Exploring Parallel Algorithms for Volumetric Mass-Spring-Damper Models in CUDA

Allan Rasmusson[1,2], Jesper Mosegaard[3], and Thomas Sangild Sørensen[1,4]

[1] Department of Computer Science University of Aarhus, Denmark
[2] Center for Histoinformatics, University of Aarhus, Denmark
[3] Alexandra Institute, Denmark
[4] Institute of Clinical Medicine University of Aarhus
{alras,mosegard,sangild}@daimi.au.dk

Abstract. Since the advent of programmable graphics processors (GPUs) their computational powers have been utilized for general purpose computation. Initially by "exploiting" graphics APIs and recently through dedicated parallel computation frameworks such as the Compute Unified Device Architecture (CUDA) from Nvidia. This paper investigates multiple implementations of volumetric Mass-Spring-Damper systems in CUDA. The obtained performance is compared to previous implementations utilizing the GPU through the OpenGL graphics API. We find that both performance and optimization strategies differ widely between the OpenGL and CUDA implementations. Specifically, the previous recommendation of using implicitly connected particles is replaced by a recommendation that supports unstructured meshes and run-time topological changes with an insignificant performance reduction.

Keywords: Mass-Spring-Damper Models, GPGPU and Deformable Models.

1 Introduction

In the past many biomechanical models have been suggested to simulate soft tissue deformations [1]. Most popular are the finite element models (FEMs) and mass-spring-damper models (MSDMs). Of these models the non-linear FEMs provide the most accurate description of the tissue behavior [2][3]. They require however a significant amount of computation which can be prohibitive in many real-time applications. At the expense of precision, particularly for large deformations, linearized FEMs have been suggested to alleviate this problem [4][5]. MSDMs combine non-linear tissue characteristics *and* fast computation. Due to these properties the MSDMs have been widely used to simulate tissue deformation in existing surgical simulators.

Commodity graphics hardware (GPUs) is an emerging platform for general purpose computation (GPGPU). It is increasingly utilized to accelerate the computation of biomechanical models [6][7][8]. The results obtained in these references were accomplished by exploiting well-known graphics APIs such as

F. Bello, E. Edwards (Eds.): ISBMS 2008, LNCS 5104, pp. 49–58, 2008.

OpenGL or DirectX. This is, unfortunately, a cumbersome process with a steep learning curve. In order to make GPGPU more accessible, one of the major hardware vendors, NVIDIA, recently released a new framework named CUDA - Compute Unified Device Architecture [9].

The purpose of this paper is to investigate which benefits the CUDA framework offers over OpenGL (or equivalently DirectX) in the computation of MSDMs, particularly with respect to speed and functionality. We evaluate multiple parallel implementation strategies for the MSDMs in CUDA and compare their performance to highly optimized OpenGL code [7]. Specifically, we evaluate whether the conclusions derived in [7] can be transferred from OpenGL to CUDA.

2 Theory

2.1 CUDA Overview

The CUDA framework exposes the multiprocessors on the GPU for general purpose computation through a small number of simple extensions of the C programming language. Compute intensive components of a program can be offloaded to the GPU in so-called *kernels*, each of which can be executed in parallel on different input data. This is known as SIMD, Single Instruction Multiple Data. A configuration of *threads* that will execute the kernel in parallel is specified as *blocks* of threads with constant width, height and depth. As the maximum number of threads in a block is limited (currently to 512), multiple blocks are distributed in a rectangular *grid* in order to obtain the desired number of threads. CUDA maps this grid to the GPU such that each multiprocessor executes one or more blocks of threads.

To achieve optimal performance it is important to minimize the cost of memory accesses in a kernel. This is achieved through careful utilization of the different memory pools, which are depicted in Fig. 1. The figure illustrates the small amount of on-chip shared memory which can be used within a block for inter-thread communication. Utilizing this shared memory efficiently yields memory accesses as fast as register accesses (2 clock cycles). In contrast, the main device memory accessible by all threads, has a worst-case access time of 400-600 clock cycles. Finally, Fig.1 shows the texture memory which can be utilized for very fast cached access to any give subset of the device memory.

As any CUDA-device is able to read 32-bit, 64-bit or 128-bit in a single instruction it is furthermore important to organize data in device memory to utilize this. In particular data should be aligned to the appropriate 4, 8 or 16 byte boundaries, even for data of sizes not matching 32-bit, 64-bit or 128-bit. If three floats are needed, performance is optimized by storing four properly aligned floats which can be read in a single 128-bit memory, contrary to storing only three float which are accessed using a 64-bit *and* a 32-bit memory access. The cost is the extra memory needed for padding.

The real benefit is, however, when threads executed in parallel access a contiguous part of the device memory. If consecutive threads access consecutive

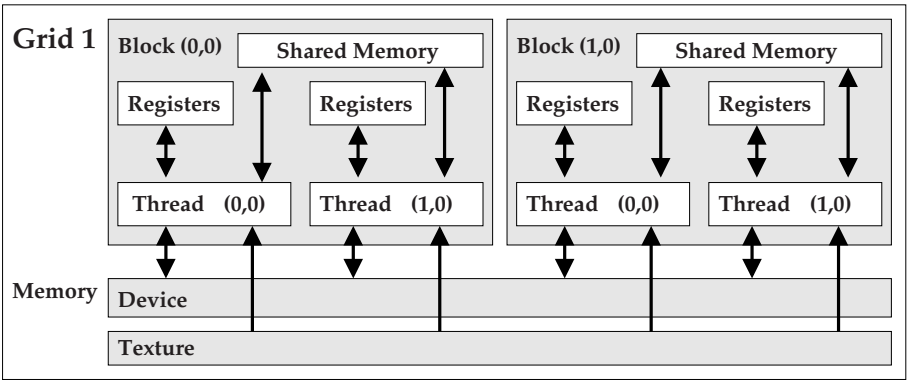

Fig. 1. CUDA Memory Model, adapted from [9]. The arrows indicate read or write access for a thread, ie. textures are *non*-writable. In this example the configuration of threads is a 2×1 grid of 2×1 blocks resulting in 4 threads in total.

memory address, say 32-bit floats, the individual memory instructions are replaced by a *single* memory access. This is known as memory coalescence and is at present possible for groups of 32 consecutive threads.

2.2 Mass-Spring-Damper Model (MSDM)

In a mass-spring system particles $p_i,\ i = 1 \ldots N$ with masses m_i are interconnected by springs. Every particle is displaced by the forces induced by its interconnecting springs. The relation is described by a second order differential equation according to Newton's second law of motion:

$$m_i \mathbf{a}_i = \sum_j \mathbf{f}_{ij}, \tag{1}$$

where $\mathbf{a}_i$ is an acceleration vector for particle i and the force $\mathbf{f}_{ij}$ along a spring between particle i and particle j is expressed using Hookes Law:

$$\mathbf{f}_{ij} = k_{ij}(l_{ij} - \|\mathbf{x}_i - \mathbf{x}_j\|)\frac{\mathbf{x}_i - \mathbf{x}_j}{\|\mathbf{x}_i - \mathbf{x}_j\|}. \tag{2}$$

The term l_{ij} constitutes the rest length of the spring, and k_{ij} is the spring constant that determines the elastic property of the spring. The non-linearity of the MSDM stems from the term $\|\mathbf{x}_i - \mathbf{x}_j\|$.

One method to numerically integrate equation (1) is the *Verlet* integration scheme. It is a particularly good choice as the updated position for each particle is calculated solely from the force vector $\mathbf{f}$ and the particle's two previous positions. Moreover, it can easily be computed for each particle in parallel. The Verlet integration is given by:

$$\mathbf{x}(t + \Delta t) = 2\mathbf{x}(t) - \mathbf{x}(t - \Delta t) + \frac{\mathbf{f}(t)}{m}(\Delta t)^2 \tag{3}$$

Artificial damping can be introduced in an ad-hoc manner [10].

$$\mathbf{x}(t + \Delta t) = (2 - \lambda) \cdot \mathbf{x}(t) - (1 - \lambda) \cdot \mathbf{x}(t - \Delta t) + \mathbf{f}(t)(\Delta t)^2. \quad (4)$$

2.3 Parallel Implementation Strategies

As presented in [7], on the gpu there are two implementation strategies utilizing either an *implicit* or *explicit* representation of the springs connected to a given particle.

The explicit strategy is the more general of the two strategies. For each particle it maintains a list of indices to particles to which it is connected. Two memory accesses are required to look up the position of each neighboring particle; one memory access to acquire the index of the neighbor particle, and one further memory access to determine the position of the particle at that index. Using this data structure it is straight-forward to represent any given connectivity of the MSDM.

The implicit strategy on the other hand requires datasets in which the particles are located in a regular three-dimensional grid (or structured mesh). Here, a particle can be connected to its neighboring particles only. For any particle, the addresses of neighbor positions can now be calculated by using a fixed set of constant offsets. Hence only one memory access is required to retrieve the position of each neighbor particle. To represent arbitrary morphology, particles are marked either active or inactive and only the active particles are considered part of the desired morphology.

3 Methods

This section describes how the strategies presented in section 2.3 for solving the MSDM have been implemented in CUDA.

The basic layout of test data used in this paper is a grid of particle positions represented as a 3D float array. Only adjacent particles in the 3D grid are connected by springs. Spring rest lengths are thus fixed as illustrated in Fig. 2.

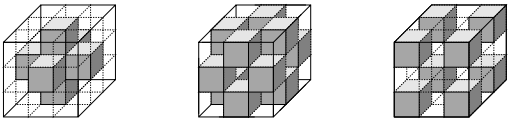

Fig. 2. Categorization of neighbors for a particle in the center of a $3 \times 3 \times 3$ cube. (a) rest length 1, (b) rest length $\sqrt{2}$ and (c) rest length $\sqrt{3}$. The configurations shown denote spring groups 1, 2 and 3 respectively.

3.1 Implicit Addressing

The computation associated to each particle, both active and inactive, is addressed by a single thread in CUDA. From the thread and block ids in the grid a mapping is established to the corresponding memory address of the particle. A neighbor mask (bitpattern) is used to differentiate active and inactive neighbors. Figure 3 shows an example of the bitpattern for a particle which is connected to three neighbors. Each bit is tested iteratively using logical operators. If the entire neighbor mask is zero, the particle corresponding to the current thread is an inactive particle. Hence, a single memory access determines whether a particle is active or inactive.

Fig. 3. Neighbor mask for a particle connected to neighbors 1,2 and 10. Spring group 1 is represented as the lowest 6 bits for which it is known that the rest length is equal to 1. Similarly, bits 6-17 represent spring group 2 with a rest length of $\sqrt{2}$. Spring group 3 is omitted in this example.

The two most recent arrays of the particle positions are required to compute equations (2) and (4). These arrays should be accessed either using cached device memory or shared memory. Memory access can easily be cached, whereas utilizing shared memory requires us to manually handle memory transfer to shared memory from device memory.

Shared Memory Implementation: The "rectangular" subset of device memory corresponding to the threads (particles) in the current block is initially copied from device memory to shared memory from where it is subsequently accessed. The shared memory layout is padded with an additional layer of particles in order for the threads (particles) at the border of the block to access their neighbor particles corresponding to a different block. This is illustrated in figure 4. To access neighbors of different depth, three rectangles like the one depicted in figure 4(b) is copied to shared memory for each block initially in the kernel. This strategy limits access to device memory to a few reads per particle.

Memory Coalescence: The particles are represented in memory as three floats for position and a 32-bit word for the neighbor mask. As it is possible to encode the 32-bit neighbor mask in a float, all information for a particle fits neatly into a float4 datatype. By using a block width to a multiple of 32, the criteria for memory coalescence is supported for all groups of 32 threads executed in parallel.

3.2 Explicit Addressing

The basic layout of data for the explicit addressing strategy is a *compact* array of particle positions containing only active particles. This array is accessed from

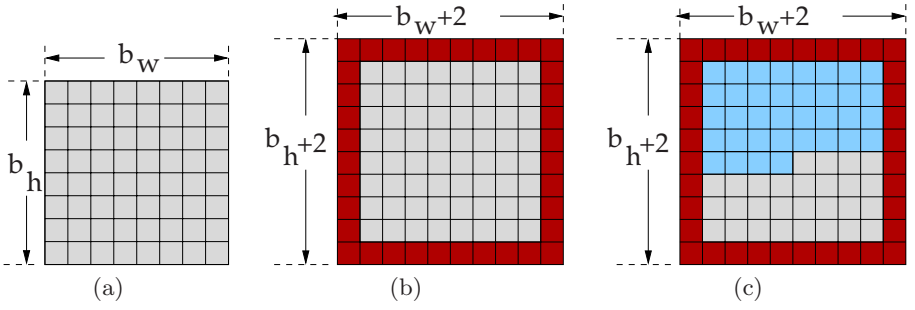

Fig. 4. Copying particle positions from device to shared memory. (a) depicts the block of particles of width b_w and height b_h, (b) shows the additional red frame of neighbor particles, (c) the blue particles are responsible of reading the red frame into shared memory.

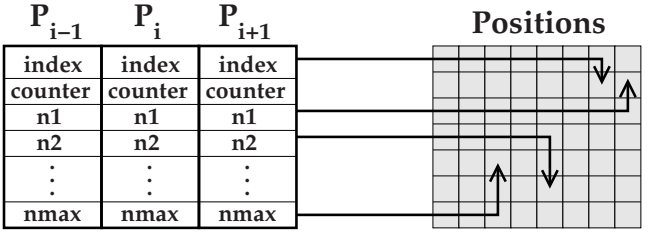

Fig. 5. Explicit connections for particles p_{i-1}, p_i and p_{i+1}

device memory using a 1D cache. Shared memory is not used due to the lack of locality in general unstructured meshes.

For each thread a list containing the particle indices of the corresponding particle and all connected particles is stored explicitly as illustrated in Fig. 5. The list has a predetermined maximum number of spring entries to allow easy indexing into an array of lists based on the thread and block id. Furthermore, a counter is included in each list to indicate the number of springs actually present. This data structure makes run-time changes to the topology very easy to support, i.e. to erase or add springs due to cutting or suturing.

A secondary list is maintained to provide per spring parameters such as spring stiffness and rest lengths. In the special case that particles are known to be laid out in a regular grid/structured mesh, this secondary list can be omitted; spring rest lengths can be determined from the corresponding spring group indicated by an individual counter per group. In this special case we furthermore assume that the spring stiffness is constant for all springs.

Memory Coalescence: Memory coalescence is not achieved if the neighbor lists are simply concatenated and stored in memory. The reason is that there is a separation of $32*(nmax+2)$ bits between the individual words read in parallel by the threads. Ie., the necessary contiguous memory layout is not present. In order

P_0	index	counter	n1	n2	· · ·	nmax
P_1	index	counter	n1	n2	· · ·	nmax
P_2	index	counter	n1	n2	· · ·	nmax

(a) Data layout *not* supporting memory coalescence since indices, counters and neighbor addresses are store linearly for each thread.

· · · P_{i-1}	P_i	P_{i+1} · · ·
index	index	index
counter	counter	counter
n1	n1	n1
n2	n2	n2
⋮	⋮	⋮
nmax	nmax	nmax

(b) Data layout supporting memory coalescence. By storing the neighbor lists "vertically" yields consecutive indices, counters and neighbor addresses for threads in parallel.

Fig. 6. Memory layout ensuring coalesced access for the explicit strategy

to have consecutive threads read consecutive memory addresses, it is necessary to first store all 32-bit indices, followed by the counter and the neighbor addresses. In a sence, the lists are stored "vertically". This is illustrated in Fig. 6. Again, a block width of a multiple of 32 fullfills the memory coalescence criterias for all groups of 32 threads.

3.3 Test Setup

To test the SMDM implementations we selected two datasets; a compact box and a realistic morphological dataset of a heart obtained from 3D MRI and previously used in a cardiac surgery simulator [11]. Both datasets, listed in Tab. 1, are stored as a binary three-dimensional grid indicating whether a voxel is part of the morphology or not. This data layout can be used by both the implicit and explicit layout and is thus suitable for comparison. The box dataset is optimal for the implicit method since no inactive particles are present.

Consistent with the OpenGL implementation to which we intend to compare performance [7], all implementations use a fixed connectivity pattern consisting of spring groups 1 and 2.

Furthermore, tests were made on boxes of different sizes in order to investigate how performance scales with increasing input size. This was done only for the explicit implementations as the ratio of active vs. inactive particles is irrelevant for this method.

The test platform was Windows XP 32bit, AMD 64 FX55 at 2.61Ghz on MSI K8N-Diamond, PCI-Express x16, GeForce 8800 GTX GPU (768 MB RAM), CUDA Release 1.1, Nvidia Forceware v. 169.21.

Table 1. Technical data about datasets

Dataset	Total Particles	Active Particles	Springs	Ratio Active/Total
Heart	140760	29449	220495	0.209
Box	32768	32768	280240	1.00

4 Results

Results of simulation runs for the described strategies are summarized in Fig. 7. For comparison results of an OpenGL implementation of the implicit strategy are also included.

The execution times for the explicit methods run on boxes of varying sizes are shown in Fig. 8.

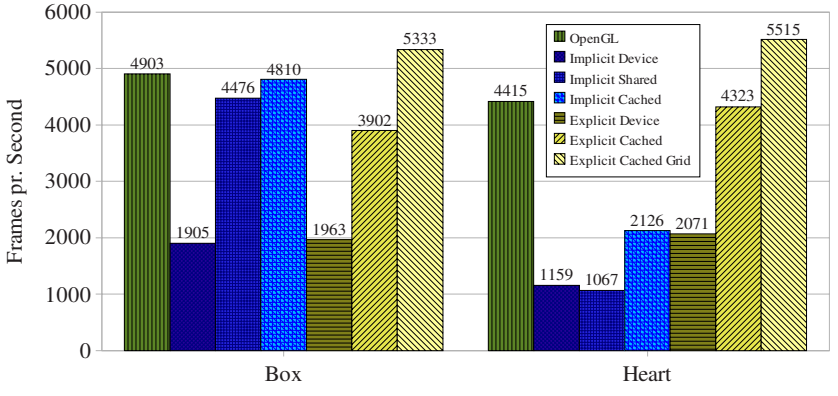

Fig. 7. Chart of performance for the original OpenGL and the implicit and explicit CUDA implementations. For the implicit method three memory strategies (device, shared, and cached) have been tested, while the explicit method is tested for device memory access and cached access. The cached is further divided into a general and a version optimized for grids, the "Explicit Cached Grid".

5 Discussion

In this paper we investigated various GPU implementations of the MSDM using CUDA. The purpose was to evaluate whether previously published results utilizing OpenGL for GPGPU could be reproduced or even improved, and to evaluate if the recommended strategies obtained using OpenGL transfers to CUDA.

Our results (Fig. 7) showed that the fastest CUDA implementation, the strategy utilizing cached explicit memory addressing, outperforms or performs equally well as the fastest OpenGL implementation.

In [7] it was reported that the OpenGL implementation using explicit addressing achieves only 50% of the performance obtained using implicit addressing due

to the extra memory indirection. For this reason implicit addressing was recommended over the explicit version. Using CUDA we notice however that this recommendation must be "inverted" to instead recommend explicit addressing. This change in recommendation has several positive side effects, most interestingly added flexibility and ease of implementation. The explicit method is very flexible since it can represent arbitrary geometry and easily allows for run-time changes of topology. Moreover, it turns out that the simplest of the implemented strategies performs superiorly.

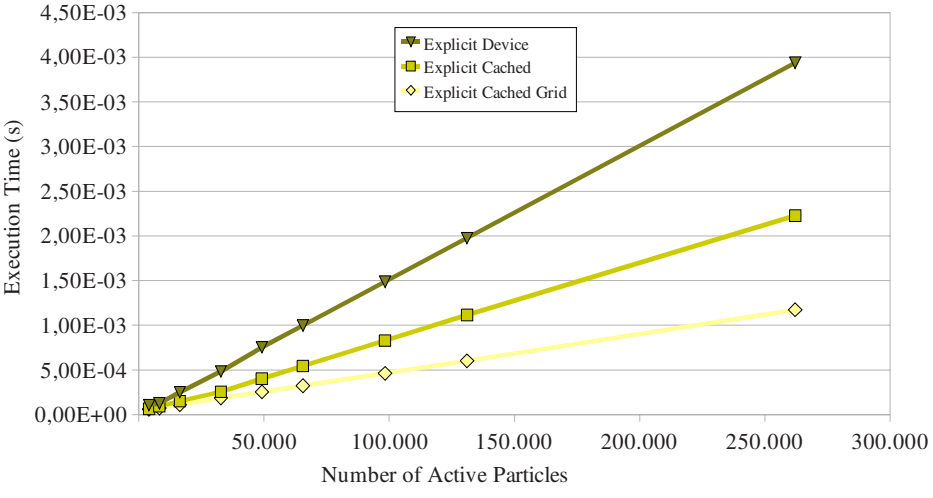

Fig. 8. Execution times for boxes of different sizes using the three explicit methods

Examining Fig. 7 in more detail, it becomes clear that the implicit method in CUDA does in fact perform well on the box dataset and only dissatisfactory on the heart dataset. Using OpenGL on the other hand, the implicit method performs comparably on both dataset. The reason is that the implicit method in CUDA cannot eliminate computation for inactive particles as effectively as the OpenGL implementation, which utilizes a hardware accelerated mask. For the box dataset this does not show since it contains no inactive particles.

Figure 8 shows that the execution times scales linearly with increasing input size (more than 50.000 active particles). For small datasets this may not be the case since small datasets do not utilize the parallel powers of the GPU fully.

From the discussion above it is evident that CUDA is a very interesting new platform to compute MSDMs in a surgical simulator. This requires however that additional aspects of a simulator, such as visualization and haptic feedback, are also adapted to the GPU. Fortunately, several techniques to implement such functionality are already described [6] [12] [13].

References

1. Liu, A., Tendick, F., Cleary, K., Kaufmann, C.: A survey of surgical simulation: applications, technology, and education. Presence: Teleoper. Virtual Environ. 12(6), 599–614 (2003)
2. Miller, K., Joldes, G., Lance, D., Wittek, A.: Total lagrangian explicit dynamics finite element algorithm for computing soft tissue deformation. Communications in Numerical Methods in Engineering 23, 121–134 (2007)
3. Irving, G., Teran, J., Fedkiw, R.: Invertible finite elements for robust simulation of large deformation. In: Eurographics/ACM SIGGRAPH Symposium on Computer Animation, pp. 131–140 (2004)
4. Bro-Nielsen, M., Cotin, S.: Real-time volumetric deformable models for surgery simulation using finite elements and condensation 15, 57–66 (1996)
5. Delingette, H., Cotin, S., Ayache, N.: A hybrid elastic model allowing real-time cutting deformations and force feedback for surgery training and simulation. In: Proceedings of the Computer Animation, May 1999, pp. 70–81 (1999)
6. Sørensen, T.S., Mosegaard, J.: An introduction to gpu accelerated surgical simulation. In: Harders, M., Székely, G. (eds.) ISBMS 2006. LNCS, vol. 4072, pp. 93–104. Springer, Heidelberg (2006)
7. Mosegaard, J., Sørensen, T.S.: Gpu accelerated surgical simulators for complex morphology. Proceedings of Virtual Reality 323, 147–154 (2005)
8. Taylor, Z., Cheng, M., Ourselin, S.: High-speed nonlinerar finite element analysis for surgical simulation using graphics processing units. IEEE Transactions on Medical Imaging (in Press, 2008)
9. NVIDIA: CUDA Programming Guide v. 1.1
10. Jakobsen, T.: Advanced character physics. In: Game Developers Conference (2001)
11. Sørensen, T., Stawiakski, J., Mosegaard, J.: Virtual open heart surgery: Obtaining models suitable for surgical simulation. Stud Health Technol. Inform. 125, 445–447 (2007)
12. Mosegaard, J., Sørensen, T.S.: Real-time deformation of detailed geometry based on mappings to a less detailed physical simulation on the gpu. In: Proceedings of Eurographics Workshop on Virtual Environments, vol. 11, pp. 105–111. Eurographics Association (2005)
13. Sørensen, T.S., Mosegaard, J.: Haptic feedback for the GPU-based surgical simulator. Proceedings of Medicine Meets Virtual Reality 14. Studies in Health Technology and Informatics 119, 523–528 (2006)