variable $\vec{F}_{ij}$ is the force exerted by the spring connecting the two masses M_i and M_j that is obtained from Hooke's law:

$$\vec{F}_{ij} = k_{ij}(y_{ij} - y^0_{ij})\frac{y_{ij}}{|y_{ij}|}, \tag{3}$$

where k_{ij} is the spring stiffness, y_{ij} is the current spring length or distance between masses M_i and M_j, and y^0_{ij} is the spring rest-length in equilibrium.

3.2. Integration Methods

In order for the mass-spring system to exhibit movement, it is necessary to translate these forces into positions. Considering Equation (1), a naive method to derive a position is to take the acceleration $\vec{a}_i$ and assume a non-existing velocity:

$$x_i(t + \Delta t) = x_i(t) + \frac{1}{2}\vec{a}_i \Delta t^2, \tag{4}$$

where t is the time, Δt is the fixed time step, and $\vec{a}_i$ is acceleration of mass M_i. The acceleration $\vec{a}_i$ is determined from the total force $\vec{F}_i$ on mass M_i by using Equation (1). Obviously, a formula like this is incomplete as it does not take any velocity in the system into account.

To obtain a set of useful results instead, a numerical integration method can be used to approximate the velocities and positions in the system based on current and previous data. Obtained accuracy directly corresponds to the order of the numerical integration method, where lower-order methods yield more error but also decrease complexity. Examples of such methods are Euler (first-order), Verlet (second-order) and Runge–Kutta (e.g., fourth-order). First-order explicit Euler integration is defined as follows:

$$x_i(t + \Delta t) = x_i(t) + \vec{v}_i \Delta t,$$
$$v_i(t + \Delta t) = v_i(t) + \vec{a}_i \Delta t,$$

where $\vec{v}_i$ is the velocity of mass M_i. When looking at the Euler method, it can be seen that the velocity term always lags one step behind. When dealing with mass-spring systems, this can potentially lead to springs overshooting their positions resulting in unwanted oscillation of the system. For a better balance between error and complexity, Verlet integration in second-order is a better choice for our mass-spring system:

$$x_i(t + \Delta t) = 2x_i(t) - x_i(t - \Delta t) + \frac{1}{2}\vec{a}_i \Delta t^2. \tag{5}$$

Albeit slightly more complex than Euler, the two last positions of the mass M_i are used to determine velocity (saving performance), while the error of the outcome stays second-order. It is possible to reduce the error even further by using the higher-order Runge-Kutta integration method, though at the cost of significantly more memory space and reduced performance [Zhang et al. 2010]. Runge-Kutta is generally considered to be a good choice when accuracy is preferred above performance. Since performance is our foremost concern, we opt to use Verlet integration instead.

Additionally, other factors such as universal gravity can be added by further extending the Verlet equation with additional forces:

$$m_i\vec{a}_i = \vec{F}_i + m_i\vec{g}, \tag{6}$$

where $\vec{g}$ is the universal gravity vector. For simplicity, the variable m_i is further omitted by assuming that $m_i = 1$.

3.2.1. Limitations

To make the results of the Verlet (and similar) integration methods as exact as possible, two important criteria have to be met: ideally, the acceleration vector $\vec{a}_i$ and the time step Δt have to be kept constant.

It is not difficult to see that keeping the acceleration constant is virtually impossible, considering that the integration is applied to a mass-spring model. This imposes a considerable limitation on the integration's output-precision, which in turn poses restrictions on the acceleration vector and implies various upper bounds for any variables that affect the acceleration. This practically means that variables such as k_{ij} (spring stiffness) must have a limited range (depending on the time step), to prevent catastrophic structural failure of the model.

As mentioned before, the time step is also to be kept constant. It may prove to be difficult to control the time difference between iterations, and given that frame rate in software is not very easy to control, it is possible to pass in a constant time step anyway. Then, regardless of the frame rate at which the software runs, the calculation of the model advances by a constant time step every frame. The downside of this approach is that the model does not advance at constant speed in relation to the software, e.g., it will seem to advance slower when the frame rate drops considerably.

In practice, using a time step that approximates the software's (ideal) update rate provides sufficient model stability, assuming a single integration iteration-step per frame. Note that the ideal update rate and, thus, time step has to be "small enough" to ensure a stable model. We find that a rate equal to the haptic update rate (1000 Hz) with a single iteration-step per frame is sufficient.

4. Mass-spring Algorithm

The mass-spring algorithm calculates all variables within the model by using the physics equations of Section 3.1. Each time the algorithm is invoked on the model, the result is a new physical state of the model containing the masses and springs with new positions and other variables adjusted according to all applied forces (e.g., by springs, or by interaction with a virtual tool) within the predefined time step 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 shown in Figure 5.

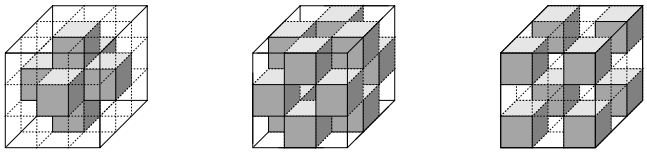

Figure 5. Three-dimensional uniform grid layout connecting the center mass to at most 26 neighboring masses (in grey) at Euclidean distance (a) 1, (b) $\sqrt{2}$, and (c) $\sqrt{3}$. (Image courtesy of [2008].)

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 neighboring masses, forming springs, all of which are within a $3 \times 3 \times 3$ 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 neighboring masses M_j forming the spring. Because of the discrete grid layout, this distance is either 1, $\sqrt{2}$, or $\sqrt{3}$. For every mass, we can define at most $3*3*3-1=26$ neighboring masses (and thus springs); however, for example, masses at the outer surface of the grid will have fewer neighbors.

4.2. Model Limitations

The physically approximative nature of the mass-spring model introduces practical limitations in the behavior of the model. Partly due to the uniform grid structure of the model, the most significant limitation is spring inversion: a situation where a mass, in (normal) equilibrium, is displaced beyond a certain point until a new equilibrium is formed where one or more springs now have a direction that is inverse to the earlier equilibrium. This problem manifests itself as irreversible "collapse" or "tangling" of masses, especially at the surface of the volumetric model and becomes very apparent with frequent mass displacement (e.g., by means of collision tools).

To alleviate the issue of spring inversion, different workarounds with varying degrees of effectiveness are available:

Shape-preserving springs
This solution proposed by Choi et al. [2005] introduces a shape-preserving spring that reduces the collapsing by restoring the model to the original shape. It is a zero-length spring, located at the initial position of the corresponding mass and prevents excessive displacement by exerting a force that pushes the mass back to its initial position. This is an elegant solution, but less effective for models that are to be pushed around and manipulated, as is often the case in virtual surgery.

Torsion springs
By rewriting Hooke's law in Equation (3) in angular form, it is possible to simulate a spring that exerts a force proportional to an angle instead of a length. The result is a torsion spring (e.g., as used in mousetraps), and it can be added in between every connected mass in the model. Shearing of the model can now be resisted by the torsion springs and, as a result, spring inversion is further reduced.

Increased surface stiffness
As spring inversion occurs mostly at the surface of the model due to the interaction with virtual tools, the stiffness of the springs that form (or are connected to) the surface of the model is doubled. As stiffer springs are harder to invert, this typically reduces the occurrence of the problem without modifying the majority of the model.

Increased model resolution
While increasing the resolution of the model itself does not solve the issue of spring inversion, it does make its effect significantly less apparent. As more masses are available and their springs are inherently smaller and stiffer, it becomes more difficult to push the well-connected masses beyond their equilibrium.

As our proposed design allows for ever-increasing model resolution, we find that the two last options are the most straightforward and non-invasive workarounds.

4.3. Collision Handling

As mentioned in previous sections, the mass-spring model must be capable of being manipulated by external factors such as a virtual surgical tool. The position and orientation of this virtual tool is typically controlled by corresponding input from a haptic

feedback device, and collision handling is used to exert forces on any masses where the tool touches the model.

As the collision handling is very straightforward, we only consider the simple case of a probe tool (e.g., a hand) that is capable of pushing masses away. A capsule (or sphere) with predefined geometry is placed at the exact virtual position of the haptic device, and any masses within the radius of the shape are placed on the closest surface point of the geometry, in a constraint-based manner. Equally important, all masses that are displaced as a result of this collision interaction are marked and their total spring force $\vec{F}_i$ is summed and provided as output to the haptic feedback device, completing the haptic feedback cycle.

4.4. Straightforward Serial Algorithm

The most straightforward variant 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** *springs* S_{ij} **do**
2: **get** mass positions $\vec{x}_j(t)$ and $\vec{x}_i(t)$
3: determine spring length $\{y_{ij} \leftarrow \vec{x}_j(t) - \vec{x}_i(t)\}$
4: calculate spring force $\{\vec{F}_{ij} \leftarrow k_{ij}(y_{ij} - y_{ij}^0)\frac{y_{ij}}{|y_{ij}|}\}$
5: calculate spring force on M_i $\{\vec{F}_i = \vec{F}_{ij}\}$
6: calculate spring force on M_j $\{\vec{F}_j = \vec{F}_{ji} = -\vec{F}_{ij}\}$
7: update $\sum \vec{F}_i$ and $\sum \vec{F}_j$
8: **end for**
9: **for all** *masses* M_i **do**
10: $\vec{F}_i \leftarrow \sum \vec{F}_i$
11: integrate $\{\vec{x}_i'(t+\Delta t) \leftarrow 2\vec{x}_i(t) - \vec{x}_i''(t-\Delta t) + \frac{1}{2}\vec{F}_i\Delta t^2\}$
12: **set** mass position $\vec{x}_i'(t+\Delta t)$
13: **end for**

For each mass M_i, forces exerted by the connected springs are first accumulated in $\sum \vec{F}_i$. Given the total force $\vec{F}_i$ and using Equations (5) and (6), Verlet integration is used to determine the new position x_i of mass M_i.

4.5. Initial Parallel Algorithm

The first trivial step towards parallelizing Algorithm 1 is to process the existing for-loops in parallel, instead of doing sequential iterations. When each of the iterations is processed by a different thread in parallel, the accumulation of variables $\sum \vec{F}_i$ and

$\sum \vec{F}_j$ will, however, lead to race conditions due to different threads trying to read and write the variables 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 Initial parallel implementation.

1: **for all** *masses* M_i **do**
2: **get** mass position $\vec{x}_i(t)$
3: **for all** *neighbouring masses* M_j **do**
4: **get** mass position $\vec{x}_j(t)$
5: determine spring length $\{y_{ij} \leftarrow \vec{x}_j(t) - \vec{x}_i(t)\}$
6: calculate spring force $\{\vec{F}_{ij} \leftarrow k_{ij}(y_{ij} - y^0_{ij})\frac{y_{ij}}{|y_{ij}|}\}$
7: calculate spring force on M_i $\{\vec{F}_i \leftarrow \vec{F}_{ij}\}$
8: update $\sum \vec{F}_i$
9: **end for**
10: $\vec{F}_i \leftarrow \sum \vec{F}_i$
11: integrate $\{\vec{x}_i'(t+\Delta t) \leftarrow 2\vec{x}_i(t) - \vec{x}_i''(t-\Delta t) + \frac{1}{2}\vec{F}_i\Delta t^2\}$
12: **set** mass position $\vec{x}_i'(t+\Delta t)$
13: **end for**

In Algorithm 2, there is an iteration over every mass M_i instead of every spring S_{ij} as in the previous algorithm. The combination of a particular mass M_i and any of its neighboring masses M_j represents the spring S_{ij}. Since every possible combination is iterated, all springs in the model are guaranteed to be iterated. The for-loop is parallelized, and race conditions are eliminated because the only data that is updated ($\vec{x}_i'(t+\Delta t)$) is unique to every iteration and not accessed by any other thread within the same time step.

Note that the algorithm has a number of implicit memory accesses, e.g., the list of neighboring masses at line 3 and variables k_{ij} and y^0_{ij} at line 6, which will be further explained in the next section.

4.6. Extended CUDA Implementation

In this section, we extend the initial parallel algorithm from the previous section to target the CUDA platform. Algorithm 3 shows the extended algorithm with all memory access transactions explicitly stated, containing predictable coalesced transactions as well as unpredictable random transactions.

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: **for** $k = 0 \rightarrow 26$ **do**
6: **get** neighbour mass at k $\{M_j \leftarrow N_i(k)\}$
7: **if** M_j exists **then**
8: **get** mass properties p_j
9: **get** mass position $\vec{x}_j(t)$
10: **get** spring properties k_{ij} and y_{ij}^0
11: determine spring length $\{y_{ij} \leftarrow \vec{x}_j(t) - \vec{x}_i(t)\}$
12: calculate spring force $\{\vec{F}_{ij} \leftarrow k_{ij}(y_{ij} - y_{ij}^0)\frac{y_{ij}}{|y_{ij}|}\}$
13: calculate spring force on M_i $\{\vec{F}_i \leftarrow \vec{F}_{ij}\}$
14: update $\sum \vec{F}_i$
15: **end if**
16: **end for**
17: **get** mass position $\vec{x}_i''(t - \Delta t)$
18: synchronize threads
19: perform collision handling
20: $\vec{F}_i \leftarrow \sum \vec{F}_i$
21: integrate $\{\vec{x}_i'(t + \Delta t) \leftarrow 2\vec{x}_i(t) - \vec{x}_i''(t - \Delta t) + \frac{1}{2}\vec{F}_i \Delta t^2\}$
22: **set** mass position $\vec{x}_i'(t + \Delta t)$
23: **end for**

Every iteration runs in a separate thread and requires multiple memory access transactions with different memory access characteristics. At line 2, variable $\vec{x}_i(t)$ is retrieved by thread M_i. As index i is always known in advance, the memory transactions for variables $\vec{x}_i(t)$, p_i, $\vec{x}_i''(t - \Delta t)$, and $\vec{x}_i'(t + \Delta t)$ at respectively, lines 2, 3, 17, and 22 are thread-unique and can be predicted and optimized for data coalescing by CUDA. By using thread synchronization at lines 4 and 18, these memory transactions—performed by threads in parallel—are coalesced into single transactions by CUDA, saving considerable memory bandwidth. Transactions for variables p_j and $\vec{x}_j(t)$ at lines 8 and 9 are costly random memory accesses as index j can represent any arbitrary neighboring mass connected to M_i.

At line 6 function $N_i(k)$ retrieves the index j for the neighboring 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. At line 19, the algorithm performs the necessary collision handling for haptic interaction, as further explained in Section 5.1.

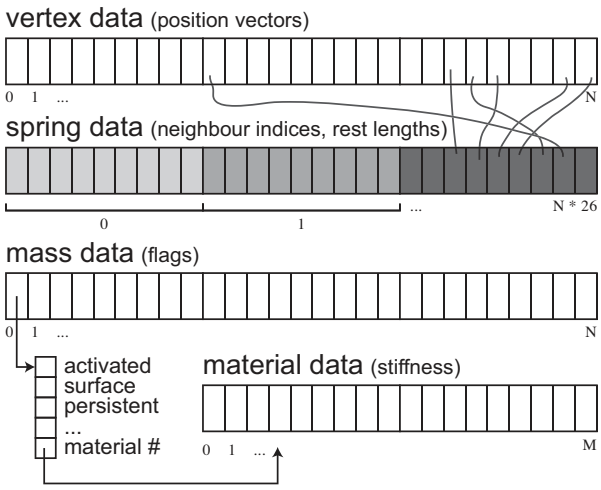

Figure 6. Layout overview of aligned data arrays, from top to bottom: *vertex data*, *spring data*, *mass data* and, *material data*.

4.7. Data Structure

Data in memory that is accessed by predictable transactions should be structured in such a way that allows for coalescence optimization. As a general rule for CUDA, data elements should preferably be aligned to 32-bit, 64-bit, or 128-bit words so that multiple parallel transactions at sequential addresses are coalesced into single transactions.

The data structures used in the CUDA algorithm are organized as simple one-dimensional arrays of aligned data. Figure 6 gives an overview of how these structures are positioned in memory. The *vertex data* array contains the new and old positions of $\vec{x}$ as accessed by Algorithm 3 at lines 2, 9, 17, and 22. The *spring data* array contains the indices for $N_i(k)$ at line 6 as well as unique spring properties such as y_{ij}^0 at line 10. The *mass data* array contains the mass properties p_j at line 8.

4.8. Additional Optimization

Since the number of springs (at most $N \times 26$) quickly increases with more complex models, additional optimizations are added to decrease memory resources and bandwidth. Since spring properties, such as stiffness k_{ij}, are often similar in local regions of springs, we reduce memory resources by classifying every spring S_{ij} by a predefined mass material $Mat(l)$, where $l = p_i.material$ is defined by mass M_i. Spring properties, such as k_{ij}, are therefore defined by material $Mat(l)$, and, thus, implicitly shared with all springs connected to mass M_i as well as all other springs sharing the same material. The *material data* array in Figure 6 contains the predefined spring properties defined for every material $Mat(l)$ where $l = [0, L-1]$, while the mate-

rial index $p_i.material$ for every mass M_i is encoded in the *mass data* array. Since a model usually only contains a few materials (e.g., $L = 16$, with the encoded index $p_i.material$ only requiring 4 bits), considerable memory resources are saved: now only L, instead of at most $N * 26$ spring properties are required, excluding the overhead of the encoded index $p_i.material$.

Total computation time can be even further reduced by skipping the computation of masses with negligible movement. If the total force $|\vec{F}_i|$ of a mass M_i does not exceed a predefined threshold ε, its force calculations are skipped and its position $\vec{x}_i$ is left unchanged, preventing time and resources being spent on masses with negligible impact on the model.

5. System Design

Given the extended CUDA implementation of the algorithm in the previous section, we now have the means to accurately calculate the physical state of the model at given time t. At this point, our algorithm is implemented as an integral part of the existing VICTAR surgery simulation framework. Figure 7 shows the multi-threaded system diagram of VICTAR containing all of the relevant subsystems in their own threads and contexts. To properly integrate the CUDA mass-spring model algorithm, it is necessary to employ a thread-safe, but fast data synchronization strategy to ensure that the model data is kept up-to-date and synchronized in all subsystems. The figure provides a diagram of the complete data flow for the system and also describes whether the data is residing on the CPU or GPU.

User input is provided by a haptic device representing a virtual tool for manipulating and deforming the mass-spring physics model. Any subsequent force response from the physics model acting on the virtual tool is then delivered back to the device as haptic feedback output, providing the user with full two-way haptic interaction with the model.

The visual representation of the model is generated by the system's graphics subsystem, implemented using Microsoft DirectX 10 and the shader capabilities of Shader Model 4.0. A specialized volumetric multi-pass shader-rendering technique is used to render the visual representation of the mass-spring model. While details of the renderer fall beyond the scope of this paper, the synchronization between the graphics subsystem and CUDA is especially important, as will be explained later in this section. Needless to say, while we have chosen to use DirectX, alternative graphics API's such as OpenGL are also an option.

5.1. Haptic Device Interaction

The framework implements two-way haptic interaction by means of a haptic device and the SensAble OpenHaptics programming interface. The device has tactile sensors

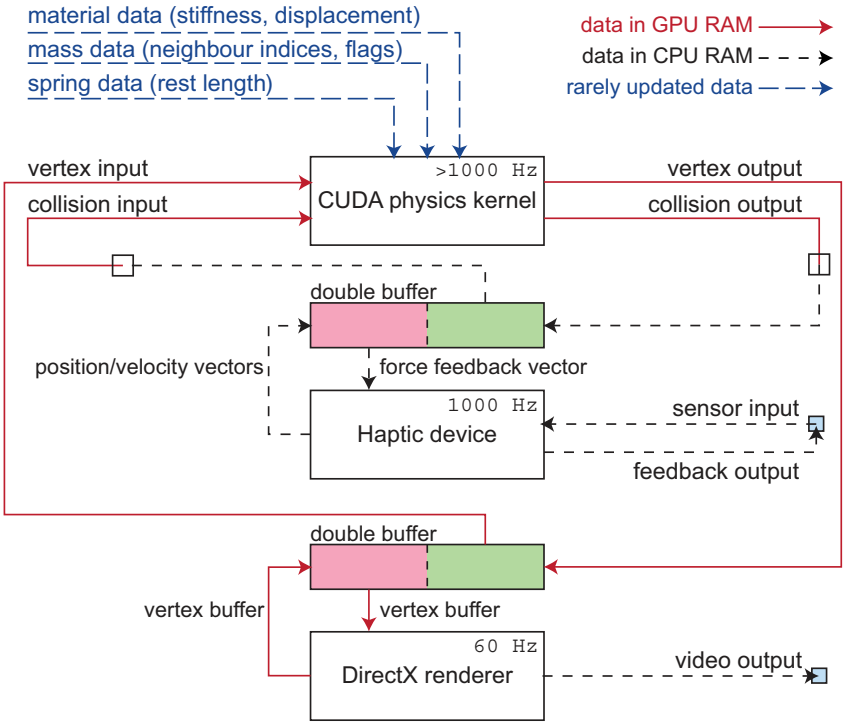

Figure 7. Detailed overview of data flow within the system as implemented by VICTAR consisting of various subsystems (and threads) as defined by the white boxes. Squares represent the memory copies required to move the data between RAM regions in CPU and GPU. A variant of double buffering is used for data synchronization.

in six degrees of freedom to register the current stylus orientation and velocity while built-in actuators provide the ability to apply force feedback to the device in three degrees of freedom. It is controlled through a real-time priority haptic device I/O thread scheduled to run at a constant rate of 1000 Hz in which the sensors are read out and an actuator force is applied. The minimum I/O streaming rate of 1000 Hz is necessary, as lower rates will result in uncontrollable oscillation and irregularities in the actuators that may cause a negative user experience.

In Figure 7, the vector representing the position of the virtual tool is acquired from the device sensors in the haptic device I/O thread and is immediately transferred to the CUDA host thread, where it is subsequently copied into GPU RAM where it is eventually read by the CUDA physics kernel. At the same time, another vector containing the force on the virtual tool is calculated by the kernel and copied back to CPU RAM, where it is immediately transferred to the haptic device I/O thread and written to the device actuators. To prevent read-while-write situations, where either of the threads is reading a value that is currently being written to by the other thread, locking can normally be applied. However, in this case, locking would introduce

unacceptable execution stalling in either of the real-time priority threads. Instead, we employ a variant of the *double buffering* strategy usually found in the field of computer graphics. Our case consists of a single-reader single-writer situation where 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 are flipped so that the reader is able to read the updated data in B while the writer can overwrite outdated data in A. This strategy requires very little synchronization and lowers execution stalling in the CUDA thread hosting the mass-spring algorithm.

Both vectors can now be quickly synchronized in a thread-safe manner. The CUDA physics kernel uses the virtual tool position vector to perform collision detection and deformation of the model as it interacts with the model, generating a force by summing the spring forces $\vec{F}_{ij}$ acting on the masses that are currently within the bounding geometry of the virtual tool, as explained in Section 4.3. The user is therefore able to interact with the mass-spring model by feeling the resistance of the model acting on the virtual tool as it touches the surface.

5.2. Graphics Rendering

Visual output is accomplished by close cooperation of the CUDA host thread and the DirectX graphics renderer. The renderer takes a *vertex data buffer* as input and presents this buffer to the GPU through DirectX, where it is rendered onto the monitor. Without going into too much detail, specialized HLSL shaders for geometry processing and filtering are used to make sense of the volumetric vertex data (mass positions) in combination with the available data of neighboring masses to turn the model into a visual representation that can be rendered on the screen. 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 (and an accompanying neighbor data structure) that is identical to the structure used by the CUDA physics kernel. The vertex data structure is simply an aligned one-dimensional array of floating-point vectors of size N. As both CUDA and DirectX will only access data that is residing on the GPU, the vertex data never has to leave the GPU by using the CUDA graphics interoperability API, saving unnecessary memory copies between GPU and CPU.

5.2.1. Single Device

Up to this point, it can be assumed that the CUDA physics kernel and graphics renderer are executed on the same GPU device. In this single device configuration, it is important to notice that the renderer and physics kernel typically run at vastly different rates and in different threads. The CUDA DirectX interoperability API ensures that all buffers that are to be mapped in CUDA are thread-safe, and thus free from data corruption. This is done by stalling execution while waiting for buffers that are

already in use by DirectX and, subsequently, holding a lock while the buffers are mapped and used by any CUDA kernels.

As a consequence, the vertex data structure is locked by both the DirectX thread (during rendering) and CUDA thead (during physics calculation) and, given the different rates of these threads, frequent stalling may occur. As frequent stalling will incur a serious performance penalty, we apply the double buffering strategy as described in Section 5.1 and Figure 7 on the vertex data structure.

5.2.2. Multi-device

In a more ideal situation, the physics kernel is allowed to run exclusively on a dedicated GPU device, unhampered by any other (graphical) GPU activity and any resulting locking. This multi-device configuration consists of at least two GPU devices, one of which functions as the primary DirectX display device, while the other is used exclusively to run the CUDA physics kernel.

Needless to say, it is still necessary to provide some sort of synchronization between the GPU device running the CUDA physics kernel and the primary GPU graphics device; this presents a challenge similar to that in the single device configuration. Again, the double buffering strategy can be used to avoid unnecessary locks and stalls.

6. Results and Discussion

We have implemented the parallel mass-spring model as described in the previous sections as part of the VICTAR surgery simulator framework. More specifically, the physics engine of VICTAR is now composed of our mass-spring model implementation with the required buffering and is able to interface with one or more haptic devices and an existing graphics renderer.

In order to test our implementation within the framework, we use a minimal general-purpose scripted scenario that generates a simple cubical mass-spring model at increasing resolutions and interfaces the model with a single haptic device. A single virtual probe tool is added to verify the model's haptic response. The increasing resolution of the model will provide a set of measurements at increasing algorithmic workload and will allow us to observe and quantify effects such as CPU overhead.

Our proposed system design allows for two configurations, namely single and multi-device, with two buffering strategies—single and double buffering. Note that single buffering merely consists of a single buffer that is locked during either a read or a write. To investigate the advantage of double buffering, all combinations of configuration and buffering are implemented and tested with the above models.

Note that for the implementation of the multi-device synchronization, it is necessary to copy buffers (peer-to-peer) between two different GPU devices. Unfortunately, at the time of writing, CUDA does not allow peer-to-peer copies for GPU devices involved with DirectX, so it is necessary to perform two additional copies to and from

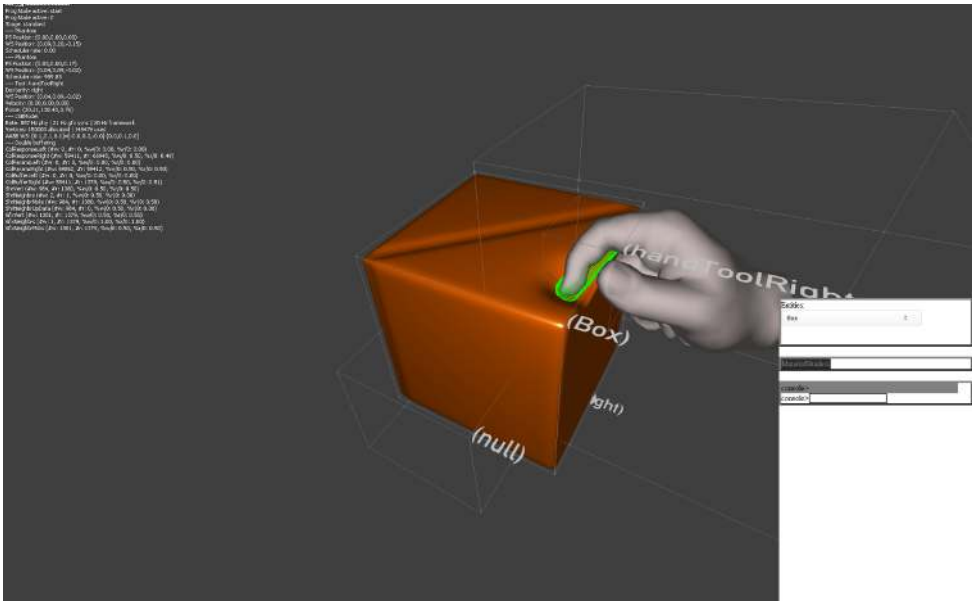

Figure 8. Simple cubical mass-spring model being tested in VICTAR with visible debugging tools and virtual haptic probe (hand).

CPU host memory. Since copies between CPU and GPU have limited memory bandwidth, this introduces an additional memory bottleneck. It is therefore not currently possible to implement and test the single buffer locking strategy in the multi-device configuration.

6.1. Test Setup

The configuration of our hardware is a workstation with an Intel Core i7-860 CPU, 6 GB of RAM with Windows 7 x64 and two NVIDIA Tesla C2050 GPU devices with Compute capability 2.0. Each of our Tesla C2050 GPU devices contains 15 streaming multiprocessors (SMs) with a maximum of 1536 threads each at 100% kernel occupancy. For the single device configuration, the non-primary graphics card is left unused. Haptic interaction is provided by SensAble PHANTOM Omni device as can be seen in Figure 1, supported by the SensAble OpenHaptics programming interface and operating at 1000 Hz. The CUDA Visual profiler is used to perform profiled runs of our implementation, providing GPU kernel timing data, register usage, and occupancy results. Our implementation uses CUDA 4.2 and is profiled by running the complete framework (with above script and scenarios) inside the profiler, allowing it to capture measured data of each iteration of the physics engine. Practical timing measurements are provided through the framework, as explained below.

Note that the CUDA Visual profiler can be set to capture a set of specific measurements and requires more separate runs of the application when more detailed infor-

mation (such as branch divergence, coalesced global memory reads and writes, etc.) is required. As our implementation is only available as part of a bigger application, and not as a stand-alone executable, it is very difficult to perform multiple profiler runs: the application behavior may differ between runs (e.g., due to varying loading times) causing issues whenever the profiler can no longer compare the results. As we are solely interested in production performance of this initial implementation, we have chosen specifically not to create a stand-alone version, and instead performed our tests with a single profiler run, capturing only the necessary performance measurements (kernel time, kernel occupancy, and register usage) thus, leaving in-depth CUDA optimization as a later exercise. These measurements represent the actual on-chip GPU performance of the implementation.

Additionally, we have added high-precision timing functionality (typically in nanoseconds) to the VICTAR framework by using the available timing APIs of the operating system (QueryPerformanceCounter). This second class of time measurements represents the actual time that passes between calls inside the framework (e.g., one iteration of the physics engine using our implementation) including all overhead such as CUDA API calls and other blocking calls and represents the real practical performance of the physics engine or of our implementation as it is integrated in the framework. These measurements give insight in the amount of (CPU) overhead, as measured on the CPU or host, that is used in the framework and outside the GPU kernel.

With regard to the model, we have chosen not to implement the force threshold optimization described in Section 4.8, in order to allow for full consistent testing where no mass is assumed to have negligible movement; thus, all possible computations within the model are implicitly performed. Furthermore, we found that the proposed two workarounds presented in Section 4.2 reduced the spring inversion problem to such an extent, that it was no longer a significant issue in our tests when performing typical surgery model interaction.

6.2. Performance Characteristics

Table 1 shows the total performance rate of the implementations for both single- and multi-device configurations and single and double buffering strategies, where the total framework performance consists of GPU kernel performance, as measured on the GPU device, and any CPU overhead, as measured on the CPU or host. The rates from this table are also visualized in Figure 9.

The most important variable to consider when looking at the performance is the total number of springs, as it dictates the amount of spring computations and is therefore closely related to the overall performance of the CUDA kernel, while the total number of masses provides better insight on the parallel scalability as will be explained in Section 6.4.

springs	masses	frequency; single device (Hz)		frequency; multi-device (Hz)	
		single buffer	double buffer	single buffer	double buffer
3614788	146476	53	183	901	902
2336252	95487	72	271	1246	1246
1638652	67477	96	362	1564	1564
1004416	41851	128	492	2137	2138
499364	21276	162	694	3102	3103
395772	17000	175	771	3167	3167
349844	15096	184	805	3205	3205
268944	11726	192	943	3840	3838
251700	11000	195	1012	4057	4057
233660	10248	196	1025	4064	4063
202068	8876	211	1043	4089	4090
146588	6569	220	1151	4584	4584
102660	4688	222	1164	4574	4574
54282	2575	223	1272	4584	4585
24024	1210	227	1348	4853	4853
5142	307	235	1382	5000	5000

Table 1. Total framework performance rate for implementations with single and multi device configurations and single and double buffering, tested on models with increasing resolution in terms of springs and masses.

It is evident that the multi-device configuration provides the fastest available implementation. Allowing the CUDA physics kernel to run on its own dedicated GPU device, where it is unhampered by any other GPU activity, significantly increases the overall practical performance. This can be seen in Figure 9, where the multi-device implementation performs four to eight times faster than the fastest single-device implementation.

Differences between single and double buffering on the multi-device configuration are virtually non-existent, due to the fact that peer-to-peer copies are currently not supported by CUDA. This makes it impossible to implement a proper single-buffer implementation with locking, as a second CPU host buffer is always present to act as a bridge between the two GPU devices and effectively cancels out any effects of single buffering. Both cases for multi-device are therefore combined into a single set of measurements.

The advantage of double buffering (over single buffering) on a single device is immediately clear when looking at the figures. The significantly lower performance of single buffering is found to be caused by internal command queues and resulting "greedy" locking in DirectX. In this case, performance is hampered by execution stalls triggered by the CUDA interoperability API waiting for the release of buffer

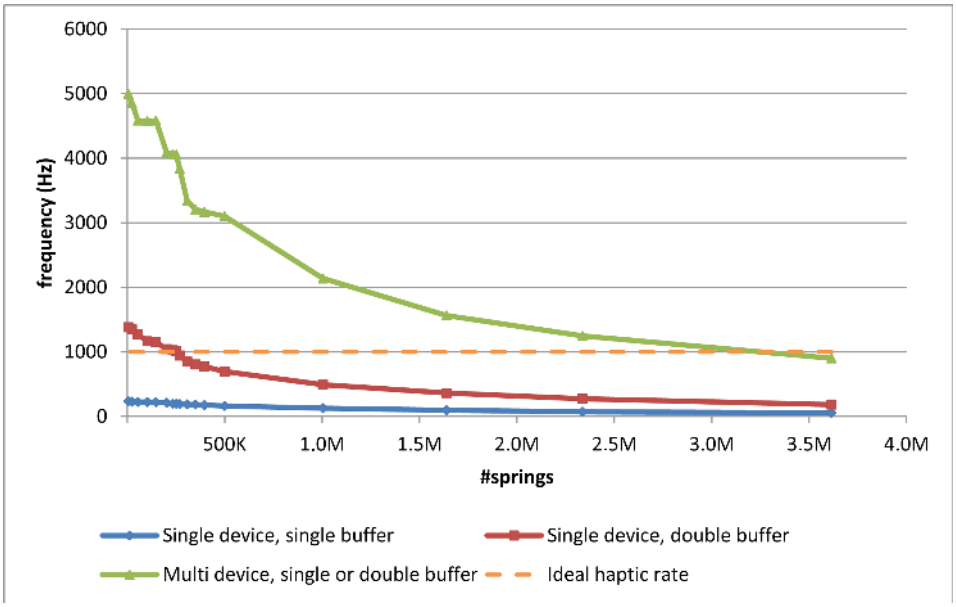

Figure 9. Framework frequency for all tested implementations.

locks by DirectX. In other words, DirectX keeps a lock on any buffer used in the graphics pipeline until all render commands involving that buffer (which have been internally queued up) have been processed. As a result, it is very difficult to control the locking, and locks will be held far longer than necessary, typically until a command buffer flush or back-buffer swap occurs. Double buffering proves to be an effective solution to this issue.

6.3. GPU Overhead

Table 2 and Figure 10 show the detailed performance results of the multi-device configuration with double buffering, currently the fastest implementation of our algorithm. The total framework time, or practical performance of our implementation inside the software, is the sum of the GPU kernel time and any time caused by overhead on the CPU. This overhead primarily consists of delays induced by other operations that are executed in between subsequent launches of the CUDA kernel. This includes unavoidable operations such as CUDA kernel argument setup, thread synchronization, device management, and implementation-specific memory copies of collision and vertex data and provides only little room for further optimization.

The CPU overhead is very visible in Figure 10 as a variable that is mostly constant regardless of the total number of springs (or model resolution). This implies that as the GPU kernel time is increased, e.g., when the model resolution is increased, the effect of the overhead becomes less and less apparent. Note that the CPU overhead may also vary for different GPU architectures.

springs	masses	time (ms)			time/spring (ns)	
		GPU	CPU-Ov	Total	GPU	Total
3614788	146476	0.886	0.224	1.110	0.245	0.307
2336252	95487	0.589	0.214	0.803	0.252	0.344
1638652	67477	0.433	0.206	0.639	0.264	0.390
1004416	41851	0.281	0.187	0.468	0.280	0.466
499364	21276	0.148	0.174	0.322	0.296	0.646
395772	17000	0.142	0.174	0.316	0.359	0.798
349844	15096	0.137	0.174	0.312	0.392	0.892
268944	11726	0.0888	0.172	0.260	0.330	0.968
251700	11000	0.0766	0.170	0.246	0.302	0.979
233660	10248	0.0755	0.171	0.246	0.321	1.05
202068	8876	0.0732	0.172	0.245	0.361	1.21
146588	6569	0.0553	0.163	0.218	0.375	1.49
102660	4688	0.0543	0.164	0.219	0.529	2.13
54282	2575	0.0551	0.163	0.218	1.01	4.02
24024	1210	0.0433	0.163	0.206	1.80	8.58
5142	307	0.0423	0.158	0.200	8.23	38.9

Table 2. Performance average of fastest implementation (multi-device with double buffering) in terms of pure GPU kernel, CPU overhead (CPU-Ov), and total framework time.

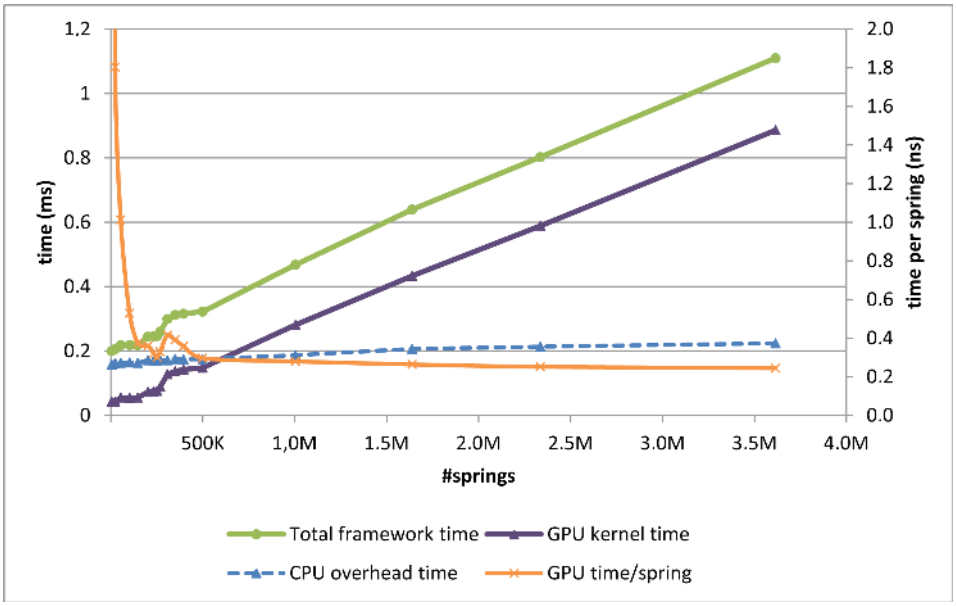

Figure 10. Relevant times for our fastest (multi-device) implementation. Note the linearity of the GPU time and total time, the constant CPU overhead (as the difference of the former two), and the horizontally asymptotic GPU time per spring.

Measurement unit or property	Profiler output
Kernel block size	256
Kernel occupancy	50%
Shared memory usage	0 KB
Register count	39

Table 3. Analysis of our CUDA implementation as provided by the CUDA Visual profiler.

6.4. Parallelization and Scalability

Our performance results show that our algorithm and its implementations scale almost linearly with the total number of springs, which is an overall satisfactory result. This, however, does not tell much about the algorithm's parallel scalability: the efficiency of the algorithm when using increasing numbers of parallel processing elements (threads).

To quantify the parallel scalability, we first have to look at the CUDA kernel profiling results in Table 3. Our particular Tesla C2050 GPU devices are capable of running a maximum of $1536 \times 15 = 23040$ parallel threads with all 15 SMs 100% occupied. Our profile results show an occupancy of 50%, mainly due to the high register usage of the kernel. The maximum amount of parallel threads at any time for our CUDA kernel is thus $23040 * 0.50 = 11520$. Since our kernel requires a thread for every mass, the relevant performance range to inspect is that between 0 and the parallel maximum of 11520, after which the device is saturated, as visualized in Figure 11. When looking at the typical model resolutions (e.g., 95487) in comparison to the parallel maximum of 11520 masses, one can observe that the device is quickly saturated, emphasizing the relevance of the overall performance over parallel scalability.

An ideal parallel kernel will achieve linear scaling when performance stays constant while the workload is increased in direct proportion to the number of threads. Note that the kernel is not computation-bound, but memory-bound, due to its frequent accesses in global GPU memory. This creates a memory access bottleneck that causes a certain degree of serialization to occur during execution. Parallel scalability can therefore not be ideally linear, and, thus, the performance pattern cannot be fully constant.

Figure 11 shows a significant stepped pattern that alternates between near-constant sustained performance and sudden decrease and confirms that the kernel is indeed parallelized. The stepping can be explained by the internal mechanisms (e.g., warp scheduling, memory access handling) of the GPU device that are involved with scheduling the kernel's threads on the available SMs and may continue in a similar pattern beyond the point of device saturation. As this typically varies with different GPU architectures, it provides an opportunity for improvement by taking advantage of more recent, more efficient CUDA hardware. Together with the suboptimal distribution of

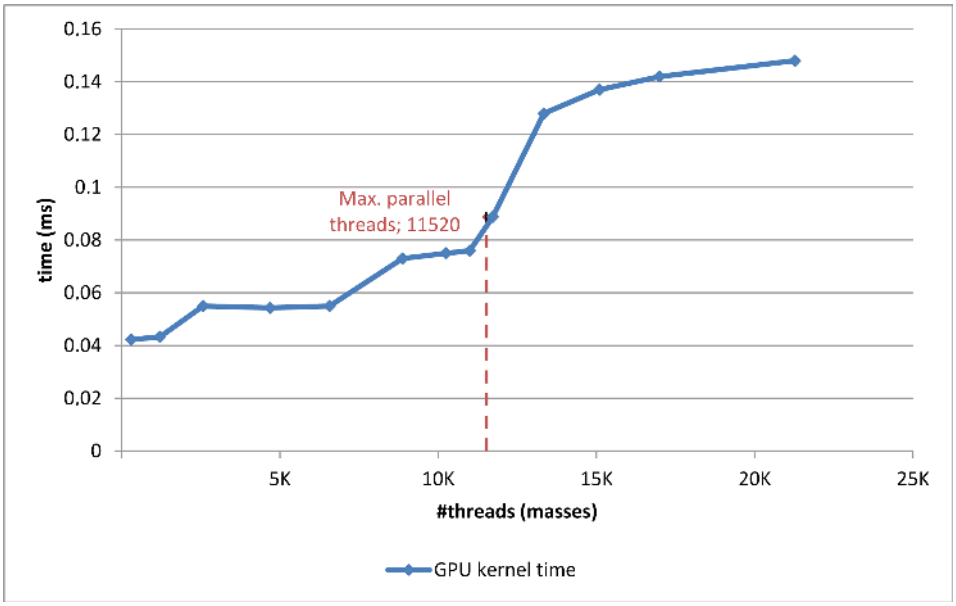

Figure 11. GPU kernel time of our fastest (multi-device) implementation for threads up to (and exceeding) the CUDA parallel maximum of 11520.

our CUDA kernel on the SMs, this leaves enough room for future kernel optimization in terms of parallel scalability.

7. Future Work

In future work, we would like to further investigate the scalability of our parallel algorithm and perform in-depth optimization of our CUDA implementation. If we consider future GPU architectures with improved resources in computational power and increased memory bandwidth, we would like to achieve better parallel scalability. Additionally, we would like to find out if other platforms such as OpenCL, provide any benefits in terms of practical use and performance. The algorithm may also be suitable for distributed setups with multiple GPUs or clusters, but this would require a new level of synchronization and smart resource management.

In this paper, we chose to fully focus on the performance analysis of our algorithm and its implementations. A detailed haptic response analysis, in which the haptic correctness (e.g., no oscillation, stable output) can be quantified, requires a larger scale assessment and is therefore considered as future work.

It is nevertheless important for us to focus on the addition of new features to our existing algorithm in the context of virtual surgery. Virtual tools such as scalpels, tweezers, and staplers would allow an even higher level of model interaction and more extensive scenarios. We believe that such tools could easily be added to our current algorithm by extending the existing properties and force

calculations.

We would also like to investigate the application of our algorithm in other fields that utilize soft-body dynamics, such as computer games. Without haptic feedback, we can afford to run the algorithm at far lower rates, opening up opportunities for even more complex data sets.

Acknowledgements

We thank M. Weber and M. Stoelinga of the University of Twente for their supervision and insights.

References

AHMAD, A., ADLY, S., TERRAZ, O., AND GHAZANFARPOUR, D. 2007. Stability analysis of filtered mass-spring systems. In *Proceedings, Theory and Practice of Computer Graphics*, Eurographics, Aire-la-Ville, Switzerland. 32

CHADWICK, J. E., HAUMANN, D. R., AND PARENT, R. E. 1989. Layered construction for deformable animated characters. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, SIGGRAPH '89, 243–252. 31

CHEN, D. T., AND ZELTZER, D. 1992. 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*, ACM, New York, NY, SIGGRAPH '92, 89–98. 31

CHOI, Y.-J., HONG, M., CHOI, M.-H., AND KIM, M.-H. 2005. Adaptive surface-deformable model with shape-preserving spring. *Computer Animation and Virtual Worlds 16*, 1, 69–83. 36

COURTECUISSE, H., JUNG, H., ALLARD, J., DURIEZ, C., LEE, D. Y., AND COTIN, S. 2010. GPU-based real-time soft tissue deformation with cutting and haptic feedback. *Progress in Biophysics and Molecular Biology (Special Issue on Biomechanical Modelling of Soft Tissue Motion) 103*, 2-3, 159–168. 32

FARIAS, T. S. M. C. D., ALMEIDA, M. W. S., TEIXEIRA, JO, A. M. X. N., TEICHRIEB, V., AND KELNER, J. 2008. 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*, IEEE Computer Society, Washington, DC, 45–52. 31

KIM, J., CHOI, C., DE, S., AND SRINIVASAN, M. A. 2007. 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. 31

KÜHNAPFEL, U., NEISIUS, B., KRUMM, H., AND HÜBNER, M. 1993. Cad-based simulation and modelling for endoscopic surgery. *Proc. Med Tech, SMIT 94*. 31

MOSEGAARD, J., AND SØRENSEN, T. S. 2005. GPU Accelerated Surgical Simulators for Complex Morphology. In *VR '05: Proceedings of the 2005 IEEE Conference 2005 on Virtual Reality*, IEEE Computer Society, Washington, DC, USA, 147–154, 323. 31

MOSEGAARD, J., HERBORG, P., AND SØRENSEN, T. S. 2005. A GPU accelerated spring mass system for surgical simulation. *Studies in Health Technology and Informatics 111*, 342–348. 31

MÜLLER, M., HEIDELBERGER, B., HENNIX, M., AND RATCLIFF, J. 2007. Position based dynamics. *J. Vis. Comun. Image Represent. 18*, 2, 109–118. 31

NEDEL, L. P., AND THALMANN, D. 1998. Real Time Muscle Deformations using Mass-Spring Systems. In *Proceedings of Computer Graphics International 1998*, IEEE Computer Society, Washington, DC, CGI '98, 156–165. 31

PROVOT, X. 1996. Deformation constraints in a mass-spring model to describe rigid cloth behavior. In *Proceedings of Graphics Interface*, A K Peters Ltd., Natick, MA, 147–154. 31

RASMUSSON, A., MOSEGAARD, J., AND SØRENSEN, T. S. 2008. Exploring Parallel Algorithms for Volumetric Mass-Spring-Damper Models in CUDA. Springer-Verlag, Berlin, Heidelberg, 49–58. 31, 35

SHI, H., AND PAYANDEH, S. 2008. GPU in haptic rendering of deformable objects. In *Haptics: Perception, Devices and Scenarios*, M. Ferre, Ed., vol. 5024 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin Heidelberg, 163–168. 31

TERZOPOULOS, D., PLATT, J., BARR, A., AND FLEISCHER, K. 1987. Elastically deformable models. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, SIGGRAPH '87, 205–214. 30

ZHANG, J.-S., CHEN, H., WU, W., AND HENG, P.-A. 2010. An interactive high-fidelity haptic needle simulator with gpu acceleration. In *Proceedings of the 9th ACM SIGGRAPH Conference on Virtual-Reality Continuum and its Applications in Industry*, ACM, New York, NY, VRCAI '10, 347–352. 32, 34

Author Contact Information

Cecill Etheredge
P.O. Box 96
7500AB Enschede
The Netherlands
c@ijsf.nl

Eelco Kunst
Vrest Medical
Moutlaan 20
7523MD Enschede
The Netherlands
ekunst@vrest.nl

Anton Sanders
Vrest Medical
Moutlaan 20
7523MD Enschede
The Netherlands
asanders@vrest.nl

C. E. Etheredge, E. E. Kunst, A. J. B. Sanders, Harnessing the GPU for Real-Time Haptic Tissue Simulation, *Journal of Computer Graphics Techniques (JCGT)*, vol. 2, no. 2, 28–54, 2013
http://jcgt.org/published/0002/02/03/

Received:	2013-01-10		
Recommended:	2013-04-30	Corresponding Editor:	Tomer Moscovich
Published:	2013-07-19	Editor-in-Chief:	Morgan McGuire