Data Structures and Transformations for Physically Based Simulation on a GPU

Perhaad Mistry[1], Dana Schaa[1], Byunghyun Jang[1], David Kaeli[1], Albert Dvornik[2], Dwight Meglan[2]

[1] Department of Electrical and Computer Engineering
Northeastern University, Boston, MA, U.S.A.
{pmistry, dschaa, bjang, kaeli}@ece.neu.edu
[2] Simquest LLC, Boston, MA, USA
{advornik, dmeglan}@simquest.com

Abstract. As general purpose computing on Graphics Processing Units (GPGPU) matures, more complicated scientific applications are being targeted to utilize the data-level parallelism available on a GPU. Implementing physically-based simulation on data-parallel hardware requires preprocessing overhead which affects application performance. We discuss our implementation of physics-based data structures that provide significant performance improvements when used on data-parallel hardware. These data structures allow us to maintain a physics-based abstraction of the underlying data, reduce programmer effort and obtain 6x-8x speedup over previously implemented GPU kernels.

1 Introduction

In any useful surgical simulation system, in order to meet the strict requirements of proper visual and behavioral illusion of reality, the system must solve a number of physics-based problems such as cutting and deformation at interactive speeds [12]. However surgical simulation cannot leverage the tricks that are used in "game physics". Physics engines for soft body simulation concentrate on real time and visually plausible results, while surgical simulation requires numerical stability and accuracy due to the critical nature of the simulation [12].

The present development trend of computational science software libraries is not driven by changes in problem-specific methodology [9], but by the fundamental shift of the underlying hardware towards heterogeneity and parallelism. This is particularly true for data-intensive problems such as finite element analysis. GPUs have become the technology of choice for data-parallel applications due to their potential for impressive speedups and their ability to accelerate a range of general purpose programs [3, 5, 10].

Our current work involves accelerating the physics simulation library PhysBAM [19] using the Compute Unified Device Architecture (CUDA) of NVIDIA GPUs for a real time surgical simulator. PhysBAM is an object oriented library that works with dynamically generated data structures to simplify the modeling of the underlying physics. Physically-based modeling techniques have been used to properly model time-varying properties such as geometry and topology [20].

Physics simulation algorithms possess inherent data parallelism, however parallelizing such algorithms naively leads to high overhead preprocessing since data parallelism is interspersed throughout the simulation and few compute-intensive "kernels" exist on which optimization efforts can be concentrated. This leads us to search for optimization techniques which can be applied more widely across an application and can improve data parallel performance irrespective of the underlying data structures and algorithms. In this paper, we describe methods to improve data layout and use them to accelerate physical simulation. We present a framework for physically- based simulation that automatically translates dynamic data structures to match the requirements of the GPU memory subsystem.

2 Related Work

A number of prior studies have addressed acceleration of physics simulation and finite element analysis using GPUs [6–9]. The GPU implementation of FEAST [8] is based on a scalable recursive multi-grid algorithm which prevents us from using it for our surgical simulation implementation due to the real time requirements of our environment and the need to simulate cutting. Farias et al. [16] discuss physically precise deformation and demonstrate very good performance for their particular methodology. Our work attempts to be agnostic to specific algorithms and provides a framework to implement different types of data-parallel physics algorithms that can effectively exploit the resources of a GPU.

The motivation behind our work is to build a simulation engine similar to Bullet [18] that models soft tissue deformation and cutting accurately enough to be applied to surgical simulation. Physics simulation for game and visual realism has been implemented using NVIDIA GPUs in PhysX [2] and is available as a middleware for CUDA capable GPUs. Other physics simulation work for CUDA-based hardware includes [17, 18]. For our simulated environment we need to provide accurate soft tissue deformation, so our goal is to more closely couple the physics of the problem with our data parallel implementation.

The implementation described in HONEI [9] is relevant to our work since Dyk et al. also explored the heterogeneity and parallelism between GPUs and CPUs. Our work is different from HONEI in the sense that our work is specific to Physics simulation and the relationship between data-parallel structures and the underlying physics parameters. The data structures provided within HONEI are oriented towards finite element analysis. The Simulation Open Framework Architecture (SOFA) [6] is a framework for surgical simulation, but does not provide the computational infrastructure that will be required for our future work. The Fenics Form Compiler [11] is also related to our work since it deals with the conversion of mathematical expressions into programs that can be executed using low-level linear algebra libraries for general purpose CPUs. Our method is complimentary to this compiler since it is related to data structures and is designed to improve performance for a different computing platform (GPUs).

3 Physically Based Simulation Framework

To describe our data-parallel physics simulation framework, we first discuss how the memory coalescing requirements for NVIDIA GPUs affect the design of our physics-based data structures. We then show how our framework can be used to build structures that exploit the GPU memory sub-system.

3.1 Coalesced Memory Accesses from Arrays of Objects

Physics based data-structures are arrays of dynamically generated objects denoting multiple instances of physical quantities like force, displacement, etc. Within a generic physics simulation engine, due to the lack of a prirori knowledge of data layouts, these arrays do not typically reside in contiguous locations in memory. The typical solution for working with arrays of objects on the GPU is to allocate a contiguous block of memory that can hold all the structures, and then copy the complete array of objects into consecutive locations in memory (Figure 1).

This approach, however, will impede performance due to the CUDA memory coalescing rules. Memory coalescing in CUDA is defined as reads or writes by threads to consecutive 4-byte elements in memory. If coalescing is not achieved, accesses are serialized and bandwidth degrades significantly. Figure 1 shows memory accesses when consecutive 3-element data structures try to access memory in CUDA.

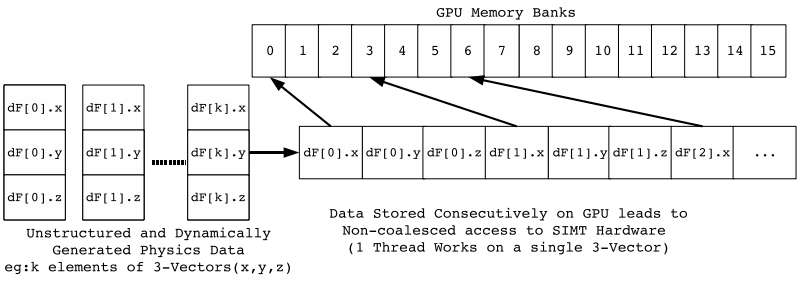


Fig. 1: For data structures stored consecutively, non-coalesced accesses occur.

As shown in Figure 2, we need to rearrange the allocated data in linear memory locations to map efficiently to the underlying data-parallel hardware.

3.2 Automated Framework for Physics Data Structures

We have implemented a framework that allows us to create data structures for physics simulation algorithms adhering to the memory coalescing requirements. The motivation behind our framework is illustrated by listing some calculations and data structures required for modeling deformation of a silicone cube. The technique used for deformation modeling is based on [14] and entails three main steps.

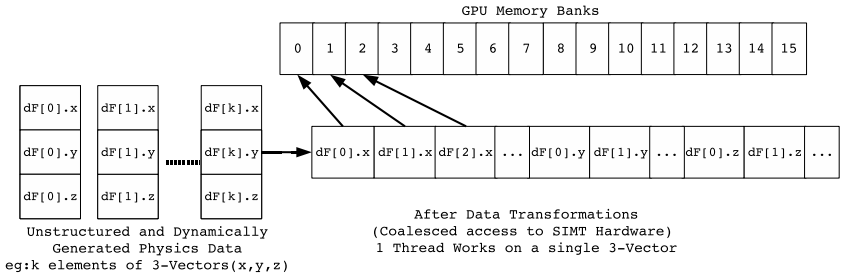


Fig. 2: Data structures that are transformed for coalesced memory access.

1. **Singular Value Decomposition(SVD):** The SVD of an array of 3x3 matrices is calculated using approximate methods [14, 13].
2. **Stress Derivative:** An array of structures of stress parameters denoting the constitutive model. The results are used to update the stiffness matrices for the iterative solver.
3. **Solving Linear System:** Arrays of 3x3 matrices and 3x3 symmetric matrices for stiffness arrays. This step also requires arrays of vectors for force and displacement.

In our simulator, the data structures and parallel algorithms are closely coupled to the physics theory. The physics based data structures denoted above are only a small subset of the possible dynamically generated arrays of objects. Due to the variety of algorithms that could be implemented using PhysBAM [19], little information is known apriori about the data structures and kernels[3] that will be invoked in a simulation. Due to this characteristic of our application, providing a limited set of optimized data. structures is not beneficial.

The architecture of our framework called **GPUPhysBAM** (GPU Physics Based Modeling) is shown in Figure 3. The physics simulation library dynamically generates data whose structure is determined as per the physics algorithm being simulated. The intermediate layer of GPUPhysBAM will allocate data in the GPU memory while keeping in mind constraints like alignment and ordering for inter-thread access. These constraints have to be satisfied in order to take advantage of the high-bandwidth memory bus between the GPU memory and the GPU's SIMD processors.

Implementation of data structures that can be reused and adapted to different simulation algorithms without sacrificing performance is key since we wish to maintain the generic nature of the simulation library and allow our data-parallel structures to impact a wider range of algorithms.

3.3 Data Transformations and Hierarchically Designed Data Structures

Due to the complicated nature of physics based simulation code where both the CPU and the GPU play a significant role in computation, data has to be organized such that computation on both the CPU and the GPU yields optimal performance.

[3] We refer to kernels as code that is parallelized and offloaded to the GPU

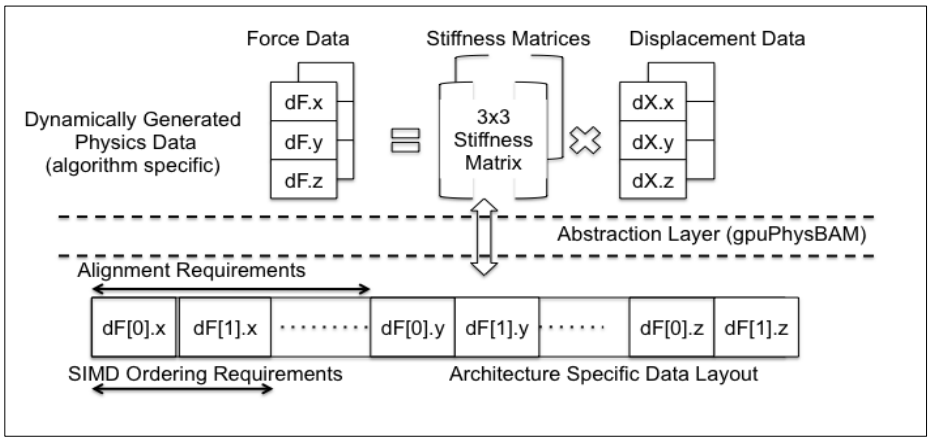


Fig. 3: gpuPhysBAM Architecture

A simple example that illustrates the power of properly designed data structures for the CPU and GPU is shown in Figure 4. We present a simple program $Y = f(X)$ where X and Y are arrays of 3-element vectors indexed as $X_i[0] - X_i[2]$ and $f()$ is a function that operates on each element of each vector independently. For the CPU layout, function $f()$ was applied as in the loop nest in Figure 4. For the loop in Figure 4, data layout 1 would be optimal. However, as discussed in Section 3.1 such a layout would prevent coalesced reads and writes on the GPU if each iteration of the loop maps to a thread. The optimal layout for CUDA (assuming each iteration of the loop is mapped to 1 thread) is shown in data layout 2, where the access pattern would follow Figure 2. An alignment factor (pitch) is needed, since GPU memory is divided into banks and optimized accesses can only begin at the starting location of the first bank. However, if a large amount of computation is carried out on the CPU as well, we should transform data from layout 2 back to layout 1. These transformations are automated and abstracted using our data structures[4].

The fundamental structure defined within our framework shown in Figure 3 is an **OBJECT ARRAY** which denotes a grouping of similar objects. The term "object" in this context refers to any ordering of data such as arrays, vectors, or matrices. Such structures represent most physics-based simulation data (e.g., force, displacement, stress, etc.) which consist of arrays of vectors or arrays of matrices. Even for the simplistic simulation discussed in Section 3.2, there exist a large number of different data structures. To make our data-parallel simulation framework extensible and independent of any particular algorithm, we design a framework for dynamically building data structures such that the data layout generated is always optimal when implemented in CUDA[5].

[4] We assume no dependencies occur across elements in each vector and no conflicting compiler optimizations are used

[5] Optimal refers to the optimal usage of memory bandwidth which occurs for coalesced data accesses

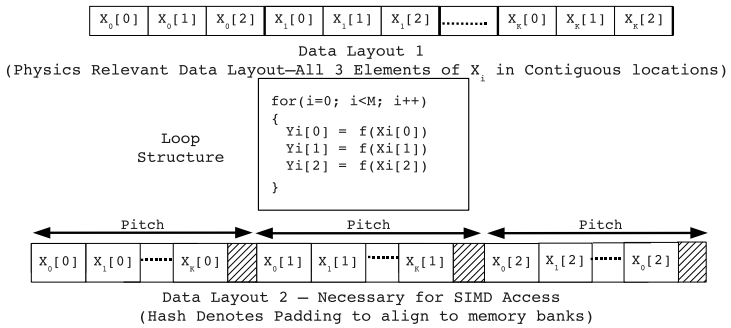


Fig. 4: Optimal data layouts for the CPU and the GPU.

The Single Instruction Multiple Thread (SIMT) model for CUDA requires well-structured access patterns across threads to coalesce memory accesses. Matching the memory access patterns to the architecture is critical since GPU performance is dependent on exploiting the high bandwidth present between the device memory and the Single Instruction Multiple Data (SIMD) units [3, 10, 1].

Figure 5 denotes the layout of our data structures used in the implementation of our GPUPhysBAM framework. Our underlying base class behaves as a CPU or GPU memory container. As shown in Figure 5, we abstract out GPU-specific operations in the 2^{nd} level where we store information such as strides, number of elements in an object and number of objects. The 3^{rd} level simply contains indexing functions for each element within an object.

Next, we demonstrate the benefits and extensibility of this approach. Figure 5 shows the implementation of an Object Array of 3x3 symmetric matrices which is commonly used in stiffness matrices. By inheriting from our GPU-generic class and implementing functions ($Get()$ and $Put()$ in SYMMETRIC_MATRIX_ARRAY to access data) **within** each object[6] of the array, we build data-parallel structures that will be efficiently accessed in a SIMD fashion on a GPU. The lower level $getdata()$ and $putdata()$ functions handle the indexing and the pitch calculations that are done to return data when given the object number and the element within the object. Thus, we provide the algorithmically relevant "3x3 Symmetric Matrix Array" while exploiting the wide memory bandwidth and many-core parallelism of a GPU.

A second example (shown in Figure 5) is an Object Array of structs used to describe a constitutive model. The Constitutive Model Array is also created in a similar fashion to the Symmetric Matrix Array. We derive the new class and simply write functions to access the respective value out of 12 elements that make up each object. As shown, similar to Symmetric Matrices, the lower level base class functions of $getdata()$ and $putdata()$ implement the required indexing.

By using contiguous memory allocated in base classes and controlling indexing using derived objects, we maintain the coalescing conditions for SIMT hardware, and the close coupling of the computation to the original physics theory. The utility of our

[6] in this case one object is 6 elements of a symmetric matrix considered column major

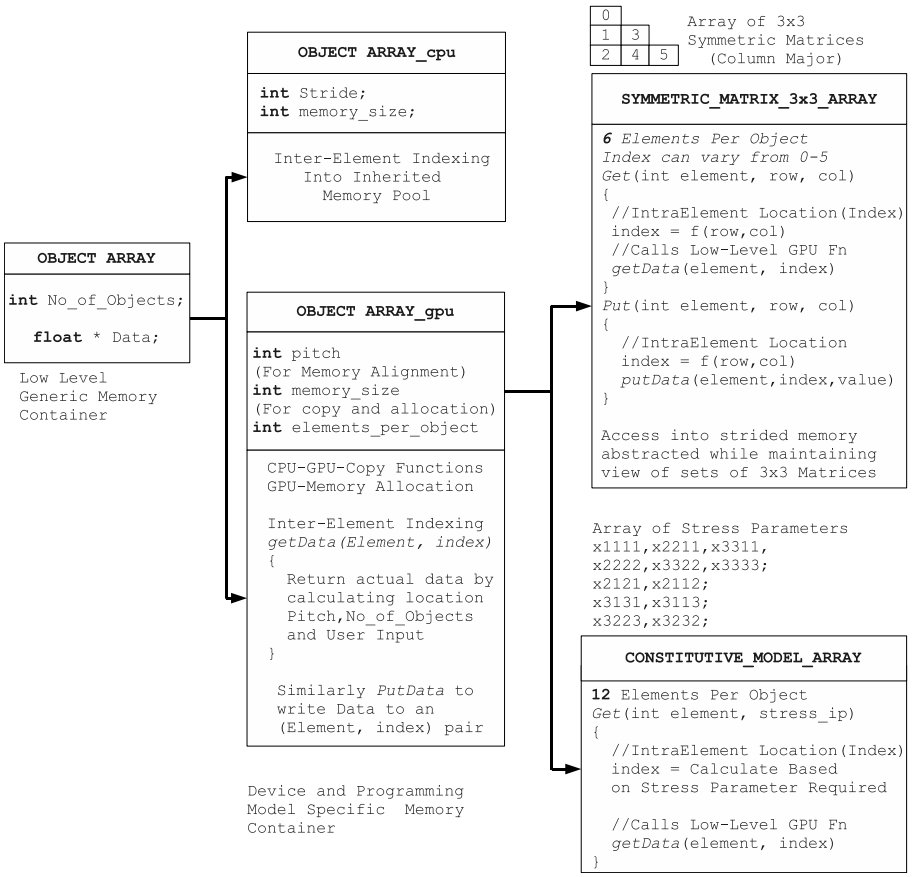
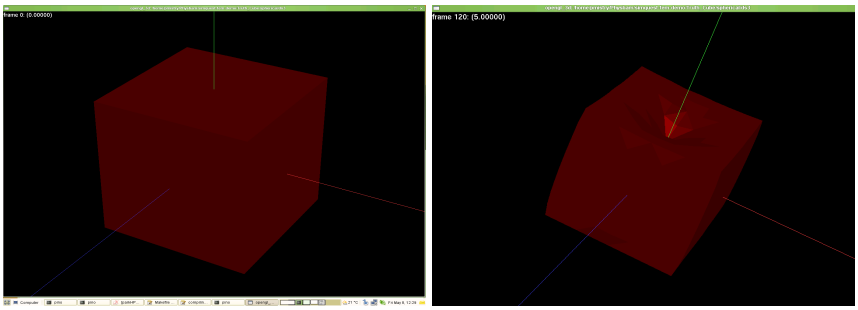


Fig. 5: Hierarchically declared data structures for both the CPU and the GPU.

framework lies in the fact that a domain expert could create the appropriate physics based data structures by simply inheriting the base class for the GPU functionality, and based on his/her expertise, simply write functions to access data within each object of the Object Array without knowing how the set of objects are laid out in memory.

4 Performance Results

The datasets used in our work are based on a popular model called the Truth Cube [15]. The Truth Cube serves as a model to validate soft-tissue deformation algorithms by comparing deformation obtained by any prototype method to known mechanical values. We use the Truth Cube to verify our data-structures. Figure 6a shows the Truth Cube in an undeformed position. Figure 6b shows the Truth Cube after the Quasistatic simulation for deformation



(a) Before Applying Deforming Forces (b) After Quasistatic Deformation simulation

Fig. 6: Deformation of a Truth Cube

The Quasistatic simulation of the Truth Cube shown in Figure 6 was implemented by using data-structures created within our framework. We benchmark 3 physics-relevant kernels from this simulation which take up a majority of the execution time. We compare performance of only the execution time of GPU kernels. Our baseline is naive only with respect to data layout, it is architecturally aware of the GPU and written using shared memory, textures and an optimal thread execution configuration. The CPU-GPU I/O and memory transformation overhead does not change across data layouts because the baseline also incurs transformation overhead since data generated is not in contiguous memory. The performance was measured on a system using an NVIDIA GTX-285 GPU, Intel Core 2 Duo with $4GB$ of RAM running Ubuntu 9.04, and CUDA 2.3.

Table 1: Performance Results and Benchmark Characteristics

Benchmark	Coalescing Improvement		Performance(ms)		Speedup
	Baseline	Hierarchial	Baseline	Hierarchial	
dP_from_dF	0.031	0.167	20.240	2.430	8.33x
3x3 Matrix, Diagonal Matrices, 1-element vectors					
Isotropic Stress Derivative	0.026	0.222	10.480	1.560	6.72x
Constitutive Mode Structs, Diagonal Matrices					
Add Force Differential	0.013	0.066	1.150	5.900	5.13x
Symmetric Matrices, 3-element Vectors					

The performance improvements of the physics kernels have been denoted in Table 1. The performance improvements are substantial even in computationally intensive kernels like Add Force Differential which use the shared memory of the device to hide most of device memory latency.

"Coalescing Improvement" in Table 1 denotes ratio of *requests* to *actual* memory transactions measured using the CUDA profiler [1]. The increased ratio when using our framework denotes the improvement memory access efficiency due to coalescing. The improvement in kernel performance when using our framework is due to the reduction in the number of *actual* memory transactions that the GPU memory subsystem processes.

The performance improvements presented here for each physics-based kernel do translate to an improvement in application level performance because these kernels constitute the bulk of the computation. For eg the *Add Force Differential* kernel is similar to a sparse matrix vector multiplication which is known to consume the bulk of the time spent in the Quasistatic simulation [14, 19]. The performance improvements presented in Table 1 are obtained for essentially no increase in programming effort or development time for an application developer because the same physics-derived data structure design API is mantained which allows to the improved data strutures to be simply inserted from underneath the physics simulator.

5 Conclusion

In this work we describe techniques that allow us to implement physics-based simulations efficiently on NVIDIA GPUs. Due to the variety of algorithms that can be implemented using our physics simulator, we focus on implementation techniques and optimizations that are extensible and generic so that they can have impact on a broader class of data-parallel physics simulations. Our framework is extendable to different types of physics-related objects and can also be adapted to other algorithms targeting NVIDIA GPUs. We have used this same framework to implement other deformation algorithms based on MultiGrid methods and Backward Euler solvers. Our future work includes supporting more complicated models and evaluating the associated performance enhancements possible.

References

1. NVIDIA: NVIDIA CUDA Programming Guide 2.0 2008, http://www.nvidia.com/cuda.
2. NVIDIA: NVIDIA Physx 2008. http://www.nvidia.com/physx.
3. Nguyen, H., Gpu gems 3, 2007, Addison-Wesley Professional.
4. Harris, M.: Optimizing parallel reduction in cuda, NVIDIA Developer Technology, 2007.
5. Luebke, D., Harris, M.,Govindaraju, N., Lefohn, A., Houston, M., Owens, J., Segal, M. Papakipos, M. Buck, I., GPGPU: general-purpose computation on graphics hardware, Proceedings of the 2006 ACM/IEEE conference on Supercomputing,(2006).
6. Allard, J. and Cotin, S. and Faure, F. and Bensoussan, P.J. and Poyer, F. and Duriez, C. and Delingette, H. and Grisoni, L.: Sofa-an open source framework for medical simulation, Studies in health technology and informatics, Vol **125**, 13.
7. Orion Lawlor and Sayantan Chakravorty and Terry Wilmarth and Nilesh Choudhury and Isaac Dooley and Gengbin Zheng and Laxmikant Kale: ParFUM: A Parallel Framework for Unstructured Meshes for Scalable Dynamic Physics Applications, "Engineering with Computers" **22** (2006) 215–235, Springer London.

8. Turek, S. and Becker, Ch. and Kilian, S.: Some concepts of the software package FEAST, Vector and Parallel Processing VECPAR 1998, Springer, Berlin, 271–284.
9. Danny van Dyk, Markus Geveler, Sven Mallach, Dirk Ribbrock, Dominik Gddeke, and Carsten Gutwenger: HONEI: A collection of libraries for numerical computations targeting multiple processor architectures, Computer Physics Communications, Vol **180** Issue 12 (2009) 2534–2543.
10. Joshua A. Anderson and Chris D. Lorenz and A. Travesset: General purpose molecular dynamics simulations fully implemented on graphics processing units, Journal of Computational Physics, **227** 2008 (5342 – 5359).
11. Kirby, Robert C. and Logg, Anders: A compiler for variational forms, ACM Trans. Math. Softw.**32**, (417–444).
12. Bro-Nielsen, M.: Finite element modeling in surgery simulation, Proceedings of the IEEE, Vol **86** (1998) 283–291.
13. Garcia, E.: Information Retrieval Tutorial (2005), www.miislita.com,
14. Teran, Joseph and Sifakis, Eftychios and Irving, Geoffrey and Fedkiw, Ronald: Robust quasistatic finite elements and flesh simulation. SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation, (2005).
15. Amy E. Kerdok, Stephane M. Cotin, Mark P. Ottensmeyer, Anna M. Galea, Robert D. Howe and Steven L. Dawson: "Truth cube: Establishing physical standards for soft tissue simulation" Medical Image Analysis, Vol **7** (2003) 283–291
16. de Farias, T.S.M., Almeida, M.W.S, Teixeira, J.M.X., Teichrieb, V. Kelner, J: A High Performance Massively Parallel Approach for Real Time Deformable Body Physics Simulation, Computer Architecture and High Performance Computing, 2008. SBAC-PAD '08. 20th International Symposium. (2008), 45–52.
17. Joselli, Mark and Clua, Esteban and Montenegro, Anselmo Conci, Aura Pagliosa, Paulo: A new physics engine with automatic process distribution between CPU-GPU, Sandbox '08: Proceedings of the 2008 ACM SIGGRAPH symposium on Video games. (2008), (149–156).
18. Coumans, E: Bullet physics library, 2009, www.bulletphysics.com
19. Fedkiw, R. and Stam, J. and Jensen, HW: PhysBAM. http://physbam.stanford.edu.
20. Melek, Zeki and Keyser, John: Multi-representation interaction for physically based modeling. SPM '05: Proceedings of the 2005 ACM symposium on Solid and physical modeling **2005**, (187–196),ACM.