EUROGRAPHICS 2016 / J. Jorge and M. Lin
(Guest Editors)

Volume 35 (2016), Number 2

CAMA: Contact-Aware Matrix Assembly with Unified Collision Handling for GPU-based Cloth Simulation

Min Tang[1], Huamin Wang[3], Le Tang[1], Ruofeng Tong[1], Dinesh Manocha[2,1] †

[1]Zhejiang University, [2]University of North Carolina at Chapel Hill, [3]Ohio State University
http://gamma.cs.unc.edu/CAMA/

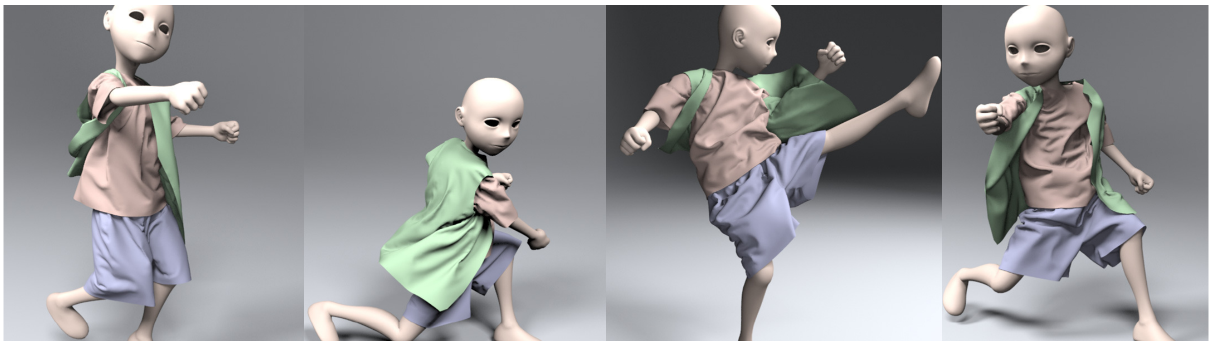

Figure 1: ***Benchmark Andy:*** *Our GPU-based approach can simulate the clothes dressed on a Kung-Fu boy. The meshes of three cloth pieces are represented by* $127K$ *triangles. Our simulator performs all of the computations, including implicit time integration and collision handling, in* $2.42s$ *per frame (on average) on an NVIDIA Telsa K40c GPU. Our new parallel algorithms for sparse matrix assembly and collision handling result in significant speedups over prior methods.*

Abstract

We present a novel GPU-based approach to robustly and efficiently simulate high-resolution and complexly layered cloth. The key component of our formulation is a parallelized matrix assembly algorithm that can quickly build a large and sparse matrix in a compressed format and accurately solve linear systems on GPUs. We also present a fast and integrated solution for parallel collision handling, including collision detection and response computations, which utilizes spatio-temporal coherence. We combine these algorithms as part of a new cloth simulation pipeline that incorporates contact forces into implicit time integration for collision avoidance. The entire pipeline is implemented on GPUs, and we evaluate its performance on complex benchmarks consisting of $100-300K$ *triangles. In practice, our system takes a few seconds to simulate one frame of a complex cloth scene, which represents significant speedups over prior CPU and GPU-based cloth simulation systems.*

1. Introduction and Background

Cloth simulation has been an active research topic for decades due to its importance in electronic games, virtual training systems, and fashion-related applications; based on this activity, a number of algorithms have been proposed. Many animation and authoring systems provide capabilities for cloth simulation and are widely used for computer-aided design and animation.

In particular, research has focused on improving the robustness and efficiency of cloth simulation and has explored the use of implicit Euler integrators [BW98, CK02], strain limiting [Pro95], and iterative optimization [LBOK13]. Given the importance of developing a good cloth simulator, researchers have also explored methods of increasing the details of cloth simulation in a local and adaptive manner [LYO*10, NSO12] or by using data-driven methods [FYK10, WHRO10, dASTH10, KGBS11, ZBO13, KKN*13]. Because collisions and contacts are difficult to handle in cloth simulation, substantial research effort has also been expanded in improving the accuracy of continuous collision de-

† {tang_m, tangle, trf}@zju.edu.cn, whmin@cse.ohio-state.edu, dm@cs.unc.edu

tection [BEB12, Wan14, TTWM14], collision impulses [BFA02], constraint solvers [OTSG09], and impact zone methods [Pro95, HVTG08] for collision response.

Current methods for reliable cloth simulation can be slow, taking tens of seconds per frame on a single CPU core to simulate a mesh with a few tens of thousands of triangles [NSO12]. Recent trends have been focused on simulating more complex clothes (see Fig. 1). The underlying geometric complexity of cloth is characterized by the *mesh resolution* defining each piece of cloth, and the *layered relationship* specifying the likelihood of frequent and severe collisions over time. For high-quality animation, cloth meshes can contain hundreds of thousands of triangles. It should be noted that even a single missed collision can cause invalid results and noticeable visual artifacts [BFA02]. To prevent cloth from falling into inter-penetrations as a result of those self-contacts, most existing simulators use continuous collision detection (CCD) and appropriate collision response computations. Unfortunately, these collision handling techniques are often computationally expensive and can easily become the bottleneck of a simulator. Therefore, a cloth simulator can take hours or even days to simulate highly complex cloth.

Many researchers have advocated the use of multiple cores on commodity CPUs and GPUs to accelerate these computations. Govindaraju et al. [GKJ*05,GLM05], Selle et al. [SSIF09], Pabst et al. [PKS10], Lauterbach et al. [LMM10], and Tang et al. [TMLT11] developed parallelized collision detection (including CCD) and handling algorithms. Li et al. [LWM11] presented a hybrid method to simulate cloth by both CPU and GPU. Schmitt et al. [SKBK13] simulated a low-resolution triangular mesh on CPU and mapped the deformation to a high-resolution cloth mesh on GPU. Cirio et al. [CLMMO14] implemented GPU-based implicit time integration, for simulating quad cloth meshes. Ni et al. [NKT15] used Charm++ for parallel cloth simulation on a Cray XC30. All these algorithms use parallelization techniques to accelerate some of the computations. Tang et al. [TTN*13] presented a GPU-based cloth simulation algorithm that performs all of the computations on GPU, including explicit time integration and CCD computations. However, this approach exploits topological regularity and is limited to quad meshes only and cannot handle complex cloth.

Our goal is to develop a highly parallel and robust GPU-based cloth simulation algorithm. One of the challenges is developing efficient approaches that can provide flexibility in terms of mesh representations, time integration, and collision handling. In addition to supporting triangle meshes, we would like to reduce the overhead of collision computations using repulsion forces [BFA02], which tend to push two elements that are close to each other apart or incorporating penetration constraints into time integration [OTSG09]. In many of these cases, the matrix resulting from time integration does not have a static or fixed layout, as is the case with many prior GPU-based linear system solvers [ACF11, WBS*13]. Matrix reconstruction in a dynamic layout is trivial on CPU, but significantly challenging on GPU because the sparse matrix in compressed format must be updated over time.

Main Results: We present a novel, GPU-based cloth simulation algorithm that exploits GPU parallelism for time integration, collision detection, and collision handling. Our formulation is general and can handle all triangular meshes accurately using a new sparse matrix assembly algorithm. We use discrete collision checking to compute the contact and repulsion forces and incorporate them into implicit time integration. We also use a unified approach for continuous collision detection and response computation that can significantly improve the runtime performance. For the purpose of developing an end-to-end GPU-based cloth simulation system, we have developed many new GPU algorithms including:

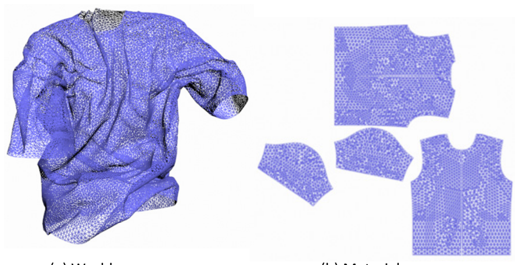

Figure 2: ***Shirt:*** *The shirt model in (a) is represented by a mesh of* 34.6*K triangles and composed of multiple patches in (b).*

- A unified streaming pipeline for time integration and collision handling (Section 2).
- A parallel sparse matrix assembly algorithm that supports implicit time integration of cloth models with arbitrary topology and accurately solving the linear system (Section 3).
- Parallel implicit solver that incorporates contact forces: We present a parallel approach for adding contact forces into time integration. This significantly reduces the number of inter-penetrations and improves the robustness of our approach (Section 3).
- Unified collision handling: We present a parallel, integrated collision detection and response algorithm. Our collision detection uses spatio-temporal coherence and localized propagation, and collision response is performed using inelastic impact zones [HVTG08] (Section 4).

We have implemented these algorithms on different NVIDIA GPUs with varying numbers of cores and used them to simulate complex cloth benchmarks represented by hundreds of thousands of triangles (Section 5). Our parallel algorithm can generate a single cloth animation frame in a few seconds. We observe $10.2-15.4$X speedups in our GPU-based implementation due to our sparse matrix assembly and collision handling algorithms. Moreover, we observe $47-58$X speedups over a single threaded CPU-based implementation, available as part of ArcSim [NSO12].

2. Overview

In this section, we present an overview of our approach and our novel GPU-based pipeline for cloth simulation. Our approach represents each piece of cloth by a triangular mesh, which consists of many patches (Fig. 2). We use a piecewise linear model proposed by Wang et al. [WRO11] to handle planar and bending elasticity. As opposed to prior GPU-based cloth simulation algorithms [TTN*13, CLMMO14], our triangle mesh formulation can be used for arbitrary cloth topologies and can be combined with

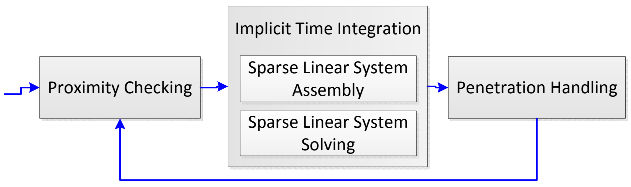
Proximity Checking
Implicit Time Integration
Sparse Linear System Assembly
Sparse Linear System Solving
Penetration Handling

Figure 3: ***Algorithm Pipeline:*** *Our GPU-based cloth simulation pipeline. All of these computations are performed on GPU.*

widely used sparse matrix formats, such as compressed sparse row (CSR) or block compressed sparse row (BCSR), for compact representations and GPU-based linear solvers [NVI15, BG13, WBS*13].

A typical cloth simulation system contains three components: time integration, collision detection, and collision response. Similar to prior approaches, we use implicit time integration as that improves numeric stability. We perform all of the computations on GPU. To reduce the overhead of collision handling, our goal is to employ a scheme that collects all of the contact forces for simulation [OTSG09] – including repulsion forces, friction forces, and adhesion forces – using proximity checking. Next, we combine the internal forces, external forces, and contact forces to form a sparse linear system using implicit time integration. After solving the linear system, we check for penetrations (including self-collisions) in the mesh using continuous collision detection (CCD). We resolve these penetrations to compute a collision-free mesh. The proximity computations are performed using discrete collision detection (DCD), which are significantly faster than CCD queries. While most prior approaches handle the contact forces as part of collision response computation, we incorporate the contact forces into time integration, which results in fewer iterations during collision response computation.

To support these computations, we require an integrated framework that can efficiently perform DCD and CCD computations as well as penetration handling on GPU. We require efficient schemes for sparse matrix assembly to support dynamic layouts because we incorporate the contact forces into implicit time integration.

Cloth Simulation Pipeline: Our novel GPU-based cloth simulation pipeline is shown in Fig. 3 and is different from those used in previous approaches [BFA02, TTN*13]. We use this pipeline to efficiently perform the computations described above. The matrix in the linear system is represented by the BCSR format [NVI15]. We present novel algorithms for sparse linear system assembly, localized collision computations, and integrated collision detection and response computation. A key issue in the design of this pipeline is to specify various *GPU streams* that are used to perform all of the computations on geometric and topological mesh data. In particular, we use the following data streams (Fig. 4):

- **Vertex stream & velocity stream:** These two streams describe the current state of cloth vertices in simulation. They contain vertex positions and vertex velocities, respectively.
- **Updated vertex stream & updated velocity stream:** These streams contain the updated vertex positions and velocities after implicit time integration. They are updated by not only internal and external forces, but also repulsion forces that are used to avoid future collisions.

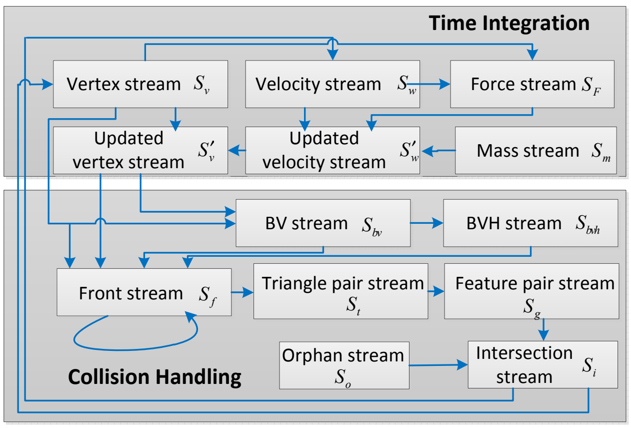

Figure 4: ***Data Streams:*** *We highlight various data streams used by our GPU-based cloth simulation algorithm. We classify them into streams for time integration and collision handling.*

- **BV/BVH/front streams:** These streams contain the data for bounding volumes (BVs), bounding volume hierarchies (BVHs), and bounding volume traversal trees (BVTTs). They are updated by the collision detection module.
- **Intersection stream:** This stream contains the data corresponding to all of the penetrations and it is used to resolve the penetrations by updating vertex and velocity streams. It is updated by the collision response algorithm.

In addition to these streams, we use some other geometric and topological streams to accelerate collision detection. For example, we use connectivity data streams – such as adjacent triangles, neighborhood lists and orphan sets – to efficiently perform elementary tests for CCD [TMLT11]. By converting all of the data elements into data streams in the GPU memory and performing all simulation steps by GPU kernels, we avoid CPU-GPU data transfer during simulation, which improves the overall performance.

3. GPU-accelerated Time Integration

A key function in our approach is to perform implicit time integration during each iteration. Specifically, our GPU-based approach results in a sparse linear system, whose structure and entries vary over time due to different contacts and boundary conditions. We present a novel parallel matrix assembly algorithm that constructs the sparse linear system dynamically and runs fast on GPU. Furthermore, our algorithm can also be used to accurately solve the linear system using a Jacobi preconditioner.

3.1. Implicit Time Integration

Given a triangular mesh with N vertices, we can formulate its dynamical system as follows: $\mathbf{M}\ddot{\mathbf{u}} + \mathbf{D}\dot{\mathbf{u}} = \mathbf{f}(\mathbf{u})$, where $\mathbf{M} \in \mathbf{R}^{3N\times 3N}$ and $\mathbf{D} \in \mathbf{R}^{3N\times 3N}$ are the mass and damping matrices, $\mathbf{u} \in \mathbf{R}^{3N}$ is the stacked displacement vector, and $\mathbf{f}(\mathbf{u})$ is the force vector depending on $\mathbf{u}$ only. Our formulation allows $\mathbf{u}$ to be inte-

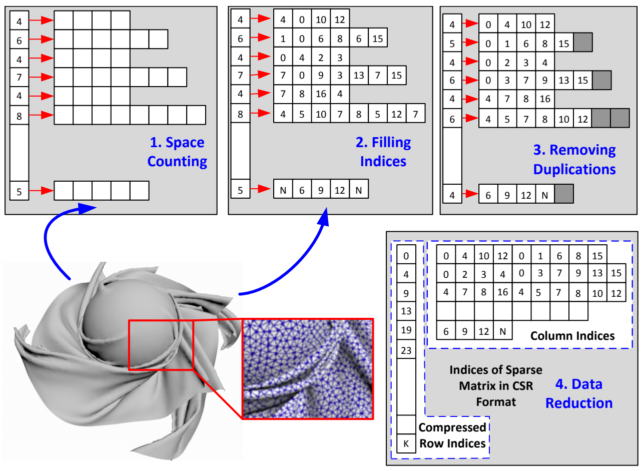

Figure 5: ***CSR Index Assembly:*** *By performing four-stage computations, the CSR index can be assembled fully in parallel.*

grated using various implicit time integrators (e.g. backward Euler method [BW98]). Following the linearization, we approximate $\mathbf{f}^{t+1}$ at time $t+1$ by: $\mathbf{f}(\mathbf{u}^{t+1}) \approx \mathbf{f}(\mathbf{u}^t) + \mathbf{K}(\mathbf{u}^{t+1} - \mathbf{u}^t)$, where $\mathbf{K} = \partial\mathbf{f}(\mathbf{u}^t)/\partial\mathbf{u}$ is the Jacobian matrix of $\mathbf{f}$ evaluated at time t, and obtain:

$$\mathbf{A}\mathbf{v}^{t+1} = (\mathbf{M} + \Delta t\mathbf{D} - \Delta t^2\mathbf{K})\mathbf{v}^{t+1} = \mathbf{M}\mathbf{v}^t + \Delta t\mathbf{f}(\mathbf{u}^t), \qquad (1)$$

where $\mathbf{v}^{t+1}$ is the unknown velocity vector at time $t+1$ that must be solved. If $\mathbf{f}(\mathbf{u})$ contains the elastic and external forces only, the nonzeros in the matrix, $\mathbf{A}$ of Equation 1, correspond to the neighborhood of each vertex, and $\mathbf{A}$ can thus be procedurally defined on top of the mesh connectivity. But because we use $\mathbf{f}(\mathbf{u})$ to handle contact forces and constraints – which can appear anywhere in simulation – the structure of the matrix varies over time and must be rebuilt in each time step. It is straightforward to perform the assembly process on CPU. However, no fast algorithm is known for assembling such a matrix for cloth simulation on GPU.

3.2. Parallel Sparse Matrix Assembly on GPU

We need a general matrix representation suitable for triangle meshes with arbitrary topology. To solve the linear system efficiently on GPU, we store sparse matrices in compressed sparse row (CSR) format. In our approach, we actually use the blocked CSR (BCSR) format, which stores non-zero entries as 3×3 sub-matrices. In this section, we present a novel parallel assembly algorithm for sparse matrices in CSR, and then extend it to BCSR.

Index Assembly: Our formulation first perform the CSR index assembly in four stages (Fig. 5):

- **Space Counting:** We scan over topological elements (triangles/edges) and precompute the memory space needed to store matrix entries. We count the entries in each row. After that, we allocate the memory for each row and use the memory entries later to store the indices. This stage is executed in parallel as the elements can be processed independently. The memory allocation is performed on CPU for efficiency reasons.

Algorithm 1 Sparse Matrix Assembly on GPU.

```
1:  // Space Counting:
2:  for each element $E_i$ do
3:      ColumnCount[$i$] += NumOfEntries($E_i$)
4:  end for
5:  Allocate memory for column indices of size $\sum_i$CoulmneCount[$i$].
6:  // Filling Indices:
7:  for each element $E_i$ do
8:      ColumnIndices[$i$] = { Entries($E_i$) }
9:  end for
10: // Removing Duplications:
11: for each row $r$ do
12:     SortAndRemoveDuplications(ColumnIndices[$r$])
13: end for
14: // Data Reduction:
15: BcsrRowIndex = PrefixSum(ColumnCount)
16: BcsrColIndex = PrefixSum(ColumnIndices)
17: // Data Filling:
18: Allocate memory for values of size of BcsrColIndex.
19: for each element $E_i$ do
20:     for each entry $(s,t,val)$ of $E_i$ do
21:         BcsrValue[location$(s,t)$] += $val$
22:     end for
23: end for
```

- **Filling Indices:** We scan all of the topological elements again. During this pass, we fill the indices into the allocated memory. Note that there can be duplicated indices in several rows. We parallelize this stage for each element.
- **Removing Duplications:** After gathering all of the indices, we use a simple serial merge/sort operator (with an $O(N\log(N))$ time complexity) for each row. All the rows are sorted in parallel.
- **Data Reduction:** We use a prefix sum operator to remove the redundant memory locations (due to the merged indices) and generate compact column index data. We also obtain the compress row index data for counting. These are used as the index data of the sparse matrix in the CSR format.

Data Filling: After generating the index data, the data filling stage is performed during another pass of element scanning. We perform the data filling computations in parallel. For each element, we locate its entry based on the index data and fill in its data using atomic operators (to avoid conflicts).

Extension of BCSR format: We use the same algorithm to perform the index assembly step. The main difference between CSR and BCSR format computations is in data filling. We fill in each entry with a double-precision number or 3×3 matrices in the value data for CSR or BCSR formats, respectively.

The overall matrix assembly algorithm, described in Algorithm 1, exploits GPU architectures by performing the entire computation over thousands of GPU cores. We incorporate all of the forces, including internal forces, external forces, contact and friction forces, into our cloth simulation algorithm, and build a sparse

Algorithm 2 Computing VF Proximity Forces to Update the Linear System: The vertex is defined by a node n_v and the face is defined by three nodes n_a, n_b, and n_c.

```
1: Find the projection of $n_v$ onto the triangle $\{n_a, n_b, n_c\}$ plane.
2: Compute the barycentric coordinates $\{w_1, w_2, w_3\}$ of $n_v$ with
   respect to the triangle.
3: Compute the proximity force $\mathbf{f}_p$ with Equation 2.
4: Distribute the proximity force among node pairs:
   $\{n_v, n_a, w_1 * \mathbf{f}_p\}$, $\{n_a, n_v, -w_1 * \mathbf{f}_p\}$,
   $\{n_v, n_b, w_2 * \mathbf{f}_p\}$, $\{n_b, n_v, -w_2 * \mathbf{f}_p\}$,
   $\{n_v, n_c, w_3 * \mathbf{f}_p\}$, $\{n_c, n_v, -w_3 * \mathbf{f}_p\}$,
   and update the corresponding entries in the linear system.
```

linear system for parallel implicit time integration with a low computation overhead.

VF/EE Proximity Forces: For every close vertex/face or edge-edge pair (computed using proximity checking module in Figure 3), we compute the penalty force:

$$\mathbf{f}_p = k * (d - thickness) * \mathbf{n}_p, \tag{2}$$

where $k, d, thickness, \mathbf{n}_p$ are the stiffness constant, proximity distance between the VF/EE pair, thickness of a given face/edge, normal direction, respectively. The force $\mathbf{f}_p$ is distributed among various nodes [BFA02] and added to the stiffness matrix as multiple entries. For example, for a VF pair, the vertex is defined by a node n_v and the face is defined by three nodes n_a, n_b, and n_c. We use the algorithm described in Algorithm 2 to update the linear system accordingly. EE pairs are processed similarly. Please note that the same entry may be updated by multiple VF or EE pairs, so we need to remove duplications, as shown in Algorithm 1.

Linear System Storage: Each element has a fixed entry number to update. For example, a VF pair needs to update 6 entries; an EE pair needs to update 8 entries. We compute NumOfEntries in Algorithm 1 by summing them. The dimension of the full linear system is $N*N*3*3$, where N is the number of nodes. It is a sparse linear system, so the number of entries stored is much less. For a cloth with 200K triangles, we only need $300-400$MB to store the linear system.

3.3. Solving Sparse Linear System on GPU

We use the standard conjugate gradient (CG) solver [BW98] to solve the sparse linear system on GPU. We use accelerated matrix-vector multiplication operator (between a sparse matrix and a dense vector) and vector-vector dot product operator (between dense vectors) in each iteration. The solver is ended when the error is below a pre-specified convergence threshold. To further improve performance, we use a Jacobi preconditioner, which can be easily integrated into our solver because we record the storage locations of all the diagonal entries of the sparse matrix during the *Index Assembly* stage. With all of these locations, the values of the diagonal entries can be retrieved in parallel after the *Data Filling* stage. Figure 6 shows the benefits of the Jacobi preconditioner. Compared with the CG solver without preconditioner, we are able to reduce the average number of iterations by 32%, and considerably reduce the overall running time of implicit time integration.

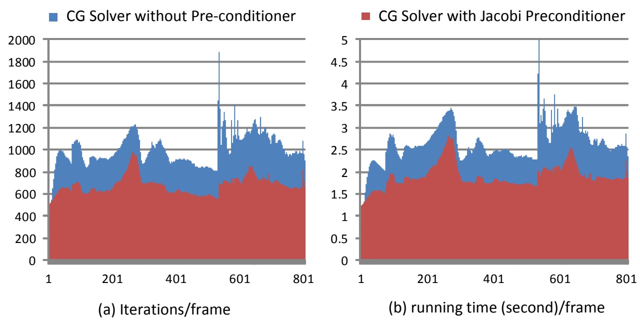

Figure 6: ***Conjugate Gradient with Jacobi Preconditioner:*** *We use our matrix assembly algorithm for PCG solve with Jacobi preconditioner. The use of Jacobi preconditioner reduces the average number of iterations by 32%. The overall running time of implicit time integration is reduced by 28% for the Funnel benchmark.*

4. GPU-accelerated Unified Collision Handling

In this section, we describe our algorithm for GPU-based collision handling, including collision detection and collision response. In general, collision handling for high-resolution cloth simulation can be time consuming and might take up to $60\% - 80\%$ of total running time in challenging scenarios [SSIF09]. In our streaming pipeline, collision detection and collision response computations are tightly integrated (see Fig. 4). In particular, collision detection algorithms are used to compute the contact constraints as well as the penetrations, and a GPU-friendly collision response algorithm is used to handle the penetrations.

4.1. Proximity and Penetration Computations

Prior cloth simulators perform fast and reliable CCD computations for collision detection [BEB12, Wan14, TTWM14, WTTM15]. Moreover, techniques have also been proposed for parallel GPU-based CCD computations [TMLT11]. These algorithms use bounding volume hierarchies and compute the global front of the bounding volume traversal tree (BVTT) [LC98] for parallel collision checking. However, for complex benchmarks the size of the front can become very large and updating the entire front can be expensive.

We present two techniques to accelerate the performance of collision detection. Firstly, our overall simulation algorithm (Fig. 3) decomposes these computations into two parts: (1) Proximity computations for contact and friction constraints using DCD. These constraints are used by our implicit time integrator to avoid many potential penetrations and thereby reduce the number of CCD queries; (2) Penetration computations using CCD. In practice, DCD queries are at least one order-of-magnitude cheaper than CCD queries and our decomposition scheme improves the overall performance. Moreover, we present a faster algorithm for CCD computation that uses localized propagation to update the front and exploits the spatial coherence between successive queries. The combination of these two techniques results in faster parallel collision detection algorithm for cloth simulation.

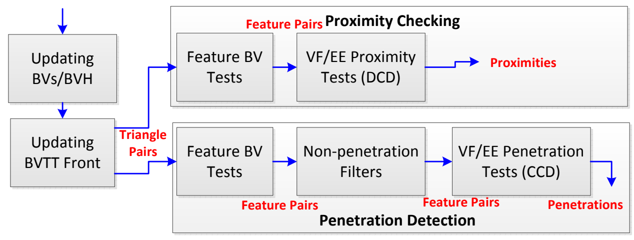

Figure 7: ***Collision Detection:*** *We use a unified framework for proximity checking and penetration detection. These modules are combined with implicit time integration and collision response.*

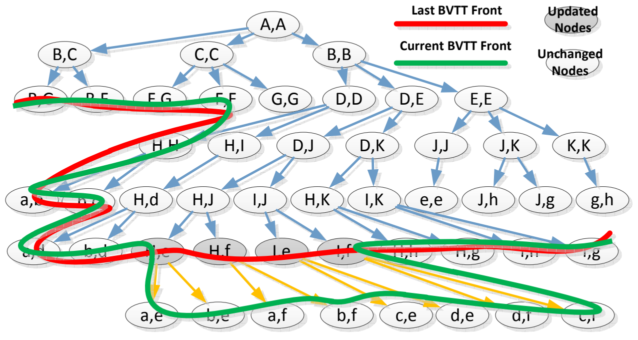

Figure 8: ***Localized BVTT Propagating:*** *Our collision detection algorithm first marks the nodes (i.e., updated nodes) whose features are associated with the updated vertices. We propagate only these marked nodes to check for new penetrations based on our localized BVTT front computation algorithm.*

Unified framework: We use the same framework for proximity checking and penetration detection in our simulation pipeline (see Fig. 3). The framework is shown in Fig. 7. We first update the bounding volume stream and bounding volume hierarchy stream by the vertex stream returned by the time integration stage. We then update the BVTT front stream and BVTT self-front (for self-collisions) stream in parallel and collect the triangle pairs that pass the bounding volume overlap test. We use vertex, edge, and face features of the triangle pairs to perform DCD for proximity computations and CCD for penetration computations (Fig. 7).

Localized Collision Detection: The collision response computation based on inelastic impact zone [HVTG08] only updates a few nodes (i.e., less than 20 in most cases, as shown in Fig. 17) to avoid penetration at each iteration. We exploit the fact that any new penetrations after that iteration are likely to appear in close geometric proximity to those nodes. As a result, we update only a subset of the front of the BVTT that corresponds to those nodes, as opposed to the entire global front. In particular, we use a localized BVTT propagating method to perform collision checking locally (see Fig. 8). By scanning the current BVTT front, the nodes whose features (vertices/faces/edges) are incident to the updated vertices are marked. We propagate only these marked nodes to check for new penetrations. The complexity of this approach is almost linear to the number of updated nodes during that iteration, as opposed to the total number of nodes in the global front. This localized BVTT update yields significantly reduced memory space and running time. In our current benchmarks, this localized propagation can provide $32\% - 54\%$ improvement to the performance of our GPU-based CCD algorithm, as compared with [TMLT11].

4.2. Collision Response

We follow the approach based on inelastic impact zones [HVTG08] and extend it to GPU parallelization. Figure 9 shows the basic pipeline of [HVTG08] and our modified approach. The key difference is that we do not group all of the impacts into isolated impact zones. Instead, we assemble all of the impacts into one linear system to perform inelastic projection and then solve the linear system to compute the new velocities of cloth nodes. This is because solving a smaller linear system is fast on CPU, while assembling and solving a large linear system is fast on GPU as it avoids launching a large number of kernels.

We use techniques similar to those described in Section 3.2 and

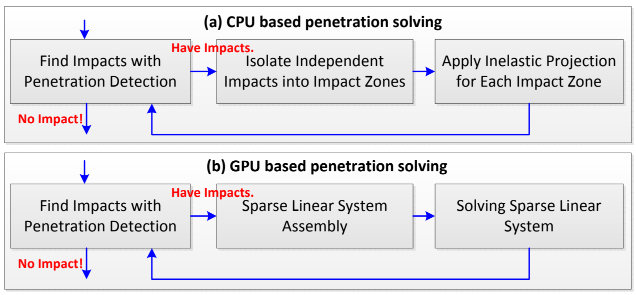
(a) CPU based penetration solving
Find Impacts with Penetration Detection
Have Impacts.
Isolate Independent Impacts into Impact Zones
Apply Inelastic Projection for Each Impact Zone
No Impact!
(b) GPU based penetration solving
Find Impacts with Penetration Detection
Have Impacts.
Sparse Linear System Assembly
Solving Sparse Linear System
No Impact!

Figure 9: ***Collision Response:*** *Difference between the original collision response algorithm [HVTG08] (a) and our GPU-based formulation (b). We do not group the impacts into isolated zones. Instead, we assemble them into a large linear system for inelastic projection and node velocity update.*

Section 3.3 to construct and solve the sparse linear system on GPU. To assemble the matrix dynamically, we execute different steps corresponding to space counting, index filling, and data filling by scanning all the impacts rather than the topological elements. We still use the Jacobi preconditioner to accelerate the CG solver. Compared with a single-core CPU implementation, our new collision response algorithm results in $55 - 80$X speedups in our benchmarks.

5. Implementation and Performance

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

Implementation. We have implemented our algorithm on three different commodity GPUs: an NVIDIA GeForce GTX 780 (with 2304 cores at 902MHz and 3G memory), an NVIDIA Tesla K20c (with 2496 cores at 706MHz and 4G memory), and an NVIDIA Tesla K40c (with 2880 cores at 745MHz and 12G memory). For these NVIDIA GPUs, we used CUDA toolkit 7.0 and Visual Stu-

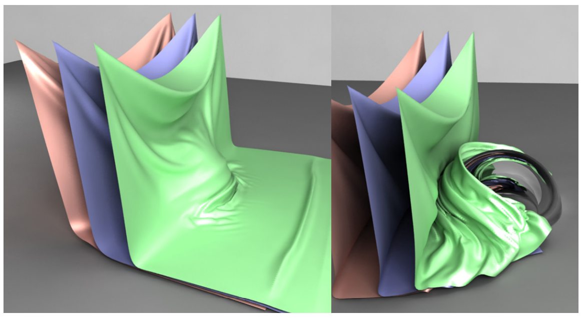

Figure 10: ***Benchmark Sphere:*** *Three pieces of hanging cloth with totally* 200*K triangles hit by a moving forward and backward sphere. Our GPU-based cloth simulation algorithm takes* 2.84*s per frame, on average.*

dio 2013 as the development environment. We use a standard PC (Windows 7 Ultimate 64 bits/Intel I7 CPU@3.5G Hz/8G RAM) as the testing environment. We use double-precision floating-point arithmetic for all the computations on GPU. We use Thrust for prefix-sum operator and cuBLAS/cuSPARSE for linear operations. No other GPU library is used in our implementation. Our GPU algorithm also uses double precision (similar to CPU methods) for all the computations, while [TTN*13] and many other approaches used single-precision arithmetic and may be prone to floating-point errors [TTWM14].

We used different kernels for each block shown in Figure 4. Our sparse matrix is stored using the BCSR format, which is regarded as the most efficient format in cuSPARSE. We could also use CSC format, but currently there is no support for block CSC in cuSPARSE. We used data coalescencing techniques, such as SOA (Structure of Arrays) instead of AOS (Array of Structures) to store the mesh topology and geometry data, including triangle node indices, edge node indices, vertex positions, etc. It can be further improved using shared memory (Section 7). We performed atomic write operators for index counting and matrix filling (Section 3.2). In practice, we did not observe much overhead due to these atomic operators, because there are not too many conflicts in the writing operations.

Benchmarks. We used three different benchmarks for regular shaped cloth simulation:

- **Sphere:** Three pieces of hanging cloth with a total of 200K triangles are hit by a forward/backward moving sphere (Fig. 10).
- **Funnel:** Three pieces of cloth with a total of 200K triangles fall into a funnel and fold to fit into the funnel (Fig. 11 (b)).
- **Twisting:** Three pieces of cloth with a total of 200K triangles twist severely as the underlying ball rotates (Fig. 11 (c)).

These benchmarks contain many inter- and intra-object collisions. We used four other benchmarks for garment simulation:

- **Andy:** A boy wearing three pieces of cloth (with 127K triangles) is practicing Kung-Fu (Fig. 1).
- **Bishop:** A swing dancer wears three pieces of cloth (with 124K triangles) (Fig. 12).
- **Falling:** A man wearing a robe (with 172K triangles) falls down rapidly under strikes (Fig. 11 (d)).

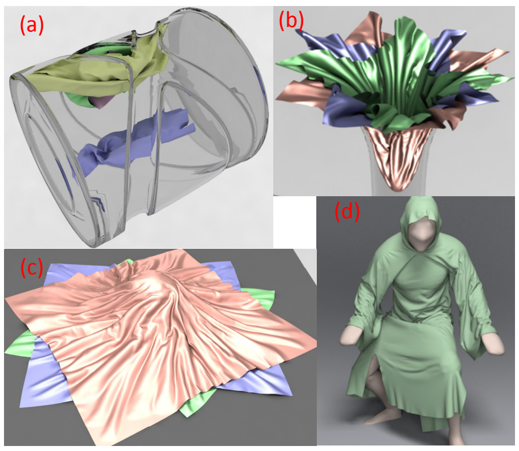

Figure 11: ***Benchmarks:*** *We use seven different benchmarks arising from regular shaped cloth simulation ((b), (c), and Fig. 10) and garment-design simulation ((a), (d), Fig. 1, and Fig. 12).*

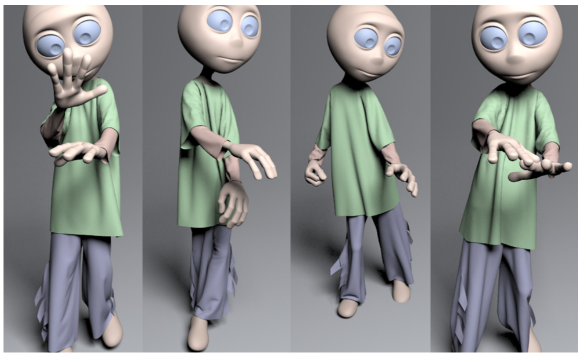

Figure 12: ***Benchmark Bishop:*** *A swing dancer wears three pieces of cloth (with* 124*K triangles). Our GPU-based cloth simulation algorithm takes* 1.19*s per frame, on average.*

- **Dryer:** Five pieces of cloth (a pair of pants, a T-shirt, a jacket, a skirt, and a robe) fall into a rotating dryer. This benchmark has $98-310$K triangles (Fig. 11 (a)).

These are complex benchmarks with multiple pieces, layers and wrinkles that result in a high number of collisions. Our algorithm can handle inter- and intra-object collisions reliably (see video).

Performance. Figure 13 shows the resolutions and time steps for different benchmarks, and highlights the performance of our algorithm on these benchmarks. This includes the average frame time of our GPU-based algorithm on three commodity GPUs with different numbers of cores. These results demonstrate that our streaming cloth simulation algorithm works well on different GPU architectures and the performance is proportional to the number of cores. We show the detailed results of the Sphere benchmark over all the frames on three different GPUs in Fig. 14. We also compare our performance with the ArcSim [NSO12] single-threaded CPU system and a simple GPU implementation using prior sparse matrix assembly and collision handling algorithms. We observe signifi-

Resolution (triangles)	Bench-marks	Time Step(s)	GTX 780	Tesla K20c	Tesla K40c	Simple GPU	CPU (s/frame)
200K	Funnel	1/300	2.22	1.82	1.68	19.03	/
200K	Moving	1/200	3.78	3.44	2.84	28.99	/
200K	Twisting	1/200	4.09	3.70	2.92	30.63	/
124K	Bishop	1/30	1.58	1.41	1.19	18.29	68.34
172K	Falling	1/30	3.74	3.07	2.49	35.68	137.31
98K-310K	Dryer	1/200	5.83	5.08	4.13	52.90	212.45
127K	Andy	1/25	3.40	2.98	2.42	28.26	143.35

Figure 13: ***Performance:*** *This figure shows the average running time for a single frame of our algorithm on the three different generations of NVIDIA GPUs. We also compare our performance with a CPU system ArcSim [NSO12] running on a single thread and a simple GPU-based system. ArcSim's collision response fails on a few benchmarks, so we can't report any results. We observe significant speedups over prior cloth simulation systems.*

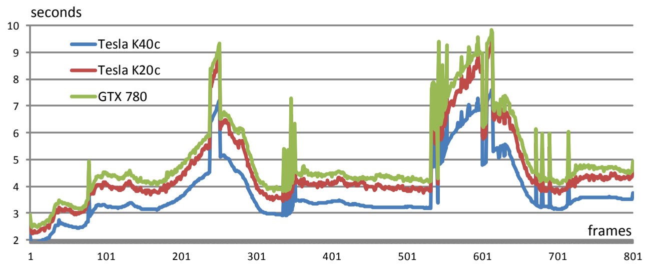

Figure 14: ***Performance on Different GPUs:*** *This figure highlights the performance of our simulator running on three different GPUs on the Sphere benchmark (200K triangles). The parallel performance almost scales linearly with the number of GPU cores.*

cant speedups over these prior systems. Note that ArcSim fails on several benchmarks due to unsuccessful penetration handling.

Figure 15 shows the simulation output for the cloth mesh represented with different resolutions: 16K, 64K, 256K, and 1M triangles. Figure 16 shows the running time per frame for the cloth benchmark represented with different resolutions. This demonstrates the change in frame rate as a function of mesh complexity. The number of wrinkles and self-collisions tend to increase at a super-linear rate for higher resolutions. The memory overhead is also proportional to the number of triangles. We also evaluated the runtime performance as a function of the *error threshold* used in our algorithms. In our current implementation, the convergence threshold in the preconditioned conjugate gradient (PCG) solver is 10^{-6}. If we decrease it to 10^{-11}, the average frame time increases by 30%. Figure 17 highlights the proximity configurations (in blue) and penetrations (in green) per frame of the Sphere benchmark. We can see only a few penetrations (< 1 per frame on average) need to be resolved during penetration handling. Figure 18 shows running time ratios for different stages of our pipeline (i.e., proximity checking(PC), sparse linear system assembly (MA), sparse linear system solving (LS), and penetration handling (PH)) and the running time per frame for the Sphere benchmark. Note that most of the time is spent in proximity query (using DCD) and penetration handling (using CCD and collision response).

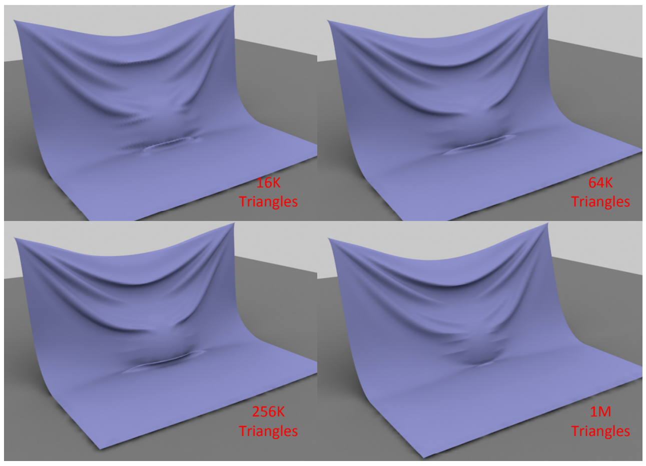

Figure 15: ***Simulation Results with Varying Resolution:*** *We highlights the results from our simulator for the cloth with different resolutions (16K, 64K, 256K, and 1M triangles, respectively).*

Resolution	16K	64K	256K	1M
Avrg. Time	0.27	1.22	5.30	35.57

Figure 16: ***Performance under Varying Resolution:*** *This figure highlights the average running time (seconds) per frame for the cloth with different resolutions.*

6. Comparisons and Analysis

We are not aware of any prior GPU-based cloth simulation algorithm or system that can handle triangle meshes of arbitrary topology and performs robust collision handling only using GPU cores. As a result, it is not possible to perform fair performance comparisons with prior work. Furthermore, other algorithms use different numeric solvers and collision handling algorithms. In this section, we compare the features and performance of our approach with prior methods.

Single-threaded CPU systems: We have compared the timings with

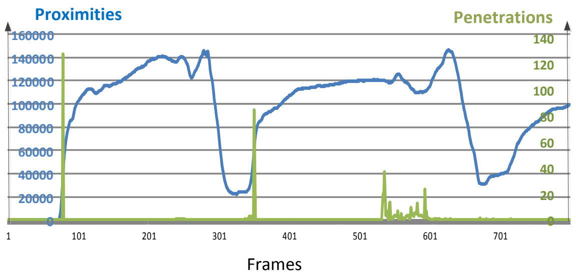

Figure 17: ***Proximity Computations/Penetrations per Frame:*** *This figure highlights proximity computations using DCD (in blue) and penetrations using CCD (in green) per frame of the Sphere benchmark.*

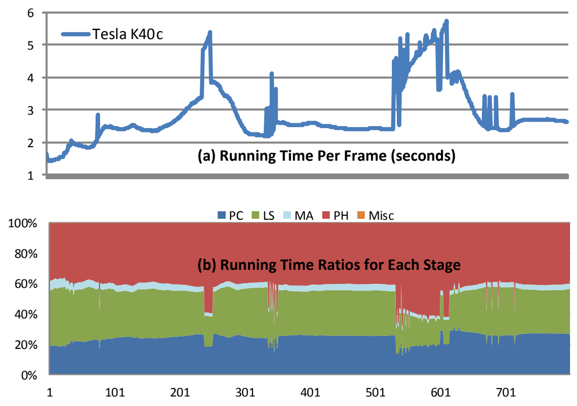

Figure 18: ***Running Time Ratios of Benchmark Sphere:*** *This figure shows the timing breakdown for different stages of the algorithm (a) and total time per frame (b) of the Sphere benchmark (with* 200*K triangles).*

a single-threaded CPU-based system, ArcSim [NSO12]. Note that the underlying pipeline in this system is slightly different from ours. Moreover, ArcSim uses a different sparse solver (TAUCS). For some benchmarks, we get very similar simulation results, as shown in the video. However, for some multi-layer benchmarks, such as Twisting, Funnel and Sphere, ArcSim fails (during a few frames) as it uses only impact zones for inter-layer collision response. In contrast, our modified pipeline incorporates inter-layer contacts into implicit time integration and is able to handle these challenging benchmarks. Moreover, we observe $47-58$X speedups in our benchmarks over this single-threaded system.

Multiple-threaded CPU systems: Selle et al. [SSIF09] accelerated cloth simulation with a 16-core workstation. For a twisting cloth with 1M triangles, they reported $5-6$ minutes per frame, which is much slower than our method. On the other hand, for a cloth benchmark with 1M triangles (Figure 15 (d)), our system takes 12s on average for each frame. ArcSim [NSO12] also can be accelerated with multiple CPU cores, but only demonstrates $30-100\%$ improvement on 8 cores. Our GPU-based implementation is about $24-44$X faster than an 8-core implementation of ArcSim on the same benchmarks. Ni et al. [NKT15] presented a scalable approach and implementation of the asynchronous contact method [HVS*09] and used that for cloth simulation. They describe the results on 384 cores of a Cray XC30 and it takes many tens of minutes on very complex benchmarks with contacts. It is hard to compare the formulation with our approach as the underlying formulation and performance are quite different from ours.

GPU-based accelerations and simulation: Some prior GPU-based methods [Kal09, Zel06] perform explicit integration and demonstrate good runtime performance, but do not perform accurate collision detection and handling. As a result, it is not clear whether these methods can reliably simulate complex benchmarks. Most of the prior GPU-based algorithms [Kal09, Zel06, TTN*13, CLMMO14] are limited to quad-mesh based cloth representations. In contrast, our approach can handle triangular meshes with arbitrary topologies. The AirMesh technique proposed by Müller et al. [MCKM15] is an alternate technique can be integrated into our approach for collision culling. As compared to the prior GPU-based cloth simulation algorithm [TTN*13], our approach offers the following benefits:

- Different GPU pipelines (Fig. 3): We incorporate proximity information into implicit time integration and perform proximity queries using DCD before time integration. In [TTN*13], DCD is used to compute repulsion forces after time integration. Our approach is more robust and can perform robust collision handling between multiple layer cloths (See Figs. 1, 10, 11, and 12). [TTN*13] would fail on these benchmarks.
- Arbitrary topology: We can represent cloth models with triangle meshes of arbitrary topology, with no restrictions. This is due to our novel matrix assembly algorithm. The algorithm in [TTN*13] can only support rectangle cloths (as shown in their benchmarks) and will not be able handle benchmarks shown in Fig. 1 and Fig. 12 (garment benchmarks).
- Robust penetration handling: [TTN*13] uses penalty and impulse based methods for penetration handling. In contrast, our method performs more robust penetration resolving by integrating proximity constraints into implicit time integration; and combines CCD with inelastic impact zone to resolve (much fewer) remaining penetrations. As a result, our GPU cloth simulation algorithm can handle complex benchmarks (see Fig. 19 & video).
- Efficient collision handling: We perform localized BVTT front propagation, which improves the performance of the integrated collision detection and response modules, as compared to [TTN*13].

Dynamic matrix assembly and unified collision handling on GPUs: In order to evaluate the benefits, we implemented various components of our algorithm, including sparse matrix assembly, linear system solving and collision handling, using prior GPU-based algorithms. The sparse matrix assembly implementation collects all the matrix entries, sorts and removes duplicated entries, and converts this matrix into the CSR format. We also use prior GPU-based collision detection algorithm in [TMLT11] without localized propagation in this system. As compared to this simple GPU system, our new algorithms for sparse matrix assembly and collision handling result in $10.2-15.4$X speedups in our benchmarks. We also tried another implementation without dynamic matrix assembly. The sparse linear system has to be assembled at CPU and transfers to GPU for solving. We observed up to $31.3-54.2$ speedups over this implementation in our benchmarks. Here the data-transfer between CPU and GPU is the main bottleneck.

Hybrid CPU-GPU systems: Some parallel algorithms are based on a hybrid use of CPU and GPU for cloth simulation. For example, Cirio et al. [CLMMO14] used GPU for implicit time integration and transferred data back to CPU for collision handling. The overhead of memory transfer can be high for complex benchmarks. Their method also limited to quad-mesh based clothes.

Prior collision handling schemes: Even with static obstacles, Tang et al. [TTN*13] requires multiple passes of impulse-based collision response (e.g., $4-5$ passes per time step for the Sphere benchmark) to overcome the potential penetrations, whereas our algorithm requires only one pass of proximity computation to generate contact/friction constraints that are used by the implicit time

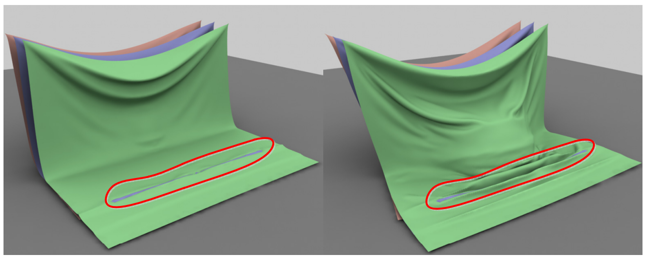

Figure 19: ***Artifacts Caused by Unresolved Penetrations:*** *The prior GPU-based algorithm [TTN*13] can result in artifacts, caused by unresolved penetrations between the multiple layers of the cloth. Our simulator can robustly handle these penetrations.*

integration. Our approach of incorporating penetration constraints into time integration is similar to the one proposed by Otaduy et al. [OTSG09], which uses an optimized LCP solver. However, our formulation is simpler to parallelize and maps well to GPU architectures. In practice, for a cloth with 120K triangles, the algorithm in [OTSG09] takes about 135 seconds for $1/200$s of simulation time step on a Intel CPU@3.2G, whereas our approach takes 3.2 seconds for the same time step on a NVIDIA Tesla K40c. Overall, our formulation of incorporating contact constraints into implicit time integration improves the robustness and efficiency.

Topological Changes: Currently, our system can not handle topological changes, e.g. adaptive remeshing, tearing, etc. In terms of all the stages of the algorithm shown in Fig. 3, the implicit time integration stage can directly support topological changes. However, the proximity checking and penetration handling stages need to be extended to handle topology changes in BVTT front computation.

Compute Bound vs. Memory Bound: For high resolution cloth simulation ($> 100K$ triangles), the performance of our algorithm is compute bounded. As a result, we can expect a higher performance with increase in the number of GPU cores. However, for low or medium resolution cloth, our system's performance is memory-bounded. A good topological/geometric data layout is important to achieve higher system performance.

7. Conclusion and Future Work

We present a GPU-based streaming cloth simulation algorithm that exploits the current GPU architectures for high parallel performance. This includes efficient parallel algorithms for sparse matrix assembly and collision handling. We use a slightly different simulation pipeline in which collision handling is closely coupled with time integration. We have demonstrated its performance on many complexly layered cloth benchmarks containing $100-300$K triangles. We observe significant speedups over prior single-threaded CPU-based systems as well as parallel GPU-based systems.

Our approach has several *limitations*. Our algorithm relies on BVTT front-based collision detection. Maintaining the BVTT front on GPU requires substantial memory space (e.g., 1.5G for the Funnel benchmark). Our time integration and collision handling algorithms do not use shared memory on GPU, since that memory is not big enough for complex benchmarks. The accuracy is governed by current GPU-based numeric libraries, such as cuSPARSE [NVI15]. Our conjugate gradient solver sometimes need a large number of iterations (from 300 to 600) to converge. Its performance may be further improved by sophisticated preconditioners. We observe good speedups due to GPU parallelization on high resolution and complexly layered cloth, and parallel CPU-based algorithms may provide equally good performance on low resolution cloth.

There are many avenues for future research. In addition to overcoming the limitations, we feel that it is possible to further improve the performance by exploiting the memory hierarchy of GPUs. The sparse matrix assembly and linear system solving algorithms could also be useful for FEM and other simulations [ACF11, WBS*13, LQT*15]. It would be useful to combine our approach with adaptive meshes [NSO12] and/or data-driven methods [FYK10,dASTH10,KGBS11,ZBO13,KKN*13] to further improve the performance and realism. Another goal would be to further parallelize the computations by using a combination of CPU and GPU, or using large clusters of CPUs and GPUs to perform interactive cloth simulation.

Acknowledgements: This research is supported in part by the National High-Tech Research and Development Program (No.2013AA013903) of China, the National Key Technology R&D Program of China (2012BAD35B01). Min Tang is supported in part by NSFC (61572423, 61170140), Zhejiang Provincial NSFC (LZ16F020003), the Doctoral Fund of Ministry of Education of China (20130101110133), and EU ANNEX project (612627). Dinesh Manocha is supported in part by ARO contract W911NF-14-1-0437 and NSF grant 1547106, and the National Thousand Talents Program of China. Huamin Wang is supported in part by NVIDIA and Adobe. Ruofeng Tong is partly supported by NSFC (61572424, 61170141). We gratefully acknowledge the support of NVIDIA Corporation for the donation of Tesla K20x/K40c GPUs used for this research. We thank FxGear for providing the models for Benchmark Andy, Bishop, and Falling. We also thank Zhendong Wang for useful discussions and making the video.

References

[ACF11] ALLARD J., COURTECUISSE H., FAURE F.: Implicit FEM solver on GPU for interactive deformation simulation. In *GPU Computing Gems, Jade Edition*, Hwu W., (Ed.). Nov. 2011, pp. 281–294. 2, 10

[BEB12] BROCHU T., EDWARDS E., BRIDSON R.: Efficient geometrically exact continuous collision detection. *ACM Trans. Graph. (SIGGRAPH) 31*, 4 (July 2012), 96:1–96:7. 2, 5

[BFA02] BRIDSON R., FEDKIW R., ANDERSON J.: Robust treatment of collisions, contact and friction for cloth animation. *ACM Trans. Graph. (SIGGRAPH) 21*, 3 (July 2002), 594–603. 2, 3, 5

[BG13] BELL N., GARLAND M.: CUSP: A C++ Templated Sparse Matrix Library, http://cusplibrary.github.io/, 2013. 3

[BW98] BARAFF D., WITKIN A.: Large steps in cloth simulation. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1998), SIGGRAPH '98, ACM, pp. 43–54. 1, 4, 5

[CK02] CHOI K.-J., KO H.-S.: Stable but responsive cloth. *ACM Trans. Graph. (SIGGRAPH) 21*, 3 (July 2002), 604–611. 1

[CLMMO14] CIRIO G., LOPEZ-MORENO J., MIRAUT D., OTADUY

M. A.: Yarn-level simulation of woven cloth. *ACM Trans. Graph. (SIGGRAPH Asia) 33*, 6 (Nov. 2014), 207:1–207:11. 2, 9

[dASTH10] DE AGUIAR E., SIGAL L., TREUILLE A., HODGINS J. K.: Stable spaces for real-time clothing. *ACM Trans. Graph. (SIGGRAPH) 29* (July 2010), 106:1–106:9. 1, 10

[FYK10] FENG W.-W., YU Y., KIM B.-U.: A deformation transformer for real-time cloth animation. *ACM Trans. Graph. (SIGGRAPH) 29*, 4 (July 2010), 108:1–108:9. 1, 10

[GKJ*05] GOVINDARAJU N. K., KNOTT D., JAIN N., KABUL I., TAMSTORF R., GAYLE R., LIN M. C., MANOCHA D.: Interactive collision detection between deformable models using chromatic decomposition. *ACM Trans. Graph. (SIGGRAPH) 24*, 3 (July 2005), 991–999. 2

[GLM05] GOVINDARAJU N. K., LIN M. C., MANOCHA D.: Quick-CULLIDE: Fast inter- and intra-object collision culling using graphics hardware. In *IEEE Virtual Reality Conference 2005, VR 2005, Bonn, Germany, March 12-16, 2005* (2005), pp. 59–66. 2

[HVS*09] HARMON D., VOUGA E., SMITH B., TAMSTORF R., GRINSPUN E.: Asynchronous Contact Mechanics. *SIGGRAPH (ACM Transactions on Graphics) 28*, 3 (Aug 2009). 9

[HVTG08] HARMON D., VOUGA E., TAMSTORF R., GRINSPUN E.: Robust treatment of simultaneous collisions. *ACM Trans. Graph. (SIGGRAPH) 27*, 3 (Aug. 2008), 23:1–23:4. 2, 6

[Kal09] KALINICH S.: 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. 9

[KGBS11] KAVAN L., GERSZEWSKI D., BARGTEIL A. W., SLOAN P.-P.: Physics-inspired upsampling for cloth simulation in games. *ACM Trans. Graph. (SIGGRAPH) 30*, 4 (Aug. 2011), 93:1–93:10. 1, 10

[KKN*13] KIM D., KOH W., NARAIN R., FATAHALIAN K., TREUILLE A., O'BRIEN J. F.: Near-exhaustive precomputation of secondary cloth effects. *ACM Trans. Graph (SIGGRAPH). 32*, 4 (July 2013), 1–8. 1, 10

[LBOK13] LIU T., BARGTEIL A. W., O'BRIEN J. F., KAVAN L.: Fast simulation of mass-spring systems. *ACM Trans. Graph. (SIGGRAPH Asia) 32*, 6 (Nov. 2013), 209:1–7. 1

[LC98] LI T.-Y., CHEN J.-S.: Incremental 3D collision detection with hierarchical data structures. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology* (New York, NY, USA, 1998), VRST '98, ACM, pp. 139–144. 5

[LMM10] LAUTERBACH C., MO Q., MANOCHA D.: gProximity: Hierarchical GPU-based operations for collision and distance queries. *Comput. Graph. Forum 29*, 2 (2010), 419–428. 2

[LQT*15] LI C., QIAN J., TONG R., CHANG J., ZHANG J.: GPU based real-time simulation of massive falling leaves. *Computational Visual Media* (2015), 1–8. 10

[LWM11] LI H., WAN Y., MA G.: A CPU-GPU hybrid computing framework for real-time clothing animation. In *Cloud Computing and Intelligence Systems (CCIS), 2011 IEEE International Conference on* (Sept 2011), pp. 391–396. 2

[LYO*10] LEE Y., YOON S.-E., OH S., KIM D., CHOI S.: Multi-resolution cloth simulation. *Comp. Graph. Forum (Pacific Graphics) 29*, 7 (2010), 2225–2232. 1

[MCKM15] MÜLLER M., CHENTANEZ N., KIM T.-Y., MACKLIN M.: Air meshes for robust collision handling. *ACM Trans. Graph. 34*, 4 (July 2015), 133:1–133:9. 9

[NKT15] NI X., KALE L., TAMSTORF R.: Scalable asynchronous contact mechanics using Charm++. In *IEEE Parallel and Distributed Processing Symposium (IPDPS)* (May 2015), pp. 677–686. 2, 9

[NSO12] NARAIN R., SAMII A., O'BRIEN J. F.: Adaptive anisotropic remeshing for cloth simulation. *ACM Trans. Graph. (SIGGRAPH Asia) 31*, 6 (Nov. 2012), 152:1–152:10. 1, 2, 7, 8, 9, 10

[NVI15] NVIDIA: cuSparse: The NVIDIA CUDA Sparse Matrix library, https://developer.nvidia.com/cusparse, 2015. 3, 10

[OTSG09] OTADUY M. A., TAMSTORF R., STEINEMANN D., GROSS M.: Implicit contact handling for deformable objects. *Computer Graphics Forum 28*, 2 (2009), 559–568. 2, 3, 10

[PKS10] PABST S., KOCH A., STRASSER W.: Fast and scalable CPU/GPU collision detection for rigid and deformable surfaces. *Comp. Graph. Forum 29*, 5 (2010), 1605–1612. 2

[Pro95] PROVOT X.: Deformation constraints in a mass-spring model to describe rigid cloth behavior. In *Proc. of Graphics Interface* (1995), pp. 147–154. 1, 2

[SKBK13] SCHMITT N., KNUTH M., BENDER J., KUIJPER A.: Multilevel cloth simulation using GPU surface sampling. In *Proceedings of VRIPHYS* (2013), pp. 1–10. 2

[SSIF09] SELLE A., SU J., IRVING G., FEDKIW R.: Robust high-resolution cloth using parallelism, history-based collisions, and accurate friction. *IEEE Trans. Vis. Comp. Graph. 15*, 2 (Mar. 2009), 339–350. 2, 5, 9

[TMLT11] TANG M., MANOCHA D., LIN J., TONG R.: Collision-Streams: Fast GPU-based collision detection for deformable models. In *Proceedings of I3D* (2011), pp. 63–70. 2, 3, 5, 6, 9

[TTN*13] TANG M., TONG R., NARAIN R., MENG C., MANOCHA D.: A GPU-based streaming algorithm for high-resolution cloth simulation. *Comp. Graph. Forum (Pacific Graphics) 32*, 7 (2013), 21–30. 2, 3, 7, 9, 10

[TTWM14] TANG M., TONG R., WANG Z., MANOCHA D.: Fast and exact continuous collision detection with Bernstein sign classification. *ACM Trans. Graph. (SIGGRAPH Asia) 33* (November 2014), 186:1–186:8. 2, 5, 7

[Wan14] WANG H.: Defending continuous collision detection against errors. *ACM Trans. Graph. (SIGGRAPH) 33*, 4 (July 2014), 122:1–122:10. 2, 5

[WBS*13] WEBER D., BENDER J., SCHNOES M., STORK A., FELLNER D.: Efficient GPU data structures and methods to solve sparse linear systems in dynamics applications. *Comp. Graph. Forum 32*, 1 (2013), 16–26. 2, 3, 10

[WHRO10] WANG H., HECHT F., RAMAMOORTHI R., O'BRIEN J.: Example-based wrinkle synthesis for clothing animation. *ACM Trans. Graph. (SIGGRAPH) 29*, 4 (July 2010), 107:1–107:8. 1

[WRO11] WANG H., RAMAMOORTHI R., O'BRIEN J. F.: Data-driven elastic models for cloth: Modeling and measurement. *ACM Trans. Graph. (SIGGRAPH) 30*, 4 (July 2011), 71:1–11. 2

[WTTM15] WANG Z., TANG M., TONG R., MANOCHA D.: TightCCD: Efficient and robust continuous collision detection using tight error bounds. *Computer Graphics Forum 34* (September 2015), 289–298. 5

[ZBO13] ZURDO J. S., BRITO J. P., OTADUY M. A.: Animating wrinkles by example on non-skinned cloth. *IEEE Trans. Vis. Comp. Graph. 19*, 1 (2013), 149–158. 1, 10

[Zel06] ZELLER C.: *Practical Cloth Simulation on Modern GPU*. Shader X4: Advanced Rendering with DirectX and OpenGL. Charles River Media, 2006. 9