Pacific Graphics 2013
B. Levy, X. Tong, and K. Yin
(Guest Editors)

Volume 32 (2013), Number 7

A GPU-based Streaming Algorithm for High-Resolution Cloth Simulation

Min Tang[1], Ruofeng Tong[1], Rahul Narain[3], Chang Meng[1] and Dinesh Manocha[2],

http://gamma.cs.unc.edu/gcloth/

[1]Zhejiang University, China
[2]University of North Carolina at Chapel Hill, USA
[3]University of California, Berkeley, USA

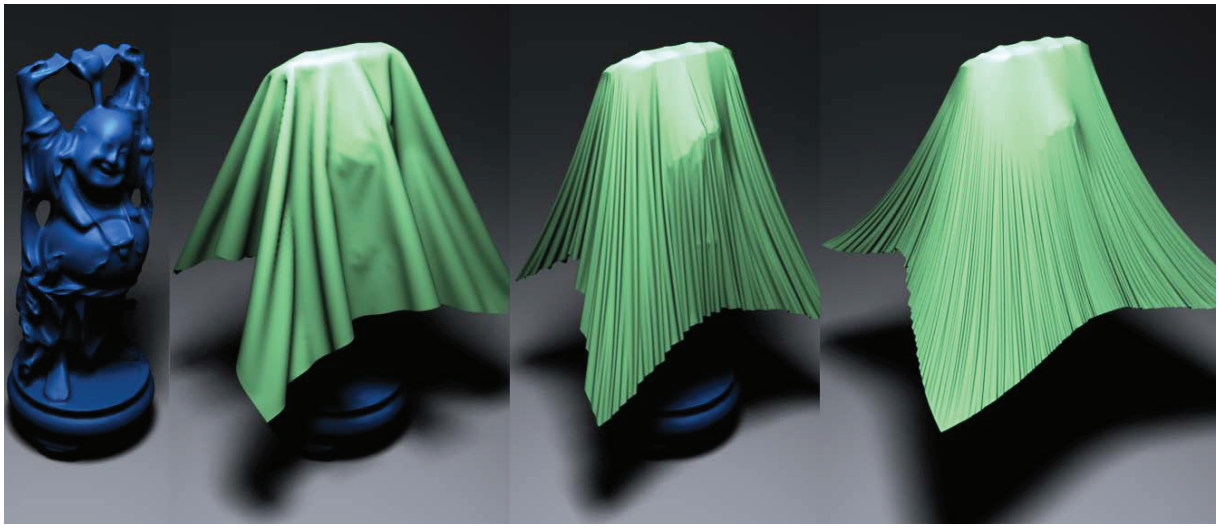


Figure 1: ***Different cloth simulations generated by varying the underlying resolution:*** *The figure highlights different simulation results generated using varying resolutions of the cloth mesh on the Buddha model:* 20*K*, 500*K*, *and* 2*M triangles (from left to right). Our new GPU-based streaming algorithm takes* 138 *seconds/frame to perform the entire simulation (including time integration, collision detection, and response) on a NVIDIA Tesla K20c GPU. It is about* 126*X faster than a single-threaded CPU-based algorithm.*

Abstract

We present a GPU-based streaming algorithm to perform high-resolution and accurate cloth simulation. We map all the components of cloth simulation pipeline, including time integration, collision detection, collision response, and velocity updating to GPU-based kernels and data structures. Our algorithm perform intra-object and inter-object collisions, handles contacts and friction, and is able to accurately simulate folds and wrinkles. We describe the streaming pipeline and address many issues in terms of obtaining high throughput on many-core GPUs. In practice, our algorithm can perform high-fidelity simulation on a cloth mesh with 2*M triangles using* 3*GB of GPU memory. We highlight the parallel performance of our algorithm on three different generations of GPUs. On a high-end NVIDIA Tesla K20c, we observe up to two orders of magnitude performance improvement as compared to a single-threaded CPU-based algorithm, and about one order of magnitude improvement over a 16-core CPU-based parallel implementation.*

1. Introduction

Cloth simulation is important for many applications, including video games, virtual environment, and fashion/garment design. This problem has received considerable attention in computer graphics and related areas for more than a decade [BW98, BFA02, CK05, SSIF09, FYK10, WHRO10, LYO*10, NSO12], though direct physically-based simulation of cloth remains limited to off-line applications such as cinematic animation. Recent trends have been on improving the performance of such techniques for high-quality simulation using higher-resolution [SSIF09] or adaptive meshes [LYO*10, NSO12].

There is considerable interest in performing fast cloth simulation for different applications, including video games and virtual environments. The most recent such techniques use reduced cloth representations based on pre-computed data or simplified physical models [FYK10, WHRO10, MC10, KKN*13]. However, these techniques are not as general as direct physically-based simulation: they are either limited to the space of the training data, or produce visually coarse results or other artifacts in highly dynamic scenarios. As a result, there is still value in developing high-quality physically-based simulation algorithms for interactive applications.

Motivation: Our goal is to develop efficient parallel algorithms for higher-fidelity cloth simulation. Furthermore, we want to develop general approaches that are applicable to a wide variety of scenarios. A key bottleneck in cloth simulation is accurate collision handing, i.e. checking for collisions between the mesh elements and computing responses. High-quality simulation often requires modeling cloth with tens or hundreds of thousands of mass particles. This combinatorial complexity leads to many collisions and numerous primitive pairs in close proximity. It is important to accurately detect all interferences, including self-collisions and collisions between the cloth and other objects. Even a single missed collision can result in an invalid simulation and noticeable visual artifacts, such as cloth passing through itself [VT94, BFA02, BWK03]. This is especially a challenge for high-resolution cloth simulation. As stated by [SSIF09], most cloth simulation techniques would fail if the mesh resolution was increased due to two problems: robustness and tractability. These problems typically manifest themselves in time integration and self-collisions or collision handling.

In this paper, we address the problem of developing highly parallel algorithms for high-resolution simulations. Our goal is to perform accurate simulation and not sacrifice the fidelity, and achieve higher performance by exploiting the SIMD capabilities and multiple cores of a GPU. There is some prior work on using GPUs for only collision detection [TMLT11] or some parts of cloth simulation [Gre04, Zel06], but our goal is to perform all the steps of the simulation on a high-end discrete GPU and avoid any data transfer between the CPU and the GPU. Ideally, we desire a technique that can scale with the increased parallelism or number of cores of a GPU.

Main Results: We present a GPU-based streaming algorithm to perform high-resolution cloth simulation. Our formulation includes a streaming pipeline for cloth simulation that maps all the components, including time integration, collision checking, response force computation, and velocity computation to GPUs based on appropriate geometric and topological data structures. We use an optimized sparse matrix representation based on the topological connectivity between cloth particles. With this representation, we reduce the memory overhead by 50%, and make it possible to use implicit time integration for cloth simulation on commodity GPUs. By handling both inter-object and intra-object collisions, contacts, and friction accurately with GPU stream data and kernels, our algorithm can simulate highly detailed folds and wrinkles.

We have evaluated our algorithm on several complex benchmarks, with different cloth mesh resolutions (with 20K, 500K, and 2M triangles, respectively). We highlight our speedups over CPU-based algorithms: $100-120$X faster compared to a single threaded CPU-based implementation; $10-14$X faster compared to a 16-core CPU-based implementation on an NVIDIA Tesla K20c GPU. As compared to prior GPU-based approaches, our algorithm can simulate accurate and high-resolution meshes (e.g. up to 2M triangles).

Organization: The rest of the paper is organized as follows. We give a brief overview of prior work in Section 2. Section 3 introduces our notation and presents our streaming algorithm. We highlight its performance on different benchmarks in Section 4 and compare with prior algorithms in Section 5.

2. Related Work

In this section, we give a brief overview of related work on cloth simulation and collision handling. We also highlight some recent GPU-based approaches in this field.

2.1. Cloth Simulation

Cloth simulation has been extensively studied by the computer graphics community. Traditionally, continuum models [TPBF87], energy-based particle systems [BHG92], and mass-spring models [VT94, Pro95] have been used for cloth representation. Implicit time integration methods [BW98, CK02] have demonstrated improved stability over explicit time integration. Keckeisen et al. [KSFS03] showed that interactive cloth simulation can be performed (at a lower fidelity) for virtual reality applications. Cordier and Magnenat-Thalmann [CMT05]

used data-driven approach for realtime cloth simulation. Volino et al. [VLMT08] gave a nice overview of various virtual cloth techniques. Techniques to perform high-resolution cloth simulation have been proposed by Selle et al. [SSIF09] and used multi-core CPUs. Recently, adaptive meshes have been used [LYO*10, NSO12] for multi-resolution cloth simulation.

2.2. Collision Handling

Collision handling (collision detection and response) is regarded as the major bottleneck in high-resolution cloth simulation, because of detecting and handling a high number of collisions, including self-penetrations. BVHs (bounding volume hierarchies) [VT94, Pro97] or spatial subdivision methods [HB00] have been used as acceleration structures. Provot [Pro97] introduced the concept of normal cones for self-intersection culling. The idea has been further extended to continuous collision detection [TCYM09, HSK*10]. Bridson et al. [BFA02] proposed an impulse-based collision handling algorithm, which has been extensively used in many cloth simulation systems. Choi and Ko [CK05] stressed the importance of accurate collision detection and handling in cloth simulation. Tang et al. proposed non-penetration filters both for triangle meshes [TMT10a, DTT12] and volume meshes [TMY*11]. Brochu et al. [BEB12] designed a geometrically exact algorithm for robust and efficient continuous collision detection.

2.3. GPU Acceleration

With the advent of programmable GPUs, researchers are utilizing the parallel processing power to speed up cloth simulation and collision handling. In 2004, Green [Gre04] demonstrated fast cloth simulation on a GPU using Verlet integration [Tho03], but could only handle collision detection with a sphere and self-collisions were not taken into account. Zeller [Zel06] presented a similar GPU-based algorithm, but with improved collision detection and responsiveness to wind. Contrary to the aforementioned methods, which handle collision detection in object space, image based methods have been employed [BW04, RNS06]. In 2009, ATI demonstrated cloth simulation results with Havok on the GPU using OpenCL API [Sea09]. Many GPU-based efficient algorithms have been proposed for collision and proximity computations between deformable models [GKJ*05, SGG*06]. Tang et al. [TMLT11] proposed a streaming algorithm to accelerate the process of accurate collision detection by using fine-grained front-based decomposition [TMT10b].

3. Streaming Cloth Simulation

In this section, we introduce our notation and present our streaming algorithm for cloth simulation.

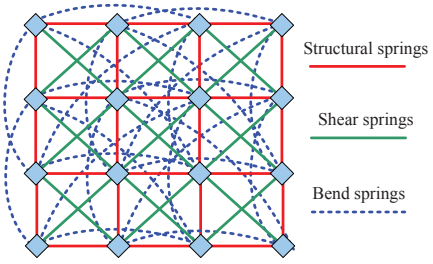


Figure 2: ***Physical Model:*** *Three different springs are used to model the internal forces between cloth particles: structural springs, shear springs, and bend springs.*

3.1. Physical Model

We use the classic mass-spring model [VT94, Pro95] to represent the cloth. Three different springs are used to model the internal forces between cloth particles: structural springs, shear springs, and bend springs (as illustrated by Figure 2).

3.2. Algorithm Overview

During each time step, our iterative cloth simulation algorithm performs two major stages: time integration and collision handling. During the stage corresponding to time integration, all the cloth particles are evolved under external forces (wind forces, gravity, etc.) and internal forces (structural forces, bend forces, stretch forces, etc.). The collision handling stage involves collision detection between the cloth model and other objects, as well as self-collisions in the cloth, and collision response. The collisions are computed based on proximity queries and continuous collision detection followed by impulse forces for all the particles that are involved in collisions. Finally, these impulses are used to update the velocities and positions of the particles.

The collision handling stage first finds all the collisions using proximity queries and continuous collision detection, then calculates the impulses for all the particles involved in the collisions. Finally, these impulses will change the velocities and consequently determine the final positions.

3.3. Streaming Pipeline

In order to fully utilize the parallel capabilities of current GPUs, we map the geometric mesh and other information (e.g., velocities of the particles) to streaming data representation and the computational procedures to GPU kernels. As illustrated in Figure 3, we use the following streaming data representations:

- **Position stream & velocity stream:** These streams describe the current state of the cloth under simulation. We pack the positions and the velocities of the cloth particles into these two streams.

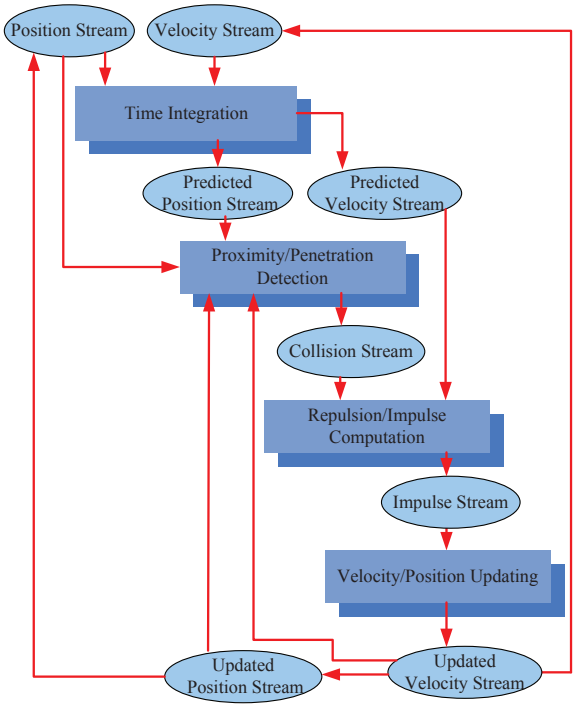


Figure 3: ***Streaming Data and Kernels:*** *All the geometric data are represented as GPU streams (shown in elliptical shapes). Different kernels (shown in rectangular boxes) operate on these streams. The arrows refer to input/output relationship between these streams and kernels.*

- **Predicted position stream & predicted velocity stream:** These streams contain the position and velocity information for the particles evolved under time integration. These streams are computed by only considering interior and exterior forces applied on the particles (i.e., ignoring inter-object and intro-object collisions).
- **Collision stream:** The stream contains the information about inter-object and intra-object collisions (VF/EE pairs, first time of contact, etc.) This collision information will be used to generate impulse stream data.
- **Impulse stream:** The stream contains the impulses for the particles involved in the collisions.
- **Updated velocity stream & updated position stream:** The update velocity stream is generated by updating the predicted velocity streams with the impulse stream. The updated position stream is consequently updated based on the updated velocity stream.

We also use some auxiliary streams corresponding to some geometric and topological data that is needed for collision checking, such as streaming data for connectivity between the triangle meshes, the BVH, the BVTT front, orphan set, etc. Please refer to [TMLT11] for more details.

3.4. Kernel Computations

The various computational stages corresponding to time integration, collision detection, impulse computation, and velocity update are abstracted as GPU kernels and performed in parallel on the GPU cores or streaming units.

- **Time integration kernel:** We support both explicit and implicit time integration on the GPU, and get predicated velocity stream and position stream based on interior forces and external forces that are applied to the mesh particles during this kernel computation.
- **Proximity/penetration detection kernel:** We extend the streaming algorithm originally proposed by Tang et al. [TMLT11] to support proximity detection and penetration detection for high-resolution cloth meshes. The main extension is in terms of reducing the memory overhead of collision-stream algorithm by storing the BVTT (bounding volume traversal tree) front into GPU memory.
- **Repulsion/impulse computation kernel:** After getting collision results (a list of overlapping VF/EE features), we compute the impulses caused by these contacts. For a VF collision pair, the impulse I_{vf} between them can be expressed as:

$$I_{vf} = k(v_r \cdot \hat{n})\hat{n}, \tag{1}$$

where v_r is the relative velocity between the VF pair, $\hat{n}$ is the normal vector of the face F, and k is a stiffness factor. We use a similar formulation to compute impulses between EE pairs.

- **Velocity updating kernel:** The predicted velocities are updated based on the impulses in this kernel (Eq. 2). We also compute the updated positions of mesh particles by using the updated velocities (Eq. 3).

$$v_u = v_p + \frac{I_v}{m_v}, \tag{2}$$

$$p_u = p_i + v_u \Delta t, \tag{3}$$

where v_u and p_u are the updated velocity and updated position of a cloth particle. v_p is the predicted velocity, I_v is the accumulated impulse on this particle, and m_v is the mass of the particle. p_i is the initial position at the beginning of the current time step, and Δt is the length of the time step.

3.5. Time Integration

We support both explicit and implicit time integration on the GPU. For explicit time integration, we used 4th-order Runge-Kutta time integration. Each cloth particle can be updated independently based on its own position and velocity. For implicit time integration, we support both the implicit method proposed by Baraff and Witkin [BW98] and the semi-implicit method proposed by Choi and Ko [CK02].

Explicit Time Integration: Explicit time integration

methods (Euler integration, midpoint integration, Verlet integration, 4 ordered Runge-Kutta integration, etc.) are relatively easy for GPU acceleration. Since each particle computation is performed independently, it is quite straightforward for executing in parallel on GPUs. We used 4th-order Runge-Kutta integration, since it exhibits good stability even for high-resolution cloth simulation (e.g., 2M triangles). To perform time integration, we collect all the external forces f_e and internal forces f_i (by processing all the springs between particles), and apply those forces to each particle p^i to get predicted velocities v_p^i and positions p_p^i in parallel.

One major drawback of explicit time integration schemes is that they are used with small time steps (comparing with implicit time integration). However, we don't use large time steps in order to not miss any collisions.

Implicit & Semi-Implicit Time Integration: For implicit and semi-implicit time integration, we construct a linear system of equations and then solve the linear system with a preconditioned conjugate gradient (PCG) solver. The system matrix corresponding to this linear system is computed on the GPU by computing the Jacobian matrices corresponding to the external and internal forces in parallel. The system matrix is typically a sparse matrix. Due to the regularity of the rectangular cloth, each particle has 16 springs attached to it. So we only need to store the 3×3 matrices on 17 diagonal lines (16 plus the principal diagonal line), as shown in Figure 4.

In order to store this sparse matrix compactly, we use a novel matrix representation, called compressed diagonal format (CDF). As shown in Figure 5(a), all the nodes on the 17 diagonal lines are matrices of size 3×3. Conventional diagonal format (Figure 5(b)) needs additional 40% space (to store the values in green). Instead, with our compressed diagonal format (Figure 5(c)), we use a compact representation for the system matrix. The system matrix in dense format will need to store $9 \times N^2$ floating-point values (where N is the number of particles). Instead, our method only needs to store $17 \times 9N$ floating-point values. Compared to other compressed data structures for sparse matrix, such as compressed sparse row format (CSR) or compressed sparse column format (CSC) [BG13, NVI13b], our representation does not need to store all the index data, and consequently reduces the total memory footprint by 50%. For a cloth with 2M triangles, a system matrix in compressed diagonal format will need 593MB GPU memory to store it.

Our CDF representation can also be used for cloth with cuts and holes that is not perfectly rectangular. For these cases, some particles and springs are missing (i.e. have 0 entries). The resulting CDF may not provide an optimal representation, but can still be used in the GPU-based algorithm.

We do not used the standard library such as CUSP [BG13] or cuSPARSE [NVI13b], since we found that our implementation uses much less storage space, and makes it feasible to perform high-resolution cloth simulation.

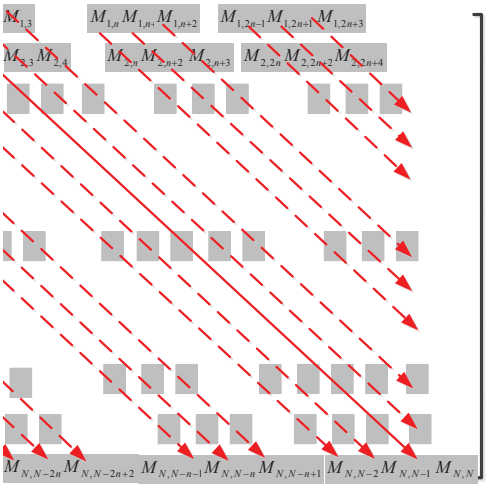


Figure 4: ***System Matrix:*** *Due to the regularity of the rectangular cloth, each particle has only 16 springs linked. So we only need to store the matrices on these 17 diagonal lines (16 plus the principal diagonal line).*

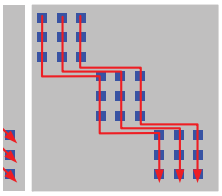


Figure 5: ***Compressed Diagonal Format:*** *To store all the values on the diagonal lines of the system matrix (a), conventional diagonal format will need additional* 40% *space (to store those values in green). Our novel compressed diagonal format (c) provides a compact storage.*

To solve the linear system, we implemented a PCG solver with the optimized matrix representation, and used the cuBLAS [NVI13a] library for efficient multiple operations between large vectors.

3.6. Collision Handling

The basic idea of our collision handling approach is inspired by Bridson's work [BFA02]. We first compute proximity repulsions by performing proximity detection with discrete collision detection (DCD), and apply these repulsions to handle penetrating features. Then we perform penetration detection with continuous collision detection (CCD) and compute the first time of contact along those

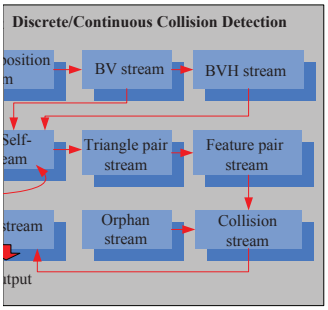


Figure 6: ***Collision Handling:*** *By perform proximity detection and penetration detection (iteratively) on the GPU, repulsion-based and impulse-based methods are used to handle inter-object and intra-object collisions (left). We use a streaming algorithm for discrete/continous collision detection (right). All the geometric data and acceleration data structures (i.e., the bounding volumes and the bounding volume hierarchies) are represented as GPU streams. By updating the BVTT fronts (for inter-object collisions) and self-fronts (for intra-object collisions) incrementally, collision information is used to generate impulse streams.*

penetrating features by performing elementary tests. The first-time-of-contact information is computed using CCDs and used to estimate the collision impulses. These impulses are applied to push back the particles and maintain a penetration-free state. This process is applied iteratively till all penetrations are resolved (as shown in Figure 6).

We extend the basic GPU-based collision detection algorithm originally proposed by Tang et al. [TMLT11] to perform collision handling. We first update the bounding volume stream and bounding volume hierarchy stream by using the predicted position stream which is updated by the time integration stage. We update the BVTT front stream and BVTT self-front stream in parallel on the GPU. These two front streams correspond to inter-object and intra-object collision information, respectively.

A major challenge in collision detection for high-resolution cloth is to overcome the memory overhead of storing the BVTT-based front in the GPU memory. In order to reduce the memory overhead on the GPUs, we use deferred BVTT fronts [TMLT11] to significantly reduce the memory footprint. In practice, we can reduce the overall memory overhead by 80%. We also used the same BVTT front stream for both proximity detection and penetration detection to further reduce the memory overhead. After getting collision results, we compute the impulses based on these collisions. Then the predicted velocities are updated based on the impulses in this kernel.

Frequency of Collision Query: Since collision detection is the most expensive part, some researchers [SSIF09] make the optimization of reducing the frequency of collision detection, i.e., performing one step of collision detection after several steps of time integration. However, we find this method can lead to worse performance, since one step of CCD between large time steps is even more expensive than several steps of CCD between small time steps. So we use the same frequency for both time integration and collision handling to make the simulation tractable.

GPU	GeForce GTX 580	GeForce GTX 680	Tesla K20c
Number of Cores	512	1536	2496
Memory Capacity (G)	1.5	4.0	4.0
Memory Clock Rate (MHz)	2004	3105	2600
GPU Clock Rate (MHz)	1544	1163	706

Figure 7: ***GPUs:*** *Three different GPUs, a NVIDIA GeForce GTX 580, a NVIDIA GeForce GTX 680, and a NVIDIA Telsa K20c, are used for testing our cloth simulation algorithm.*

4. Results

In this section, we describe our implementation and highlight the performance of our algorithm on several benchmarks.

4.1. Implementation

We have implemented our algorithm on three different commodity GPUs: a NVIDIA GeForce GTX 580, a NVIDIA GeForce GTX 680, and a NVIDIA Tesla K20c. Their parameters are shown in Figure 7. For all these NVIDIA GPUs, we used CUDA toolkit 4.2/Visual Studio 2008 as the development environment. We use a standard PC (Windows 7 Ultimate 64-bit/Intel I7 CPU@3.5GHz/8G RAM) as the testing environment.

Currently, we use a fixed time step (i.e., $\frac{1}{60}s$) for both implicit and explicit time integration algorithms. Although the implicit time integration provides stable behavior for large time steps, we still need to use small time steps to handle large, complex models (e.g. 2M triangles).

4.2. Benchmarks

In order to test the performance of our algorithm, we used three different benchmarks. Each benchmark is tested with three different resolutions of cloth (20K triangles, 500K triangles, and 2M triangles):

- **Budda:** A cloth falls on the top of a Buddha statue (100K triangles, 50K vertices, Figure 1).

Resolution (triangles)	Benchmarks	GTX 580 (s/frame)	GTX 680 (s/frame)	Tesla K20c (s/frame)	CPU (s/frame)
20k	Buddha	4.50	5.40	3.10	135.00
20k	Lion	4.20	4.60	3.10	132.00
20k	Dragon	4.74	5.80	3.40	154.00
500k	Buddha	49.80	58.38	38.92	2245.00
500k	Lion	62.31	78.94	52.72	2858.00
500k	Dragon	55.86	63.81	42.17	2408.00
2M	Buddha	N/A	280.27	137.94	17360.00
2M	Lion	N/A	332.64	156.87	20246.00
2M	Dragon	N/A	287.81	141.26	18542.00

Figure 8: ***Performance Results:*** *This figure shows the average running time (with totally 2K frames) of our algorithm on the three different generations of GPUs with varying number of cores.*

Mesh Resolution	Benchmarks	Explicit Integration	Implicit Integration
20k	Buddha	0.75G	0.92G
20k	Lion	0.72G	0.91G
20k	Dragon	0.73G	0.92G
500k	Buddha	1.07G	1.35G
500k	Lion	0.91G	1.31G
500k	Dragon	1.05G	1.32G
2M	Buddha	1.7G	3.03G
2M	Lion	1.62G	2.98G
2M	Dragon	1.68G	3.01G

Figure 9: ***Memory Footprint:*** *This figure shows the memory footprint for all the benchmarks (with different resolutions and with explicit/implicit integration).*

- **Lion:** A cloth falls on the top of a Chinese lion statue (20K triangles, 10K vertices, Figure 13).
- **Dragon:** A cloth falls on the top of a Chinese dragon statue (100K triangles, 50K vertices, Figure 14).

For all the benchmarks, our algorithm can handling inter-object and intra-object accurately and robustly. Furthermore, these are complex scenarios with high number of self-collisions and wrinkles.

4.3. Performance

Figure 8 highlights the performance of our algorithm on different benchmarks. These results show that our streaming cloth simulation algorithm works well on different GPU architectures. The relative performance of a benchmark appears to be proportional to the number of streaming units or cores on different GPUs. This indicates that our algorithm can exploit the large scale parallel capabilities of modern GPUs.

Figure 9 shows the memory footprint for all the benchmarks, and Figure 10 shows the memory occupation rates of every part for the Buddha benchmark with different cloth simulation and with explicit/implict integration. For implicit integration, up to 50% for memory are used by the linear system solver to store the system matrix and other parameter vectors (at the 2M mesh resolution).

Figure 11 shows running time ratios for each stage i.e., Time Integration, Proximity Detection, and Penetration Detection). Figure 12 compares the performance between GPU-based implementation (GTX for NVIDIA GeForce GTX 680, and Tesla for NVIDIA Tesla K20c) and CPU-based single threaded implementation. As we use a higher resolution model, we observe improved computational intensity on GPUs and this leads to higher speedups. On the NVIDIA Tesla K20c, we achieve up to 120X acceleration for cloth with 2M triangles.

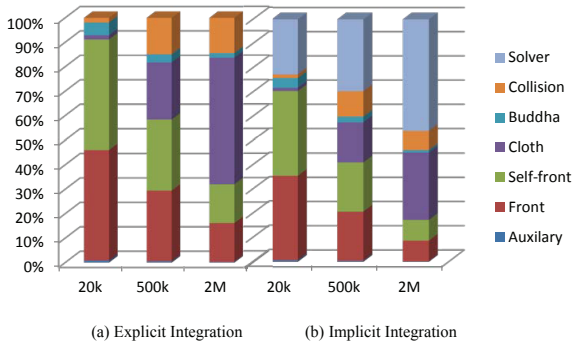
5. Analysis & Comparisons

In this section, we compare our algorithm with prior CPU-based and GPU-based algorithms and highlight some of the benefits.

5.1. Comparisons

In this section, we compare the performance of our approach with prior parallel and GPU-based algorithms.

- **CPU & multi-core based algorithms:** As compared to CPU-based algorithms [CK02, BW98, SSIF09], we obtain better performance based on exploiting the GPU parallelism. For example, some cloth benchmarks with 2M triangles in [SSIF09] takes $20-40$ minutes/frame using 16 CPU-cores, while our algorithm only need $2-3$ minutes/frame on the Tesla K20c GPU. It is approximately $10-12$X faster.
- **CPU-GPU hybrid algorithms:** These methods [PKS10] transfer the data between the CPU and the GPU frequently, while our algorithm is a purely GPU-based method and avoids such data transfers. For examples, at each time step, 2M triangles need to be transferred between GPU and CPU. Also, by performing time integration and collision handling on the same GPU platform, the data inconsistency between CPU and GPU are avoided.
- **GPU based algorithms:** These GPU-based methods [Sea09, Zel06, Gre04] demonstrate good performance, but do not perform accurate collision detection and handling. As a result, they may not be able to perform high-fidelity cloth simulation. The GPU-based algorithm in [TMLT11] is limited to performing accurate collision detection. It has been applied to models with tens of thousands of triangles. In contrast, our algorithm performs time integration, collision detection, and collision handling and can handle models with millions of triangles.



(a) Explicit Integration (b) Implicit Integration

Figure 10: ***Memory Occupation Rates:*** *This figure shows the memory occupation rates of every part for the Buddha benchmark with explicit/implict integration (Solver - storage for linear system solver, Collision - storage for collision information, Buddha - storage for the Buddha model, Cloth -storage for the cloth, Self-front and front - storage for the BVTT fronts, Auxiliary - storage for other data structures).*

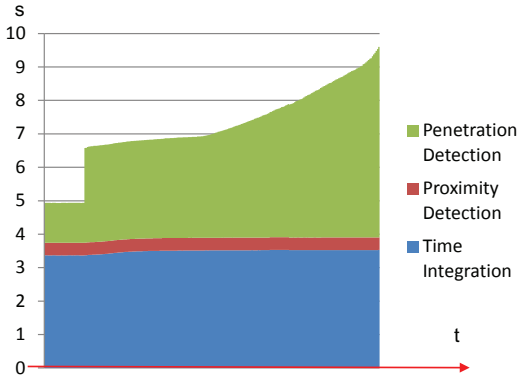


Figure 11: ***Running Time Ratios:*** *This figure highlights the timing breakdown for different stages of the algorithm (for the Buddha benchmark with 2M triangles).*

5.2. Limitations

Our approach does have some limitations, including:

- For some complex scenarios, repulsion-based and impulse-based collision handling methods can not resolve all the penetrations. Some more sophisticated methods, such as untangling (by side voting), or impact zones, can be used. This is a good avenue for future work.
- Our optimized sparse matrix representation is specially designed for rectangular meshes. In the future, we hope to generalize this representation to support cloth models represented by triangle meshes.
- Currently, our time integration and collision handling algorithms do not use the shared memory on GPUs.

	20K			500K			2M		
	Buddha	Lion	Dragon	Buddha	Lion	Dragon	Buddha	Lion	Dragon
GTX	25	29	27	38	36	38	62	61	64
Tesla	44	43	45	58	54	57	126	129	131

Figure 12: ***GPU v.s. CPU Performance:*** *This figure(a) shows acceleration rates by comparing our GPU implementation with a single-threaded CPU implementation (GTX for NVIDIA GeForce GTX 680, and Tesla for NVIDIA Tesla K20c). We demonstrate the speedup for different resolutions of cloth mesh. As we use a higher resolution model, we observe improved computational intensity on GPUs and this leads to higher speedups.*

6. Conclusion and Future Work

We present a GPU-based streaming cloth simulation algorithm for efficient high quality cloth simulation. Our approach is designed for high performance as it maps the data and various computations in terms of appropriate streams and kernels. We also present parallel algorithms for time integration and collision handling and all these computations are performed on the GPU. Moreover, our approach is flexible and maps well to current GPU architectures in terms of memory hierarchy. In practice, our algorithm can improve the performance of cloth simulation on current GPU architectures. We observe significant speedups over CPU-based multi-core algorithm, and are able to perform accurate and higher fidelity simulation as compared to prior GPU-based algorithms.

There are many avenues for future work. First, we will work on solving some limitations of our current approach. We also believe that we can further improve the performance of our algorithm by exploiting more parallelism and memory hierarchy of GPUs. Also, the cloth simulation algorithm can inspire some related simulations, such as rigid body/deformble body simulations. Finally, we would like to extend the algorithm to support adaptive meshes for better simulation and improved performance.

Acknowledgements:

This research is supported in part by NSFC (61170140), the National Basic Research Program of China (2011CB302205), the National Key Technology R&D Program of China (2012BAD35B01), and NVIDIA. Dinesh Manocha is supported in part by ARO Contract W911NF-10-1-0506, NSF awards 0917040, 0904990, 1000579 and 1117127, and Intel. Ruofeng Tong is partly supported by NSFC (61170141). Rahul Narain is supported by NSF Grant IIS-0915462 and funding from Intel Science and Technology Center for Visual Computing.

References

[BEB12] Tyson Brochu, Essex Edwards, and Robert Bridson. Efficient geometrically exact continuous collision detection. *ACM Trans. Graph.*, 31(4):96:1–96:7, July 2012. 3

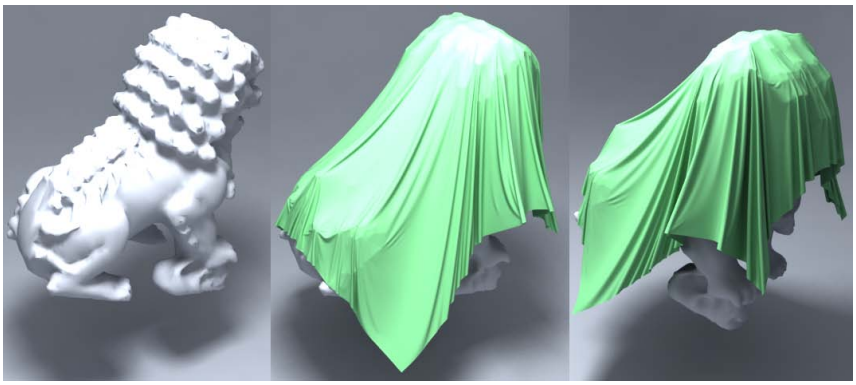


Figure 13: ***Dropping cloth on a lion:*** *A cloth falls on the top of a Chinese lion statue. The lion model consists of* 20*K triangles, and the cloth consists of* 2*M triangles. On a NVIDIA Tesla K20c GPU, we perform simulation at* 157*s/frame, which is approximately* 129*X faster than a single-threaded CPU algorithm.*

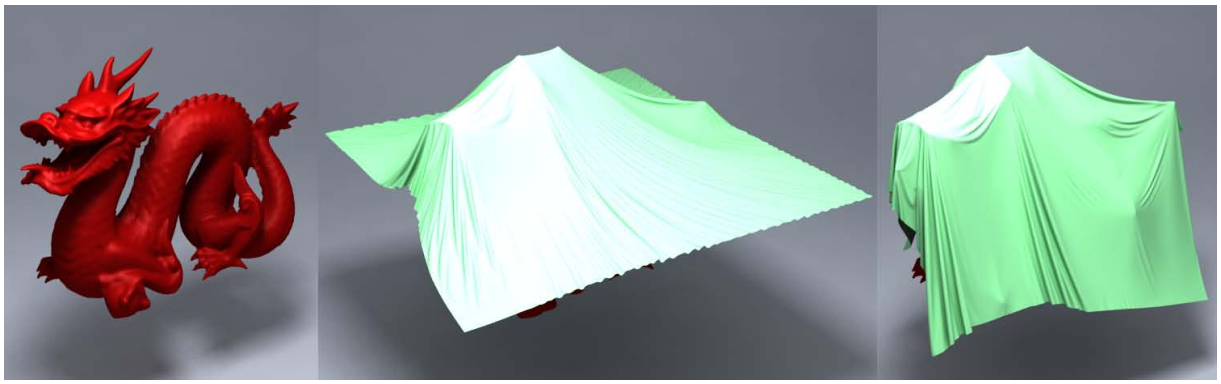


Figure 14: ***Dropping cloth on a dragon:*** *A cloth falls on the top of a Chinese dragon statue. The dragon model consists of* 100*K triangles, and the cloth consists of* 2*M triangles. On a NVIDIA Tesla K20c GPU, we perform simulation at* 141*s/frame, which is approximately* 131*X faster than a single-threaded CPU algorithm.*

[BFA02] Robert Bridson, Ronald Fedkiw, and John Anderson. Robust treatment of collisions, contact and friction for cloth animation. *ACM Trans. Graph.*, 21(3):594–603, July 2002. 2, 3, 5

[BG13] Nathan Bell and Michael Garland. CUSP: A C++ Templated Sparse Matrix Library, http://cusplibrary.github.io/, 2013. 5

[BHG92] David E. Breen, Donald H. House, and Phillip H. Getto. A physically-based particle model of woven cloth. *The Visual Computer*, 8:264–277, 1992. 2

[BW98] David Baraff and Andrew Witkin. Large steps in cloth simulation. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '98, pages 43–54, New York, NY, USA, 1998. ACM. 2, 4, 7

[BW04] George Baciu and Wingo Sai-Keung Wong. Image-based collision detection for deformable cloth models. *IEEE Transactions on Visualization and Computer Graphics*, 10(6):649–663, 11-12 2004. 3

[BWK03] David Baraff, Andrew Witkin, and Michael Kass. Untangling cloth. *ACM Trans. Graph.*, 22(3):862–870, July 2003. 2

[CK02] Kwang-Jin Choi and Hyeong-Seok Ko. Stable but responsive cloth. *ACM Trans. on Graph.*, 21:604–611, 2002. 2, 4, 7

[CK05] Kwang-Jin Choi and Hyeong-Seok Ko. Research problems in clothing simulation. *Comput. Aided Des.*, 37(6):585–592, May 2005. 2, 3

[CMT05] Frederic Cordier and Nadia Magnenat-Thalmann. A data-driven approach for real-time clothes simulation. *Comput. Graph. Forum*, 24(2):173–183, 2005. 2

[DTT12] Peng Du, Min Tang, and Ruofeng Tong. Fast continuous collision culling with deforming noncollinear filters. *Computer Animation and Virtual Worlds*, 23(3-4):375–383, 2012. 3

[FYK10] Wei-Wen Feng, Yizhou Yu, and Byung-Uck Kim. A deformation transformer for real-time cloth animation. *ACM Trans. Graph.*, 29(4):108:1–108:9, July 2010. 2

[GKJ+05] Naga K. Govindaraju, David Knott, Nitin Jain, Ilknur Kabul, Rasmus Tamstorf, Russell Gayle, Ming C. Lin, and Dinesh Manocha. Interactive collision detection between deformable models using chromatic decomposition. *ACM Trans. Graph.*, 24(3):991–999, July 2005. 3

[Gre04] Simen Green. Nvidia white paper, http://developer.nvidia.com/object/demo_cloth_simulation.html, 2004. 2, 3, 7

[HB00] Donald H. House and David E. Breen, editors. *Cloth Modeling and Animation*. A. K. Peters, Ltd., 2000. 3

[HSK+10] Jae-Pil Heo, Joon-Kyung Seong, DukSu Kim, Miguel A. Otaduy, Jeong-Mo Hong, Min Tang, and Sung-Eui Yoon. FASTCD: Fracturing-aware stable collision detection. In *SCA '10: Proceedings of the 2010 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, 2010. 3

[KKN+13] Doyub Kim, Woojong Koh, Rahul Narain, Kayvon Fatahalian, Adrien Treuille, , and James F. O'Brien. Near-exhaustive precomputation of secondary cloth effects. *ACM Trans. Graph. (Proc. of SIGGRAPH 2013)*, 32(4), 2013. 2

[KSFS03] Michael Keckeisen, Stanislav L. Stoev, Matthias Feurer, and Wolfgang Strasser. Interactive cloth simulation in virtual environments. In *Proceedings of the IEEE Virtual Reality 2003*, pages 71–71, 2003. 2

[LYO+10] Yongjoon Lee, Sung-Eui Yoon, Seungwoo Oh, Duksu Kim, and Sunghee Choi. Multi-resolution cloth simulation. *Computer Graphics Forum*, 29(7):2225–2232, 2010. 2, 3

[MC10] Matthias Müller and Nuttapong Chentanez. Wrinkle meshes. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '10, pages 85–92, 2010. 2

[NSO12] Rahul Narain, Armin Samii, and James F. O'Brien. Adaptive anisotropic remeshing for cloth simulation. *ACM Trans. Graph.*, 31(6):152:1–152:10, November 2012. 2, 3

[NVI13a] NVIDIA. cuBLAS: The NVIDIA CUDA Basic Linear Algebra Subroutines library, https://developer.nvidia.com/cublas, 2013. 5

[NVI13b] NVIDIA. cuSparse: The NVIDIA CUDA Sparse Matrix library, https://developer.nvidia.com/cusparse, 2013. 5

[PKS10] Simon Pabst, Artur Koch, and Wolfgang Straßer. Fast and scalable CPU/GPU collision detection for rigid and deformable surfaces. *Computer Graphics Forum*, 29(5):1605–1612, 2010. 7

[Pro95] Xavier Provot. Deformation constraints in a mass-spring model to describe rigid cloth behavior. In *Proc. of Graphics Interface*, pages 147–154, 1995. 2, 3

[Pro97] X. Provot. Collision and self-collision handling in cloth model dedicated to design garments. In *Graphics Interface*, pages 177–189, 1997. 3

[RNS06] Javier Rodríguez-Navarro and Antonio Susín. Non structured meshes for cloth GPU simulation using FEM. In *VRIPHYS'06*, pages 1–7, 2006. 3

[Sea09] Sean Kalinich. Havok show OpenCL based Havok Cloth on ATI GPUs, http://www.brightsideofnews.com/news/2009/3/27/havok-show-opencl-based-havok-cloth-on-ati-gpus.aspx, 2009. 3, 7

[SGG+06] Avneesh Sud, Naga Govindaraju, Russell Gayle, Ilknur Kabul, and Dinesh Manocha. Fast proximity computation among deformable models using discrete voronoi diagrams. *ACM Trans. Graph.*, 25(3):1144–1153, July 2006. 3

[SSIF09] Andrew Selle, Jonathan Su, Geoffrey Irving, and Ronald Fedkiw. Robust high-resolution cloth using parallelism, history-based collisions, and accurate friction. *IEEE Transactions on Visualization and Computer Graphics*, 15(2):339–350, March 2009. 2, 3, 6, 7

[TCYM09] Min Tang, Sean Curtis, Sung-Eui Yoon, and Dinesh Manocha. ICCD: interactive continuous collision detection between deformable models using connectivity-based culling. *IEEE Transactions on Visualization and Computer Graphics*, 15:544–557, 2009. 3

[Tho03] Thomas Jakobsen. Advanced character physics, http://www.gamasutra.com/resource_guide/20030121/jacobson_01.shtml, 2003. 3

[TMLT11] Min Tang, Dinesh Manocha, Jiang Lin, and Ruofeng Tong. Collision-streams: Fast GPU-based collision detection for deformable models. In *I3D '11: Proceedings of the 2011 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 63–70, 2011. 2, 3, 4, 6, 7

[TMT10a] Min Tang, Dinesh Manocha, and Ruofeng Tong. Fast continuous collision detection using deforming non-penetration filters. In *I3D '10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 7–13, New York, NY, USA, 2010. ACM. 3

[TMT10b] Min Tang, Dinesh Manocha, and Ruofeng Tong. MCCD: Multi-core collision detection between deformable models using front-based decomposition. *Graphical Models*, 72(2):7–23, 2010. 3

[TMY+11] Min Tang, Dinesh Manocha, Sung-Eui Yoon, Peng Du, Jae-Pil Heo, and Ruofeng Tong. VolCCD: Fast continuous collision culling between deforming volume meshes. *ACM Trans. Graph.*, 30:111:1–111:15, May 2011. 3

[TPBF87] Demetri Terzopoulos, John Platt, Alan Barr, and Kurt Fleischer. Elastically deformable models. *SIGGRAPH Comput. Graph.*, 21(4):205–214, August 1987. 2

[VLMT08] Pascal Volino, Christiane Luible, and Nadia Magnenat-Thalmann. Virtual clothing. In *Wiley Encyclopedia of Computer Science and Engineering*. 2008. 3

[VT94] P. Volino and N. M. Thalmann. Efficient self-collision detection on smoothly discretized surface animations using geometrical shape regularity. *Computer Graphics Forum*, 13(3):155–166, 1994. 2, 3

[WHRO10] Huamin Wang, Florian Hecht, Ravi Ramamoorthi, and James O'Brien. Example-based wrinkle synthesis for clothing animation. *ACM Trans. Graph.*, 29(4):107:1–107:8, July 2010. 2

[Zel06] Cyril Zeller. *Practical Cloth Simulation on Modern GPU*. Shader X4: Advanced Rendering with DirectX and OpenGL. Charles River Media, 2006. 2, 3, 7