P-Cloth: Interactive Complex Cloth Simulation on Multi-GPU Systems using Dynamic Matrix Assembly and Pipelined Implicit Integrators

CHENG LI, Zhejiang University
MIN TANG*, Zhejiang University
RUOFENG TONG, Zhejiang University
MING CAI†, Zhejiang University
JIEYI ZHAO, University of Texas Health Science Center at Houston
DINESH MANOCHA, University of Maryland at College Park

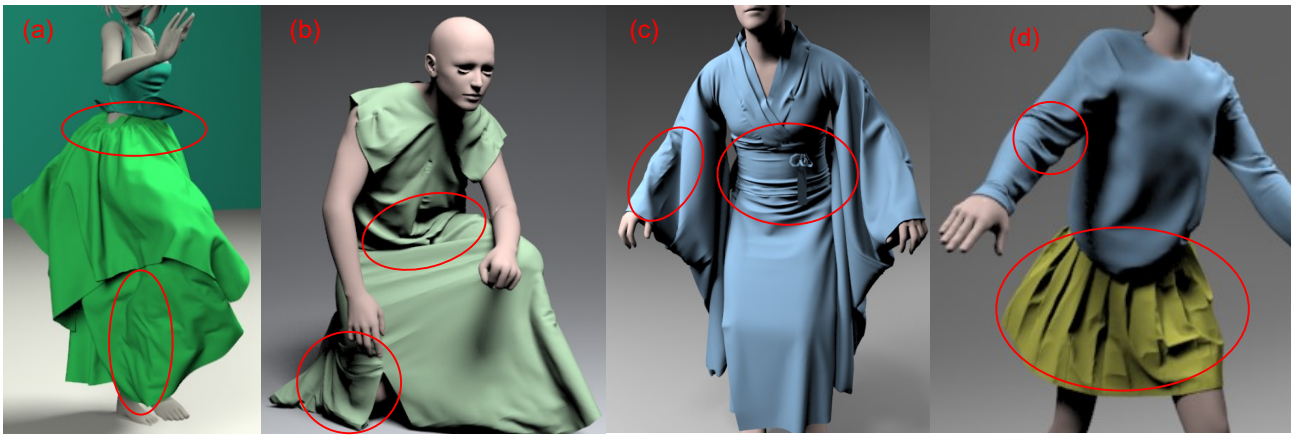

Fig. 1. **Fast Simulation on Complex Benchmarks:** Our novel multi-GPU based cloth simulation algorithm can simulate complex cloth meshes ((a) Miku with 1.33M triangles, (b) Kneel with 1.65M triangles, (c) Kimono with 1M triangles and (d) Zoey with 569K triangles) with irregular shapes and multiple layers at $2-8$ fps on workstations with multiple NVIDIA GPUs. We observe up to $8.23X$ speedups on 8 GPUs. Ours is the first approach that can perform almost interactive complex cloth simulation with wrinkles, friction and folds on commodity workstations. We highlight the areas with detailed wrinkles.

We present a novel parallel algorithm for cloth simulation that exploits multiple GPUs for fast computation and the handling of very high resolution meshes. To accelerate implicit integration, we describe new parallel algorithms for sparse matrix-vector multiplication (SpMV) and for dynamic matrix assembly on a multi-GPU workstation. Our algorithms use a novel work queue generation scheme for a fat-tree GPU interconnect topology. Furthermore, we present a novel collision handling scheme that uses spatial hashing for discrete and continuous collision detection along with a non-linear impact zone solver. Our parallel schemes can distribute the computation and storage overhead among multiple GPUs and enable us to perform almost interactive simulation on complex cloth meshes, which can hardly be handled on a single GPU due to memory limitations. We have evaluated the performance with two multi-GPU workstations (with 4 and 8 GPUs, respectively) on cloth meshes with $0.5-1.65M$ triangles. Our approach can reliably handle the collisions and generate vivid wrinkles and folds at $2-5$ fps, which is significantly faster than prior cloth simulation systems. We observe almost linear speedups with respect to the number of GPUs.

CCS Concepts: • **Computing methodologies** → **Physical simulation**; **Collision detection**;

Additional Key Words and Phrases: Cloth simulation, implicit time integration, collision handling, multi-GPU

ACM Reference Format:
Cheng Li, Min Tang, Ruofeng Tong, Ming Cai, Jieyi Zhao, and Dinesh Manocha. 2020. P-Cloth: Interactive Complex Cloth Simulation on Multi-GPU Systems using Dynamic Matrix Assembly and Pipelined Implicit Integrators. *ACM Trans. Graph.* 39, 6, Article 180 (December 2020), 15 pages. https://doi.org/10.1145/3414685.3417763

*https://min-tang.github.io/home/PCloth/
†Corresponding author.

Authors' addresses: Cheng Li, Zhejiang University, licharmy@yahoo.com; Min Tang, Zhejiang University, tang_m@zju.edu.cn; Ruofeng Tong, Zhejiang University, trf@zju.edu.cn; Ming Cai, Zhejiang University, cm@zju.edu.cn; Jieyi Zhao, University of Texas Health Science Center at Houston, jieyi.zhao@uth.tmc.edu; Dinesh Manocha, University of Maryland at College Park, dm@cs.umd.edu.

0730-0301/2020/12-ART180 $15.00
https://doi.org/10.1145/3414685.3417763

1 INTRODUCTION

Cloth simulation is an active area of research in computer graphics, computer-aided design (CAD) and the fashion industry. Over the last few decades many methods have been proposed for solving the underlying dynamical system with robust collision handling. These algorithms are implemented as parts of commercial animation and CAD systems and are widely used for different applications.

The two main components of a cloth simulation system are time integration and collision handling. Each of these steps can be time consuming and many parallel methods have been proposed to exploit multiple cores on a CPU or a GPU for higher performance. In particular, the high number of cores and memory throughput of a GPU have been used to accelerate the overall simulation pipeline [Pabst et al. 2010; Tang et al. 2016, 2018b]. Current single GPU-based cloth simulation systems can achieve $2-8$ fps on small- to medium-resolution cloth meshes with a few hundred thousand triangles. However, their performance is limited by the amount of memory available on a single GPU. Many high fidelity cloth simulation systems use cloth meshes with very high number of vertices [Eberle 2018; Kutt 2018] to generate convincing details like wrinkles, friction, folds, etc (See Fig. 1 and the video). Furthermore, fast simulation algorithms use different spatial data structures to accelerate time integration or collision detection (e.g., using bounding volume hierarchies or spatial hashing), which considerably increase the memory usage. It is non-trivial to fit such meshes and the associated data structures on commodity GPUs, which have a few gigabytes memory (e.g., 11GB memory on a NVIDIA GeForce GTX 1080 Ti GPU).

Many parallel techniques have also been proposed to utilize a large number of CPUs on a cluster [Liang and Lin 2018; Ni et al. 2015; Selle et al. 2009; Zara et al. 2004]. However, current CPUs have a lower memory bandwidth and a smaller number of cores than commodity GPUs. Moreover, workstations or computer systems with multiple GPUs are becoming widely available, e.g., NVIDIA DGX/DGX-2 workstations. As a result, it is useful to design fast parallel multi-GPU algorithms for simulating complex cloth and robustly handling the collisions. In particular, cloth simulation offers many unique challenges in terms of self-collisions or penetration handling that makes it difficult to use commonly used parallel techniques like domain decomposition. One of the major challenges is how to design methods that reduce data transfer between multiple GPUs.

In this paper, we present a novel multi-GPU based cloth simulation algorithm (P-Cloth) for high resolution meshes. Our approach parallelizes all the stages of the cloth simulation, including implicit time integration and collision handling. The novel contributions of our work include:

(1) **Pipelined SpMV:** We present a novel sparse matrix-vector multiplication (SpMV) algorithm that can handle dynamic layouts and achieves high throughput by interleaving the computations and data transfers. We also describe a new work queue generation algorithm for fat-tree interconnect topology that is used to optimize data transfer between different GPUs and improves the overall throughput (Section 3.1). We observe $1.4-2.2X$ speedups and improved scalability over prior multi-GPU SpMV algorithms.

(2) **Dynamic Matrix Assembly:** We present a new technique for matrix assembly and sparse matrix filling that accounts for dynamic contact forces and can be used with a preconditioned conjugate gradient (PCG) solver on a multi-GPU system. The matrix assembly elements are computed in a distributed manner (Section 3.2). We observe linear scalability in terms of the number of GPUs.

(3) **Parallel Collision Handling:** We present parallel algorithms for discrete and continuous collision detection using spatial hashing. Our formulation distributes the computations over multiple GPUs, such that the memory usage of a single GPU is significantly reduced. We propose a parallel non-linear impact zone solver to handle penetrations on multiple GPUs with small data-synchronization overhead (Section 4). We demonstrate that our collision detection and response computation scales linearly with the number of GPUs.

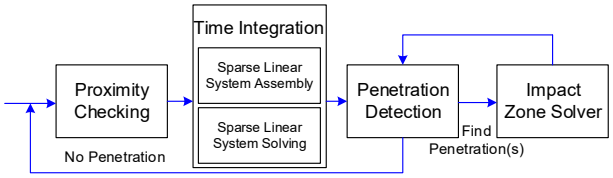

Fig. 2. **Simulation Pipeline:** The pipeline consists of time integration with contact forces, collision detection, and collision response computation. We use well-known implicit time integrator along with contact forces [Bridson et al. 2002; Otaduy et al. 2009], continuous collision detection for penetration detection [Tang et al. 2018a], and impact forces for collision resolution [Harmon et al. 2008; Tang et al. 2018b]. We present new parallel algorithms that perform these computations on multiple GPUs with reduced data transfers, including dynamic matrix assembly, pipeline implicit integrator, and parallel collision handling.

These techniques have been integrated and used to perform fast cloth simulation on complex meshes with $0.5-1.65M$ triangles on two multi-GPU workstations with 4 NVIDIA Titan Xp GPUs and 8 NVIDIA Titan V GPUs, respectively. The memory usage of P-Cloth for each GPU is about $4-8$ GB and most of the memory are used for the spatial hashing data structure and pairwise collision tests. Prior GPU-based algorithms [Tang et al. 2016, 2018b] are limited to use a single GPU and cannot handle such meshes because the memory usage can be more than 25 GB, which exceeds the memory capacity of commodity GPUs. We observe almost interactive performance ($2-5$fps) on these multi-GPU systems. In contrast, prior distributed methods for cloth simulation based on matrix-free methods can take several minutes per frame on meshes with more than $1M$ triangles on multiple Intel Xeon CPUs [Selle et al. 2009]. Moreover, we observe up to $8.23X$ speedups on 8 NVIDIA Titan V GPUs. We analyze the scalability of each algorithm and observe quasi-linear speedups on overall simulation performance. In practice, P-Cloth is the first interactive cloth simulation algorithm that can handle complex cloth meshes on commodity workstations.

2 PRIOR WORK AND BACKGROUND

2.1 Time Integrators

Many integration schemes have been proposed to increase the accuracy and performance of cloth simulation. The earlier schemes were based on explicit Euler integration [Provot 1995], which are quite fast but result in stability problems with large time steps. To overcome stability issues, numerical methods based on implicit Euler integrators [Baraff and Witkin 1998], iterative optimization [Liu et al. 2013, 2017], projective dynamics [Bouaziz et al. 2014], Anderson acceleration [Peng et al. 2018], and alternating direction method of multipliers (ADMM) [Overby et al. 2017] have been widely used. Many local/adaptive techniques have been proposed to generate the dynamic detail of the cloth and to accelerate the computation [Lee et al. 2010; Narain et al. 2012; Wang et al. 2018].

2.2 Collision Detection and Response

Robust collision handling, including collision detection and response computation, is critical for high-fidelity cloth simulation. Even a single missed or failed penetration during handling could result in noticeable artifacts [Bridson et al. 2002]. There is a considerable literature on accurate collision detection using continuous methods (CCD) and polynomial equation solvers [Brochu et al. 2012; Manocha 1998; Manocha and Demmel 1994; Provot 1997; Tang et al. 2014; Wang 2014] to ensure that not a single collision is missed between discrete steps. These algorithms are used for self-collisions and collisions between the cloth mesh and other objects in the scene. Many techniques have been proposed for collision response, including impulse computation [Bridson et al. 2002; Sifakis et al. 2008], constraint solvers [Otaduy et al. 2009], and impact zone methods [Harmon et al. 2008; Provot 1997; Tang et al. 2018b]. Other robust methods are based on elastoplastic friction models [Guo et al. 2018], asynchronous contact mechanics [Vouga et al. 2011], global intersection analysis [Baraff et al. 2003; Eberle 2018; Juntao et al. 2017], and volume simulation [Müller et al. 2015; Wong et al. 2018].

2.3 Parallel Algorithms

Given the high complexity of cloth simulation, many parallel techniques have been proposed to accelerate time integration, collision handling, or the entire cloth simulation pipeline. In particular, multiple cores on a single GPU have been used for time integration [Cirio et al. 2014; Tang et al. 2016] and collision detection [Govindaraju et al. 2005; Tang et al. 2018a]. However, these methods assume that the entire cloth mesh and the acceleration data structures fit into the memory of a single GPU and are limited to small- to medium-sized cloth meshes (e.g. up to a few hundred thousand triangles). Our approach for parallel cloth simulation is designed to exploit the computational capability and high memory bandwidth of GPUs and is complementary to prior parallel simulation algorithms.

Multi-GPU systems have been used for many scientific problems and applications, including 3D finite-difference time domain [Shao and McCollough 2017; Zhou et al. 2013], stencil computations [Sourouri et al. 2015], PDE solvers [Malahe 2016], fluid simulation [Chu et al. 2017; Hutter et al. 2014; Liu et al. 2016], and material point methods [Wang et al. 2020a,b]. Many of them are based on domain decomposition, which divides the computational region into sub-regions, then solves each sub-region independently on each GPU. However, these decomposition methods cannot be directly used for robust cloth simulation, where the linear systems for time integration are recomputed during each frame using contact forces [Otaduy et al. 2009; Tang et al. 2016]. These forces change dynamically based on the cloth/obstacle mesh configurations and result in new interface regions for the resulting sub-domains.

Sparse matrix-vector multiplication (SpMV) is an important and often time-consuming operator that arises in scientific computation. A combination of CRS and ELL formats has been used to achieve improved performance for matrices with fixed layouts [Komaritzan and Botsch 2019]. Many techniques have been proposed to perform parallel SpMV on multiple GPUs for higher throughput [Gao et al. 2017; Guo and Zhang 2016]. However, these methods may not work well in iterative solvers (e.g. conjugate gradient solvers, etc), since the results of each iteration are required to be synchronized among multiple GPUs, i.e. expensive "all-to-all" data transfers.

2.4 Interconnect between Multiple GPUs

Current multi-GPU systems or workstations support two kinds of inter-connectivity. The standard systems are based on PCI-e bus, which is a high-speed serial computer expansion bus standard based on point-to-point topology. Some newer systems support NVLink, which is a wire-based communications protocol and is used for data and control code transfer between the GPUs. In practice, NVLink specifies point-to-point connections, which provides $2-3X$ higher bandwidth than PCI-e.

Current high-end computer workstations that equipped with NVSwitches support full NVLink connection with up to 16 GPUs, allowing all GPUs to communicate with others without blocking. Other systems typically arrange multiple GPUs in a hierarchical topology, i.e. a binary fat-tree, where the physical distance between a GPU pair can have a noticeable impact on communication efficiency [Faraji et al. 2016]. Data transfer between a GPU pair with greater physical distance will need to traverse through a higher number of switches and longer paths, and thereby resulting in lower memory bandwidth. Most workstations like NVIDIA DIGITS DevBox use fat-tree topology for multi-GPU interconnect [NVIDIA 2017]. Our parallel implicit integrator takes GPU topologies and inter-connectivity into account, and strives to cut down data-transfer overhead.

2.5 Cloth Simulation Pipeline

Our basic simulation pipeline is based on implicit time integration, contact force computation, collision detection using spatial hashing, and collision response using a non-linear impact zone solver, as proposed in prior literature [Baraff and Witkin 1998; Bridson et al. 2002; Harmon et al. 2008; Otaduy et al. 2009; Tang et al. 2016, 2018b] (Fig. 2).

We use implicit integrators with contact forces due to its stability, simulation fidelity, and benefits in terms of GPU parallelization [Baraff and Witkin 1998; Narain et al. 2012; Tang et al. 2018b]. Furthermore, it can be easily integrated with GPU-based collision handling methods. We use a triangle mesh based piece-wise linear elastic model to simulate cloth with non-linear anisotropic deformations. Given a p-vertex mesh used to represent the cloth, the

overall mesh Y corresponds to a point in a high-dimensional space: $Y = R^{3p}$. We assume that the initial state of the cloth mesh is penetration-free.

We formulate the dynamical system as the following equation:

$$\mathbf{M}\ddot{\mathbf{u}} = \mathbf{f}, \tag{1}$$

where $\mathbf{M} \in R^{3p\times 3p}$ is the mass matrices of the vertices, $\mathbf{u} \in R^{3p}$ is the displacement vector of the vertices, $\mathbf{f} \in R^{3p}$ is the force vector computed using the internal and external forces. We perform time integration using the backward Euler method [Baraff and Witkin 1998], and approximate the force vector $\mathbf{f}$ at time $t + \Delta t$ using a first-order Taylor expansion:

$$\mathbf{f}(\mathbf{u}_{t+\Delta t}) = \mathbf{f}(\mathbf{u}_t) + \mathbf{J} \cdot (\mathbf{u}_{t+\Delta t} - \mathbf{u}_t), \tag{2}$$

where $\mathbf{J} \in R^{3p\times 3p}$ is the Jacobian matrix of $\mathbf{f}$ evaluated at time t. This results in the following linear equations:

$$(\mathbf{M} - \Delta t^2\mathbf{J}) \cdot \Delta\mathbf{v} = \Delta t\mathbf{f}(\mathbf{u}_t + \Delta t\mathbf{v}_t), \tag{3}$$

where $\mathbf{v} \in R^{3p}$ is the velocity vector and $\Delta\mathbf{v}$ is the increment of $\mathbf{v}$, which is the unknown variable to be solved. We solve the linear equations using a preconditioned conjugate gradient (PCG) solver. These computations are performed at each time step. The impact zone constraint-enforcement is performed, decoupled from time integration, and is similar to prior collision response algorithms [Bridson et al. 2002; Harmon et al. 2008; Tang et al. 2018b].

3 PARALLEL TIME INTEGRATION ON MULTIPLE GPUS

In this section, we present a novel parallel time integration algorithm that maps well to multiple GPUs. A key computation in the algorithm is to solve the linear system shown in Equation 3. Due to the use of contact forces, the time integration algorithm results in a new linear system during each time step. These contact forces can appear randomly due to the configuration of the cloth or environment objects, which methods based on domain decomposition can not efficiently deal with. Many conjugate gradient methods have been designed for multi-GPU systems, but are not designed to handle dynamic interface regions [Ament et al. 2010; Cevahir et al. 2009; Georgescu and Okuda 2010; Göddeke et al. 2007; Kim 2011; Müller et al. 2014; Verschoor and Jalba 2012]. Instead, we present novel multi-GPU parallel algorithms that can solve a sparse linear system with a dynamic layout. As shown in Fig. 2, during each time step we first perform sparse linear system assembly followed by sparse linear system solving. We parallelize the most computationally expensive operator, sparse matrix-vector multiplication (SpMV) using a novel method called *Pipelined SpMV*, as shown in Fig. 3(b). Furthermore, we present a new algorithm for assembling the sparse matrix dynamically during each time step on a multi-GPU system, thereby ensuring that the resulting matrix is compatible with a preconditioned conjugate gradient (PCG) solver.

3.1 Pipelined SpMV

We extend the PCG solver proposed by Tang et al. [2016] by performing vector operations and SpMV operations in parallel on multiple GPUs. Vector operations can be easily implemented on multiple GPUs: a vector can be divided into sub-vectors based on the number of GPUs and each GPU handles the corresponding part without any communication with other GPUs.

There is extensive work on fast implementations of SpMV on GPUs [Filippone et al. 2017], though most of these methods are designed for a single GPU. In the context of multi-GPU systems, one of the main issues is to reduce the communication overhead between GPUs. Previous algorithms allocate memory on the host for storing the overall vertor $\mathbf{v}$. During each iteration, the GPUs copy input vectors at the beginning for SpMV operation and send the final results back to the host memory [Cevahir et al. 2009; Georgescu and Okuda 2010; Kim 2011; Verschoor and Jalba 2012; Yamazaki et al. 2015]. In practice, waiting for such vector data from other GPUs can be time consuming and it is non-trivial to scale such methods with the number of GPUs. We present a new Pipelined SpMV algorithm that reduces the overhead on multi-GPU systems.

We use the symbol n to denote the number of GPUs in our multi-GPU system. The matrix $\mathbf{A}$ with dimension $m\times m$ is split into n sub-matrices $(\mathbf{A}_0, \ldots, \mathbf{A}_{n-1})$, each with m/n rows. Note that vector $\mathbf{v}$ is also divided into n sub-vectors $(\mathbf{v}_0, \ldots, \mathbf{v}_{n-1})$, resulting in the following SpMV computation:

$$\mathbf{Av} = \begin{bmatrix} \mathbf{A}_0 \\ \mathbf{A}_1 \\ \vdots \\ \mathbf{A}_{n-1} \end{bmatrix} \begin{bmatrix} \mathbf{v}_0 \\ \mathbf{v}_1 \\ \vdots \\ \mathbf{v}_{n-1} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_0(\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{n-1}) \\ \mathbf{A}_1(\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{n-1}) \\ \vdots \\ \mathbf{A}_{n-1}(\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{n-1}) \end{bmatrix}. \tag{4}$$

Moreover, we align $(\mathbf{v}_0, \ldots, \mathbf{v}_{n-1})$ to the same length by filling zeros at the end.

The SpMV for each sub-matrix can be assigned to a different GPU. To perform a SpMV computation for each sub-matrix $\mathbf{A}_i$ ($i \in [0, n-1]$), the entire vector $\mathbf{v}$ is required beforehand, causing "all-to-all" data transfer among the GPUs [Luehr 2016]. In other words, GPU_0 is assigned to multiply $\mathbf{A}_0$ with the entire input vector, only after it receives $\mathbf{v}_1, \ldots, \mathbf{v}_{n-1}$ from $GPU_1, GPU_2, \ldots, GPU_{n-1}$, respectively, can the SpMV start. The overall SpMV operation can be executed based on the timeline shown in Fig. 3(a).

We present a novel pipeline-based parallel algorithm to perform SpMV computation. The key idea is to perform other computations on a GPU, while they are waiting for data from other GPUs corresponding to the last iterative step. As shown in Fig. 3(b), we further split the matrix along its columns, resulting in $n \times n$ sub-matrices $\mathbf{A}_{ij}$, where $i, j \in [0, n-1]$, each with dimension $(m/n) \times (m/n)$, while the SpMV computation is performed in a similar manner. In this case, we obtain:

$$\begin{aligned} \mathbf{Av} &= \begin{bmatrix} \mathbf{A}_{00} & \mathbf{A}_{01} & \cdots & \mathbf{A}_{0,n-1} \\ \mathbf{A}_{10} & \mathbf{A}_{11} & \cdots & \mathbf{A}_{1,n-1} \\ \vdots & & \ddots & \vdots \\ \mathbf{A}_{n-1,0} & \mathbf{A}_{n-1,1} & \cdots & \mathbf{A}_{n-1,n-1} \end{bmatrix} \begin{bmatrix} \mathbf{v}_0 \\ \mathbf{v}_1 \\ \vdots \\ \mathbf{v}_{n-1} \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{A}_{00}\mathbf{v}_0 + \mathbf{A}_{01}\mathbf{v}_1 + \ldots + \mathbf{A}_{0,n-1}\mathbf{v}_{n-1} \\ \mathbf{A}_{10}\mathbf{v}_0 + \mathbf{A}_{11}\mathbf{v}_1 + \ldots + \mathbf{A}_{1,n-1}\mathbf{v}_{n-1} \\ \vdots \\ \mathbf{A}_{n-1,0}\mathbf{v}_0 + \mathbf{A}_{n-1,1}\mathbf{v}_1 + \ldots + \mathbf{A}_{n-1,n-1}\mathbf{v}_{n-1} \end{bmatrix}. \end{aligned} \tag{5}$$

As the timeline in Fig. 3(b) shows, each sub-matrix sub-vector multiplication is performed as soon as the corresponding sub-vector is received from other GPUs. In this formulation, the latency caused

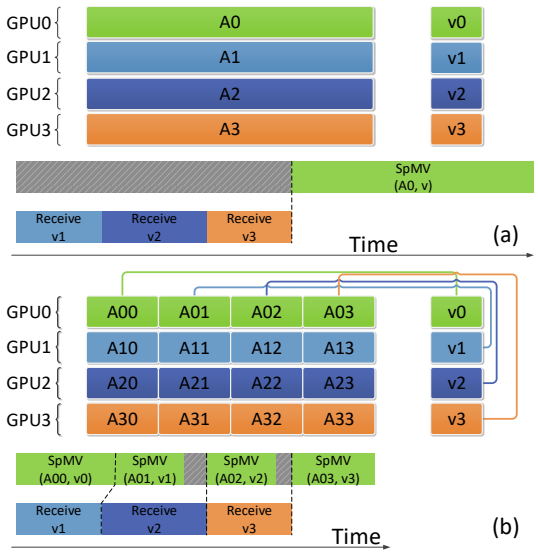

Fig. 3. **Layout of Sparse Matrix on a 4-GPU System**: We demonstrate our approach on a 4-GPU system with each GPU shown using a different color. The bottom timeline shows the computations on GPU_0. (a) A straightforward implementation of SpMV [Cevahir et al. 2009] can lead to substantial waste of computing resources (highlighted in grey color). (b) Our new Pipelined algorithm interleaves the computations and data transfers to achieve better GPU utilization.

by data communication between GPUs is mostly hidden by computation and we obtain better parallel performance for SpMV computation.

3.1.1 Overhead Analysis. We analyze the overhead that are associated with our proposed Pipelined SpMV algorithm. The performance of Pipelined SpMV is determined by the accumulative time costs of computations and data transfers. For simplicity, the following analysis is based on GPU_0 of an n-GPU system. We use the notation C_i to denote the time cost of computing $SpMV(\mathbf{A}_{0i}, \mathbf{v}_i)$, T_i for the overhead of receiving sub-vector $\mathbf{v}_i$ and S_i for accumulative time by finishing $SpMV(\mathbf{A}_{0i}, \mathbf{v}_i)$. For the straightforward implementation (Fig. 3(a)), we can easily obtain:

$$S_i = \sum_{j=1}^{i} T_j + \sum_{j=0}^{i} C_j. \tag{6}$$

And for Pipelined SpMV, operator $SpMV(\mathbf{A}_{0i}, \mathbf{v}_i)$ can be computed only after accomplishing prior computations and receiving necessary data. Therefore, the accumulative overhead is reduced to:

$$S_i = \begin{cases} \max\left(S_{i-1}, \sum_{j=1}^{i} T_j\right) + C_i, & (i > 0) \\ Ci. & (i = 0) \end{cases} \tag{7}$$

In practice, data communication overhead largely depends on the interconnectivity of the multi-GPU system, which will dominate the $\max\left(S_{i-1}, \sum_{j=1}^{i} T_j\right)$ term in Equation 7. On advanced workstation which equipped with full NVLink connection such as DGX-2 [NVIDIA 2020], data transfer among GPUs is efficient enough to allow completely hiding communication overhead, and therefore, the $\max\left(S_{i-1}, \sum_{j=1}^{i} T_j\right)$ term is equal to S_{i-1}. In this case, computation resources are fully utilized and our Pipelined SpMV can achieve ideally acceleration, which can be formed as:

$$S_i = \sum_{j=0}^{i} C_j. \quad (i > 0) \tag{8}$$

On the other hand, on systems using PCI-e bus for inter-GPU data communication, the $\max\left(S_{i-1}, \sum_{j=1}^{i} T_j\right)$ term can be approximately considered as $\sum_{j=1}^{i} T_j$, due to its limited transfer bandwidth. Then the overhead is:

$$S_i = \sum_{j=1}^{i} T_j + C_i. \quad (i > 0) \tag{9}$$

3.1.2 Efficient All-to-all Transfer on Fat-tree Topology. Recall Equation 9, although Pipelined SpMV interleaves computations and communication, in the case that data transfer via PCI-e bus, the latency caused by inter-GPU communication can still be significant and governs the overall performance. We present an efficient algorithm to reduce the transfer overhead on multi-GPU systems with fat-tree topology.

Fig. 4(a) shows an example configuration of a fat-tree topology on a 4-GPU system, where each of the sub-vectors is stored on the according GPU memory. GPUs are interconnected with switches (e.g., PCI-e internal switch, PCI-e host bridge, etc) so that they can communicate with others. GPUs within a shorter distance result in higher memory bandwidth and lower latency [Faraji et al. 2016]. For example, GPU_0 and GPU_1 can immediately communicate with each other through $Switch_1$, while communication between GPU_0 and GPU_2 is more time-consuming than immediate communication (through $Switch_1$, $Switch_0$, and $Switch_2$). We need to take these overhead into account in terms of designing data transfer and work queue generation algorithms.

During Pipelined SpMV computation, each GPU is required to collect sub-vectors from other GPUs. However, communications between different GPU pairs with overlapped traversal paths can not be performed simultaneously. For example, communication between GPU_0 and GPU_1 can be blocked while GPU_0 is communicating with GPU_2, since these two traversal paths share the part between GPU_0 and $Switch_0$. These constraints can directly impact the performance of data transfers. Moreover, the execution order of multiplications between sub-matrices and sub-vectors in the SpMV pipeline can be arbitrary. For example, GPU_0 can firstly receive $\mathbf{v}_1$ from GPU_1 and multiply $\mathbf{A}_{01}$ with $\mathbf{v}_1$. Or GPU_0 can receive $\mathbf{v}_3$ and then multiply $\mathbf{A}_{03}$ with $\mathbf{v}_3$. That means data transfers during Pipelined SpMV can also be reordered for better data traffic.

Another technique used to reduce the overhead of memory access is based on the fact that not all of data transfers via high distance paths are necessary. During Pipelined SpMV computation, each GPU collects sub-vectors for SpMV, once a GPU obtains data from a remote GPU through a high distance path, this "farther" data can be broadcast to its neighbors. In order to design an efficient scheme, we take into account all the computations that need to be performed and re-organize the execution order in the overall Pipelined SpMV.

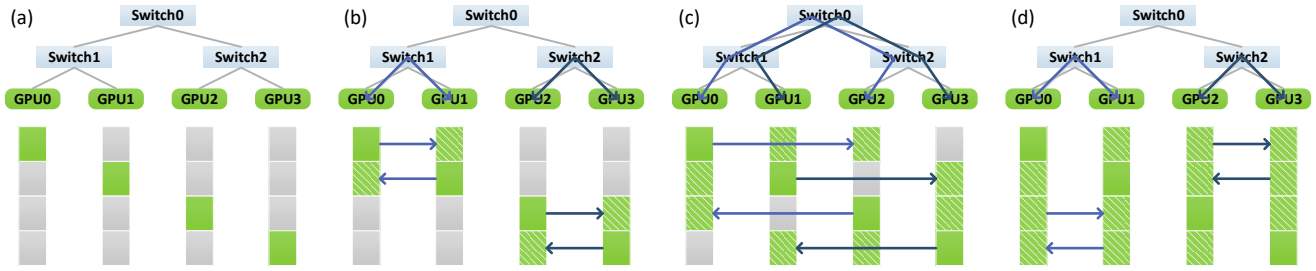

Fig. 4. **Data Transfer Scheme on Fat-tree Topology**: We highlight the performance of our novel work queue generation algorithm on a 4-GPU system with fat-tree topology. We highlight the connectivity between different GPUs using various switches and take the interconnectivity into account for data transfer. Fig.(a) shows the initial configuration before the Pipelined SpMV computation, while figures (b), (c), (d) highlight 3 transfer stages, respectively. By performing data transfers one stage at a time, followed by multiplication between the corresponding sub-matrix and sub-vector, our parallel algorithm results in higher throughput of SpMV on a multi-GPU system.

Based on these properties, we address the problem of performing $n \times (n-1)$ data transfers on an n-GPU system into n work queues, each has length $n-1$. We use the following terminology in the rest of this section.

(1) A work queue Q_i denotes the execution order of receiving required input vectors on GPU_i, each work queue contains $n-1$ queue nodes. Moreover, we extend the notation Q_i to $Q_i(s)$, where s in the brackets denotes that work queue Q_i is corresponding to a sub-system with the root switch $Switch_s$.
(2) Each queue node $N = (t, v)$ indicates a data transfer task for the corresponding GPU to GPU_t, where v represents the sub-vector.

We perform data transfers by querying all the work queues in parallel. Transfer tasks sharing the same index in the work queues are included in one **transfer stage**. During each stage, each of the n GPUs receives a sub-vector from another GPU. We synchronize all the transfer tasks at a given stage and move to the next stage.

We also use the term $Switch$ to represent the corresponding sub-system of specific GPUs in a recursive manner (see Fig. 4). For $Switch_s$ that interconnects GPUs ranging from l to r, its child switches can be considered as two sub-systems, interconnecting GPUs ranging from l to $m-1$ and from m to r respectively, where $m = (l+r)/2$.

We illustrate our transfer scheme on a 4-GPU system in Fig. 4. Fig. 4(a) highlights the initial configuration, and (b),(c),(d) correspond to 3 transfer stages, respectively. During the second stage shown in Fig. 4(b), GPU_0 obtains sub-vector $\mathbf{v}_2$ from GPU_2, which can be sent to GPU_2 during the third stage, as shown in Fig. 4(c), so that GPU_1 can receive $\mathbf{v}_2$ from GPU_1 through lower level $Switch_1$.

Our overall work queue generation algorithm uses a greedy strategy. In order to compute work queues $Q_l(s), \ldots, Q_r(s)$ of $Switch_s$, we first obtain work queues corresponding to $Switch_{2s-1}$ and $Switch_{2s}$. Next, we interchange data between these two sub-systems via $Switch_s$, ensuring that each sub-vector necessarily traverses through the root switch only once. The rest of the data transfer tasks are delegated to the sub-systems in a recursive manner. This assignment is performed by applying an offset to the vector index t of each queue node N in their work queues.

Our work queue generation algorithm works in a recursive manner using the following three steps:

(1) Compute work queues $Q_l(2s-1), \ldots, Q_{m-1}(2s-1)$ of its left child switch $Switch_{2s-1}$, then append to $Q_l(s), \ldots, Q_{m-1}(s)$. Then compute work queues $Q_m(2s), \ldots, Q_r(2s)$ of its right child switch $Switch_{2s}$ and append to $Q_m(s), \ldots, Q_r(s)$.
(2) For each queue Q_i in $Q_l(s), \ldots, Q_{m-1}(s)$, push node $(i, i+m)$ to its back. For each queue Q_j in $Q_m(s), \ldots, Q_r(s)$, push node $(j, j-m)$.
(3) Append work queues of its left child switch $Q_l(2s-1), \ldots, Q_{m-1}(2s-1)$ to $Q_l(s), \ldots, Q_{m-1}(s)$, with an offset m added to the sub-vector index of each queue node. Also, append $Q_m(2s), \ldots, Q_r(2s)$ to $Q_m(s), \ldots, Q_r(s)$ with an offset $-m$.

We assume that the work queues of the root switch are $Q_0(s), \ldots, Q_{n-1}(s)$, where n is the number of GPUs and is a power of 2. Step (2) optimizes the overall pipeline performance by minimizing the data transfers through higher level switches. After step (2), GPUs that lie in the range of the child switches obtain necessary data to deliver the overall vector to each interconnecting GPU. The rest of the transfer jobs are assigned to lower level switches for higher memory bandwidth. The pseudo-code of our work queue generation algorithm is provided in the supplementary material. We use these work queues to reduce the data transfer overhead and thereby improve the performance of SpMV based on Equation 5. As shown in Fig. 5, while the method proposed by Cevahir et al. [2009] achieves $2.2X - 2.3$ speedups a on 4-GPU system, and the performance can hardly further scale from 2-GPU to 4-GPU, our Pipelined SpMV with efficient data transfer scheme scales well among multiple GPUs and further achieves about $2.9X - 3.5X$ speedups. The input vector for SpMV changes after each CG iteration and needs to be synchronized among all the GPUs. This synchronization operation can be time-consuming (due to limited bandwidth). The performance of the simple implementation with steps (1)-(3) can be slower due to the overhead of synchronization operations (specifically in step (2)).

3.2 Matrix Assembly on Multiple GPUs

The sparse linear matrix assembly is performed based on proximity computations and contact forces, as shown in Fig. 2. Our goal

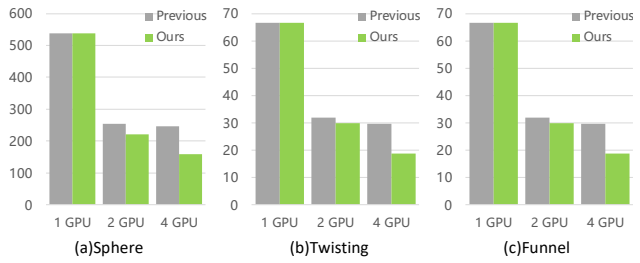

Fig. 5. **Time Cost by SpMV During a Step on Average**: We compare the time cost (ms) using our Pipelined SpMV and a straightforward method on systems with multiple GPUs. Previous approaches [Cevahir et al. 2009] (in grey) have a higher overhead due to data transfers, while our Pipelined SpMV (in green) offers better performance. As the number of GPUs increase, we observe close to linear speeds using Pipelined SpMV over a single-GPU implementation [Bell and Garland 2008].

is to design an efficient matrix assembly algorithm for multiple GPUs, that is compatible with the SpMV algorithm described above. As discussed above, we split the sparse matrix into $n \times n$ blocks, n row blocks for each of the n GPUs. Our goal is to isolate the matrix-vector multiplication computation to avoid bottlenecks during inter-GPU communication.

In order to perform matrix assembly, we mark the external and internal forces [Bridson et al. 2002; Otaduy et al. 2009] that are acting on the cloth as the assembly elements, which will form $\mathbf{f}$ and $\mathbf{J}$ in Equation 3. We first allocate memory on each GPU and then distribute all the assembly elements according to the GPU that they belong to. For each GPU, we calculate its assigned assembly elements and fill in the resulting values in the corresponding matrix blocks. We perform this computation in parallel on multiple GPUs.

3.2.1 Assembly Elements Distribution. We identify the GPUs that the vertices belong to and distribute the elements accordingly. Then each GPU fills the matrix based on its memory allocation, forming a slice of the overall $n \times n$ blocks. However, this approach requires data synchronization between the GPUs and rearranges the matrix for compatibility with the PCG solver. Furthermore, GPUs need to allocate memory buffers for the incoming data, which has additional overhead.

In order to perform these computations in an efficient manner, we preprocess all the assembly elements by grouping them according to which vertices are needed to perform the relevant computation. In other words, GPUs collect the required assembly elements beforehand according to their entries in $\mathbf{f}$ and $\mathbf{J}$ in Equation 3, then perform the relevant calculations before filling values into the n row blocks. Finally, the resulting $n \times n$ matrices $\mathbf{A}_{ij}$ $(i, j \in [0, n-1])$ on each of the n GPUs, are used as immediate input by the PCG solver.

3.2.2 Sparse Matrix Filling. The matrix assembly algorithm proposed by Tang et al. [2016] is designed for a single GPU-based simulation and stores the sparse matrix with a block compressed sparse row (BSR) format. Given the fact that SpMV is the only operator we perform on the sparse matrix, we use the ELLPACK/ITPACK (ELL) format [Grimes et al. 1979], as it offers better performance on SpMV [Bell and Garland 2008]. Moreover, we extend ELL to block ELL (BELL) to reduce the overhead of memory access, with a 3×3 block size in a structure of array (SOA) form. The matrix values corresponding to each row are stored in the *Value Table*, while the *Index Table* stores the corresponding column index of non-zero values. After computing the distributed assembly elements, each GPU can allocate memory and runs the filling algorithm for each matrix block in parallel, using the following steps:

(1) **Index Table Allocating:** We count the required memory space for each GPU during the distributing stage. We allocate memory for the *Index Table* according to that.
(2) **Index Filling:** To fill *Index Table*, we scan the distributed assembly elements and fill the column indices in the corresponding rows, using atomic operators to avoid conflicts.
(3) **Index Compacting:** Note that there can be duplicated indices, because multiple forces can be acting on one vertex. We sort each row in the *Index Table* and remove duplicated indices, one GPU thread for each row.
(4) **Value Table Allocating:** We allocate memory for the *Value Table* according to the maximum of the compacted row length.
(5) **Value Filling:** We calculate all distributed assembly elements, and fill the results in the *Value Table* using atomic operators. The value entries in the table are based on the compacted *Index Table* because their memory layouts are identical.

Since our matrix assembly algorithm is data-independent, each GPU performs these computations locally without any inter-GPU communication. Therefore the computational resources of multiple GPUs are well utilized. The pseudo-code of our sparse matrix filling algorithm is described in the supplementary material.

We are using Jacobi preconditioner with a relative residual tolerance as 0.01. The solver converges in a few iterations (< 200) in most cases. We use an upper limit of 500 in our system. As shown in Fig. 7, our matrix assembly algorithm scales well across multiple GPUs and achieves $2.2X - 2.7X$ speedups on 4-GPU system compared with CAMA [Tang et al. 2016].

4 PARALLEL COLLISION HANDLING: MULTIPLE GPUS

Collision handling is regarded as one of the major efficiency bottlenecks in cloth simulation, especially for high-resolution cloth meshes. Prior methods can take up to 82% of the total running time and have a high memory usage , e.g., 11 gigabytes [Tang et al. 2018a], to store collision handling related data structures. In this section, we present a parallel algorithm for efficient collision handling on multiple GPUs such that the memory usage on each GPU is significantly reduced.

4.1 Collision Handling Pipeline

Most prior GPU methods for collision detection are based on bounding volume hierarchies (BVHs) [Tang et al. 2016] or spatial hashing [Pabst et al. 2010]. Spatial hashing based methods [Tang et al. 2018a,b] offer improved performance in terms of GPU parallelization due to smaller memory usage and simpler computation. Our multi-GPU based collision handling algorithm extends the pipeline of Tang et al.[2018a; 2018b] by efficiently distributing the collision

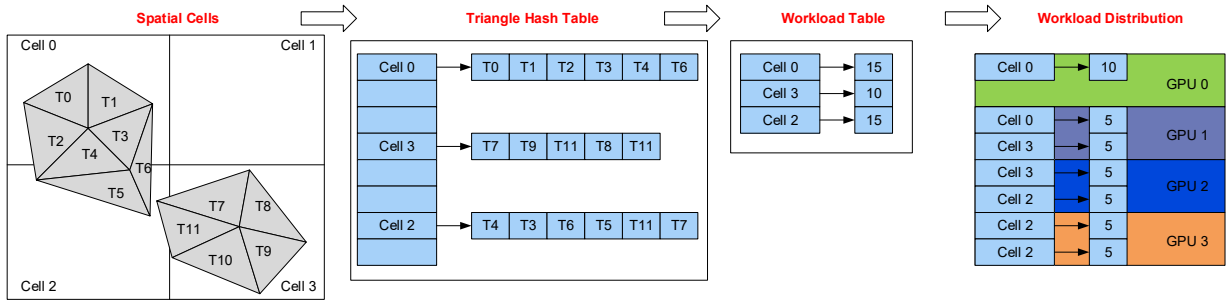

Fig. 6. **Collision Detection with Spatial Hashing:** We extend prior CCD algorithms based on spatial hashing to multi-GPU systems. We construct the triangle hash table and workload table on each GPU, and decompose the tasks based on the workload table. Each GPU performs CCD computation independently using the hash table based algorithm [Tang et al. 2018a]. Here we highlight the workload distribution on a 4-GPU system, which results lower working set size and almost linear speedups.

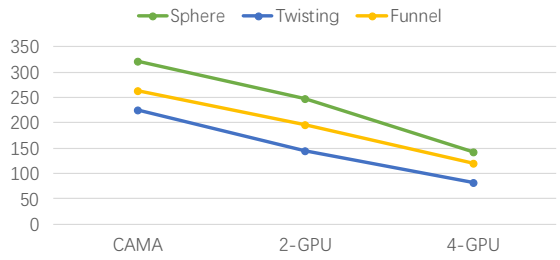

Fig. 7. **Performance comparison with CAMA**: We highlight the time cost (ms) of single-GPU implementation [Tang et al. 2016] and our parallel martrix assembly algorithm with 2 GPUs and 4 GPUs, respectively. In practice, our matrix assembly algorithm scales well across multiple GPUs.

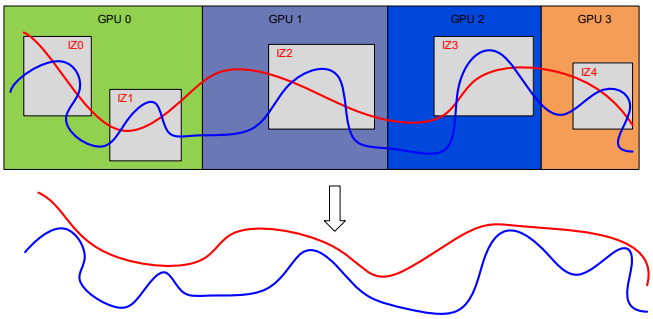

Fig. 8. **Penetration Solving with Multiple GPUs:** We highlight the approach for two colliding meshes, shown in red and blue, respectively. Multiple penetrations are computed with parallel CCD algorithm (top). All these penetrations are classified as impact zones ($IZ0, \ldots, IZ4$) and are resolved on multiple GPUs in parallel using a non-linear impact zone solver.

detection and response computations on multiple GPUs. One of our goals is to ensure load balancing between the GPUs and minimize the data synchronizing/transferring costs among all the GPUs. Our parallel scheme uses a novel workload scheme that results in more equal distribution. This reduces data synchronization and results in improved performance.

4.2 Parallel Collision Detection

For collision detection (both discrete and continuous computation), we first construct the spatial hashing table on-the-fly followed by task decomposition. We use the workload table to count computing loads, then distribute all the computing loads evenly on different GPUs. Computing related to one cell can be allocated to multiple GPUs. The algorithm is extended from prior single GPU based CD algorithm [Tang et al. 2018a,b], and differs from conventional multi-GPU based parallel CD algorithms which distribute workloads based on the cells. Our algorithm has the benefit that can distribute the workloads equally even for the configurations (e.g. Benchmark Sphere and Benchmark Funnel in the video) that many triangles converge to some spatial areas (which makes some cells have much more computing load than others). These configurations are hard to parallelize with conventional methods. After the task decomposition, each GPU performs the collision tests in parallel. The resulting pipeline is shown in Fig. 6.

4.3 Parallel Penetration Handling

A master GPU is used as an impact zone collector and used for task distribution among all the GPUs. A GPU thread block is used to solve one impact zone, as shown in Fig. 8. The impact zones are distributed among all the GPUs, and each GPU uses many threads to solve the assigned impact zones. For any colliding meshes, including self-collisions, multiple penetrations are computed using CCD. Next, these penetrations are classified as impact zones and solved on multiple GPUs in parallel. A final penetration-free status is computed in an iterative manner for each impact zone.

We incorporate proximity forces into implicit time integration process. This is different from [Bridson et al. 2002], which is a decoupled solution in which proximities and penetrations are handled separately from time integration. Our integrated approach results in much less penetrations than the decoupled solution, and provides improved stability and fidelity, similar to [Tang et al. 2018a, 2016, 2018b].

We only use SpMV for time integration. The remaining penetrations are handled using inelastic impact zones [Harmon et al. 2008], with a non-linear impact zone solver (with augmented Lagrangian method). Our impact zone solver is implemented as a

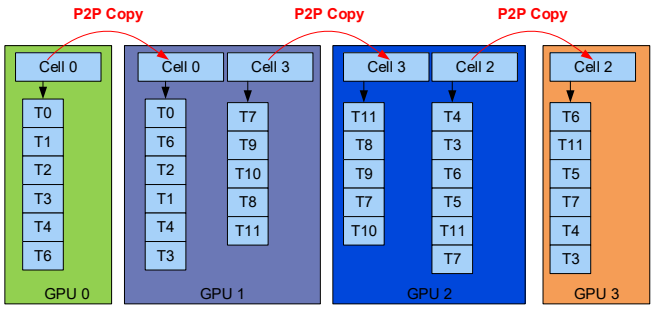

Fig. 9. **Spatial Hashing Data Synchronization among Different GPUs:** Only those cells shared by two GPUs need to be synchronized with P2P data copy on fat-tree topologies.

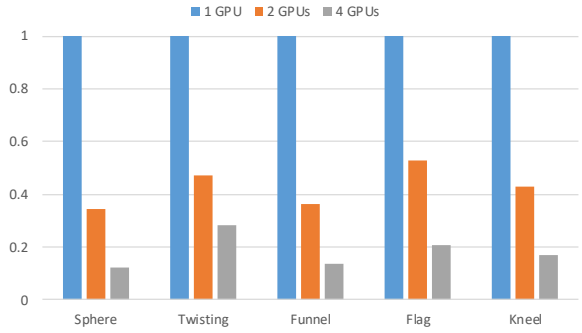

Fig. 10. **Running Time of Parallel Collision Detection with Different Number of GPUs:** The single GPU configuration is based on I-Cloth algorithm [Tang et al. 2018b] and the 2-GPU and 4-GPU configurations are correspond to our parallel collision detection algorithm. P-Cloth performance on 1-GPU is similar to that of I-Cloth. We observe almost linear speedup as the number of GPUs increase for these benchmarks: Sphere (700K triangles), Twisting (550K triangles), Funnel (550K triangles), Flag (1.2M triangles), and Kneel (1.65M triangles).

GPU-optimized gradient descending method [Tang et al. 2018b]. We mostly observe few or narrow penetrations, so the impact zone solver converges in a few iterations ($\leq$ 5). We perform a global synchronization among all GPUs after resolving all penetrations. Given few/narrow penetrations, this involves transferring a small amount data ($< 10K$) of adjusted vertices, so the data transfer cost is relatively small.

4.4 Load Balancing

As shown in Fig. 6, the on-the-fly construction of spatial hashing related data structures starts with a *Triangle Hash Table* based on the distribution of triangles among all spatial cells. This process is done on-the-fly for every frame. So it can be used for cloth meshes with dynamic topologies. Next, we perform collision tests, count how many triangle-triangle tests need to perform for each cell, and generate a *Workload Table* based on the count. Finally, we perform *load distribution* and ensure that the GPU has almost the same number of triangle-triangle test tasks. We avoid any data transfers between the GPUs by performing the construction computation on all the GPUs. As a result, each GPU maintains a copy of its own *Triangle Hash Table* and *Workload Table*. After the construction process, the collision queries are performed in parallel on all the GPUs.

Most of the collision detection computations are performed independently on different GPUs. The need for data synchronization arises for cells that are shared by two or more GPUs and have the same triangle primitives. For example, the cell $Cell_0$ is used both by GPU_0 and GPU_1, and its data is stored on both the GPUs (see Fig. 9). In total, only a maximum of $n-1$ GPUs need to be synchronized. Overall, such data synchronization has minimal impact on the overall performance since the data transfer is only need for those cells shared by GPUs.

As shown in Fig. 10, our load balancing and data synchronization methods result in an almost linear speedup with the number of GPUs. We highlight the results for several complex benchmarks: Sphere (700K triangles), Twisting (550K triangles), Funnel (550K triangles), Flag (1.2M triangles), and Kneel (1.65M triangles).

5 IMPLEMENTATION AND RESULTS

We have implemented our parallel cloth simulation algorithm (P-Cloth) and evaluated its performance on two multi-GPU workstations. Each of them has 2 Intel Xeon E5-2643 v4 CPUs with 3.40 GHz base frequency, 64 GB system memory. One of them consists of 4 NVIDIA Titan Xp GPUs (3840 CUDA cores and 12 GB memory per GPU) and the other has 8 NVIDIA Titan V GPUs (5120 CUDA cores and 12 GB memory per GPU). We run several complex benchmarks on these workstations by varying the number of GPUs to test the parallel performance of P-Cloth. Our implementation is based on CUDA toolkit 10.0/gcc/Ubuntu 16.04 LTS as the underlying development environment. We use single-precision floating-point arithmetic for all the computations on GPUs. Most GPU operations of our implementation are performed in an asynchronous manner for better resource utilization. We use *cudaMemcpyPeerAsync* for inter-GPU communications, *stream* to overlap computation and data transfers, and *cudaStreamSynchronize* for synchronization. The data transfer schedule is controlled by CPU, based on the work queues (Section 3.1).

We use various benchmarks for regular-/irregular-shaped cloth simulation:

- **Miku:** A dancing girl wearing a ruffled, layered skirt (1.33M triangles, Fig. 1(a)).
- **Kneel:** A knight is slowly kneeling down (1.66M triangles, Fig. 1(b)).
- **Kimono:** A lady bowing with beautiful kimono (1M triangles, Fig. 1(c)).
- **Zoey:** A hip hop dancer wearing a pullover and a short skirt with many pleats, (569K triangles, Fig. 1(d)).
- **Princess:** A dancer sits on the ground and generates complex folds and wrinkles on the dress (510K triangles, Fig. 16(a), right).
- **Andy:** A boy wearing three pieces of clothing (with 538K triangles) is practicing Kung-Fu (video).

Table 1. **Performance Comparison:** This table highlights the performance of P-Cloth for various benchmarks by varying the number of GPUs in the multi-GPU system. We report average FPS on various benchmarks. The speedups over the single-GPU implementation are up to $2.7X$ and $5.6X$, with 2 GPUs and 4 GPUs, respectively.

Resolution (triangles)	Benchmarks	Time Steps	1 GPU (fps)	2 GPU (fps)	4 GPU (fps)
1.33M	Miku	1/1000	0.48	1.67	2.59
538K	Andy	1/1000	0.86	2.21	4.87
1.2M	Flag	1/250	1.17	2.19	5.45
700K	Sphere	1/200	0.76	2.04	4.21
550K	Funnel	1/2000	1.74	2.87	5.20
550K	Twisting	1/2000	1.71	2.47	3.72
1M	Sphere-1M	1/200	0.23	1.31	2.11
1.65M	Kneel	1/250	0.61	1.74	2.48
510K	Princess	1/250	1.23	2.59	5.12
569K	Zoey	1/500	0.38	1.16	2.04
1M	Kimono	1/1000	0.24	0.58	1.11

- **Flag:** A flag with 1.2M triangles is waving in the blowing wind (video).
- **Sphere:** Three pieces of hanging cloth with $700K$ triangles are pushing by a forward/backward moving sphere (video).
- **Funnel:** Three pieces of cloth with $550K$ triangles is falling into a funnel and folding to fit into it (video).
- **Twisting:** Three pieces of cloth with $550K$ triangles that twist considerably, as the underlying ball rotates (video).
- **Sphere-1M:** One piece of hanging cloth with 1M triangles is pushing by a forward/backward moving sphere (video).

These are complex benchmarks with multiple pieces, layers, and wrinkles, which result in a high number of collisions. Prior methods based on a single GPU do not have sufficient memory to store these meshes and the acceleration data structures. Our P-Cloth algorithm can handle inter- and intra-object collisions reliably and efficiently on the complex benchmarks (see video). The overall accuracy of P-Cloth is the same as that of I-Cloth [Tang et al. 2018b], as we use similar geometric and numeric algorithms to compute the contact forces, implicit solvers and non-linear impact zone solver. Our simulator can support cloth with different physical parameters corresponding to various stretching/bending deformations under external forces [Wang et al. 2011]. In our benchmarks, we assign the cloth's material property as the mixture of 60% cotton and 40% polyester.

Table 1 shows the mesh resolutions and time step sizes used for different benchmarks. We also highlight the performance of P-Cloth on these benchmarks. This includes the average FPS of P-Cloth with different numbers of GPUs. We use small time steps (from $1/200$s to $1/2000$s) for high-resolution cloth simulation. Each frame corresponds to a time step. These results demonstrate that P-Cloth scales well on multiple GPUs and that the performance is almost a linear function of the number of GPUs. Compared with the performance on a single GPU, we observe significant speedups, i.e., up to $2.7X$ on 2 GPUs and up to $5.6X$ on 4 GPUs (Fig. 17). The $5.6X$ speedup is obtained on the Flag benchmark with very high mesh resolution. This super-linear speedup is due to better memory bandwidth performance and improved cache utilization. Since P-Cloth distributes the model and acceleration data structures over different GPUs, the memory usage and the working set on each GPU for P-Cloth is much lower than a single GPU-based algorithm like I-Cloth [Tang et al. 2018b]. This results in higher memory bandwidth performance and more cache hits for P-Cloth. For benchmark Sphere-1M, we obtained $5.7X$ and $9.2X$ speedups with 2 GPUs and 4 GPUs, respectively. These speedups are achieved due to the limited GPU memory size on the single GPU. The single-GPU implementation has to split the full working set into several smaller batches, and processes them separately. On the other hand, when we use 2 or 4 or 8 GPUs, each GPU has a much smaller working set, and can process them in parallel. This results in improved memory bandwidth performance. As a result, we observe super-linear speedups for these benchmarks.

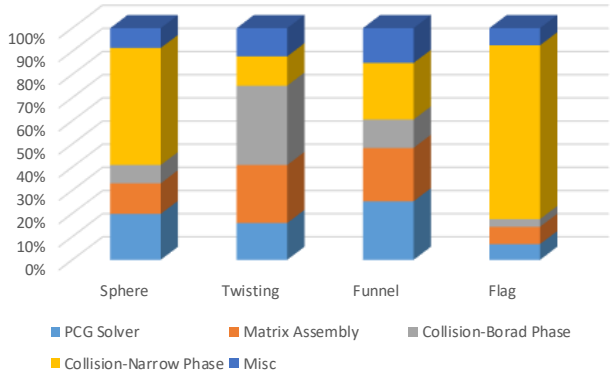

Fig. 11. **Run-time Performance Analysis:** We highlight the running time ratios of different computing stages of the cloth simulation pipeline: time integration, broad phase testing, narrow phase testing, and penetration handling. The performance data are collected by running P-Cloth on all the benchmarks on the 4-GPU workstation. As shown in the figure, time integration takes almost constant running time for all time steps. In practice, collision detection (including broad phase and narrow phase) is the major bottleneck, especially when the cloth is tangled and results in a high number of potential self-collisions.

Figure 11 shows the running time ratios of different computing stages: time integration, broad phase testing, narrow phase testing, and penetration handling. These data are collected by running P-Cloth for all the benchmarks. As shown in the figure, time integration takes almost constant running time for all the time steps. Collision detection (broad phase and narrow phase) and penetration handling appear to be the most computationally expensive parts, especially when the cloths are tangled.

We need to store the original mesh, matrices generated for implicit integration as well as the acceleration data structures used for faster collision detection. Most prior algorithms for collision detection are based on spatial hashing [Tang et al. 2018a] and BVHs [Tang et al. 2016]. In practice, these acceleration data structures can have a high memory usage, especially for techniques based on BVHs. Not only do we need to store the hierarchy, we also need to store the BVTT front and all possible triangle or feature pairs to perform the exact elementary CCD tests, including VF (vertex-face) or EE (edge-edge) continuous tests. In contrast, techniques based on spatial hashing have a lower memory usage, as we don't need to store the front.

The memory usage of P-Cloth is dominated by collision detection data structures and pairwise elementary tests. The collision detection algorithm uses considerable memory to perform the pairwise triangle/VF/EE tests in parallel. We show the average memory usage over all the frames in Fig. 12.

We also highlight the memory utilization as a function of number of GPUs. As the number of GPUs increase, we observe almost linear reduction in the memory usage. This makes it possible to perform interactive simulation on very complex meshes. For the Sphere-1M benchmark, the memory usage can be as high as 29 GB for some close proximity configurations with a very high number of potential pairs. Current GPUs may not have sufficient memory, so we need a 4-GPU system to achieve almost interactive performance. If we use 1-GPU or a 2-GPU system, the spatial data structures and collision pairs will not fit in the GPU memory, and the resulting system would not run or be too slow. We highlight the memory usage for the Sphere-1M benchmark by varying the number of GPUs in Figure 13.

Figure 14 highlights the scalability of our GPU-based parallel algorithm with respect to the number of GPUs. We collect the performance data on a workstation with 8 NVIDIA Titan V GPUs, and observe quasi-linear speedups on Benchmark Sphere and Flag (7.64X and 8.23X with 8 GPUs, respectively).

5.1 Interactive Stitching

Cloth piece stitching is an essential tools for 3D garment design [Zhang 2005]. As the result of the high performance of the P-Cloth system, our simulator can perform the stitching at interactive rate (the video). The stitching is first performed at a coarse level ($5K$ triangles), then the cloth is refined to a detailed mesh ($316K$ triangles) to improve the simulation fidelity in terms of wrinkles and folds. As shown in Fig. 15 and the video, the T-shirt model is stitching together using several cut pieces. The entire process (both the coarse and the refined levels) can be simulated at 10+ fps on the 4-GPU workstation. The pseudo-code of the stitching algorithm is provided in the supplementary material.

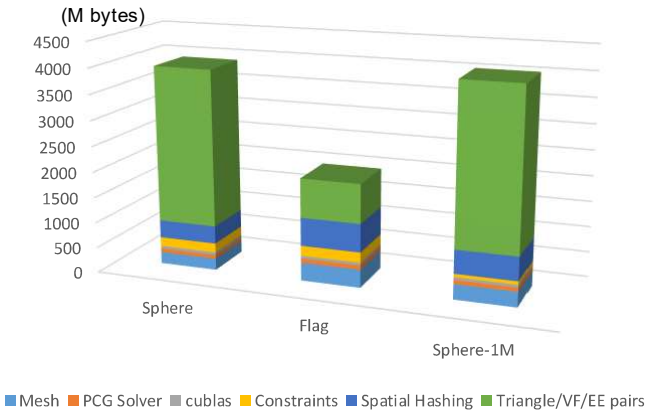

Fig. 12. **Memory Usage:** We highlight memory usage of P-Cloth on the 4-GPU workstation. This figure highlights the average memory usage on each GPU for three complex benchmarks: Sphere, Flag, and Sphere-1M. Most of the memory usage corresponds to the spatial hashing data structure and the pairwise triangle/VF/EE CCD tests. It is not possible to run this simulation on a single Titan XP GPU without splitting into sub-batches, as the collision detection algorithm needs more than 12 GB memory.

6 COMPARISON AND ANALYSIS

In this section, we compare the features and performance of our approach with prior parallel cloth simulation algorithms.

6.1 Comparison

Compared with prior GPU-based cloth simulation systems [Tang et al. 2016, 2018b], the main benefits of our algorithm include much lower memory usage due to the parallel computation (Figure 12), improved runtime performance (Table 1), and equal simulation quality (video). The reduced memory usage and smaller working set on each GPU for P-Cloth can considerably improve the runtime performance and can also result in super-linear speedups on some

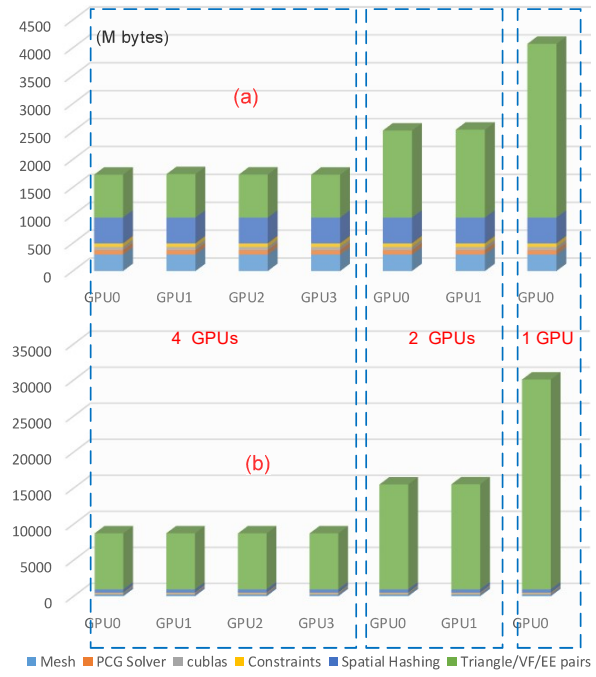

Fig. 13. **Memory Usage vs. Number of GPUs:** We highlight the memory usage (per GPU) of P-Cloth versus the number of GPUs used for simulation for Sphere-1M benchmark: (a) Highlights the average memory utilization by different data structures for a frame without many collisions; (b) Highlights the memory usage for a frame with a high number of potential self-collisions. This results in a very high number of pairwise tests after spatial hashing. For this benchmark, a single-GPU based algorithm would require about 29 GB and 2-GPU implementation of P-Cloth would need 15 GB on each GPU. On a 4-GPU system, P-Cloth takes up to 8 GB for these challenging close proximity configurations and able to perform the simulation at almost interactive rates. The reduced memory overhead on a multi-GPU system significantly increases the frame rate and performance.

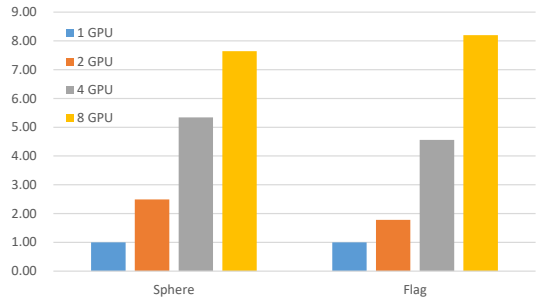

Fig. 14. **Scalability with the Number of GPUs:** This figure highlights the speedups achieved with 2, 4, and 8 GPUs, respectively, on benchmarks Flag and Sphere. The performance data are collected on the workstation with 8 NVIDIA Titan V GPUs. The quasi-linear speedups demonstrate good scalability of our GPU algorithm with the number of GPUs.

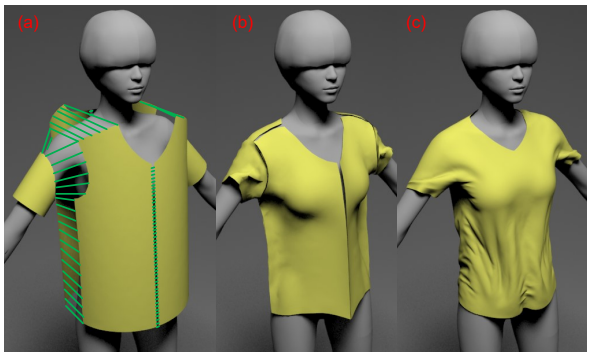

Fig. 15. **Interactive Stitching:** We use our interactive simulation algorithm to stitch cloth pieces: (left) A designer first assigns stitching pairs interactively using a simple mesh with $5K$ triangles; (middle) P-Cloth can perform the stitching operation at 10+ fps on 4 GPUs; (right) final result obtained after stitching with 316K mesh. The stitching is first performed with the coarse mesh ($5K$ triangles) and refined to the detailed mesh ($316K$ triangles).

benchmarks. With the enhanced spatial hashing data structures (see Fig. 6), P-Cloth can perform self-collision culling and perform fewer elementary tests for CCD. This suggests that the memory usage and the working set sizes can have considerable impact on the performance of collision detection and cloth simulation algorithms.

Some researchers have used matrix-free approaches for time integration to reduce the memory usage [Casafranca et al. 2020; Müller et al. 2013; Prabhune and Suresh 2020]. However, these approaches tend to be useful for large-scaled simulations (i.e., node number $> 10M$) and are not frequently used for medium-sized simulations (i.e., node number $\in [500K, 10M]$).

Compared with explicit matrix storage and assembly, matrix-free methods have the benefit of reduced memory usage. However, our matrix assembly and PCG solver takes less than 100 MB GPU memory, which is about 15% of the 8–9 GB memory used for collision handing. As shown in Fig. 12 and Fig. 13, most of the memory usage arises from collision handling due to the large number of potentially colliding triangle/VF/EE pairs. While we can use matrix-free methods to reduce the memory usage of implicit time integration, it increases the computational overhead due to the fact that force computations are non-trivial. We compare the performance of implicit time integration with a matrix-free method based implementation on a single GPU and our matrix-assembly based implementation with different number of GPUs (1 GPU/2 GPUs/4 GPUs/8 GPUs). Similar to [Casafranca et al. 2020], the matrix-free implementation (MF) evaluates the Jacobian matrix terms of the local stencils on every PCG iteration. In practice, this implementation is about $25 - 40\%$ slower than our 1 GPU implementation due to redundant memory access and computation overhead on these benchmarks (Fig. 18). This performance comparison is similar to other comparisons [Casafranca et al. 2020; Han 2015; Wang 2018].

6.2 High-resolution Cloth Meshes

We highlight the average running time of our algorithm on complex benchmarks with $510 - 1650K$ triangles (see Table 1). The actual performance, including frame rate and memory usage, also depends on the relative configuration and number of triangle pairs in close proximity. P-Cloth takes about $200 - 500$ms per frame on the 4-GPU system, which includes a high number of wrinkles, folds and self-collisions (see Fig. 16 and the video).

Comparing to the parallel cloth simulation algorithms on a CPU cluster [Liang and Lin 2018; Ni et al. 2015; Selle et al. 2009] or a hybrid combination of CPU and GPU [Pabst et al. 2010], our multi-GPU based algorithm is about 10X faster on performance for complex benchmarks. Recent cloth simulation algorithm proposed by Jiang et al. [2017] takes about 2 minutes per frame for complex benchmarks with $1.4-1.8$M triangles on an Intel Xeon system with multiple CPU cores. Not only are the underlying processors used to run these cloth simulation systems different, but also the techniques for collision detection and response computation in [Jiang et al. 2017] are different from P-Cloth. That makes it hard to make a fair comparison with prior methods. To the best of our knowledge, P-Cloth is the first cloth simulation algorithm that can perform almost interactive cloth simulation on complex meshes using commodity workstations.

Adaptive/multigrid cloth simulation algorithms [Lee et al. 2010; Narain et al. 2012; Wang et al. 2018] have been used for cloth simulations. They are complementary to our GPU-based algorithms that are based on uniform mesh tessellation. However, due to the dynamic typologies of the cloth meshes, GPU parallelization and load balancing is much harder for adaptive methods.

7 CONCLUSION AND LIMITATIONS

We present a multi-GPU based cloth simulation algorithm, P-Cloth, for high resolution meshes. It is based on three novel multi-GPU algorithms: SpMV for time integration, matrix assembly, and collision handling. Our approach is designed for sparse linear systems with a dynamic layout, which are widely used for robust cloth simulation. We have evaluated the performance on complex cloth meshes with more than a million triangles and observe almost linear speedups on workstations with 4 or 8 GPUs. P-Cloth is the first interactive cloth simulation algorithm that can handle complex cloth meshes on commodity workstations.

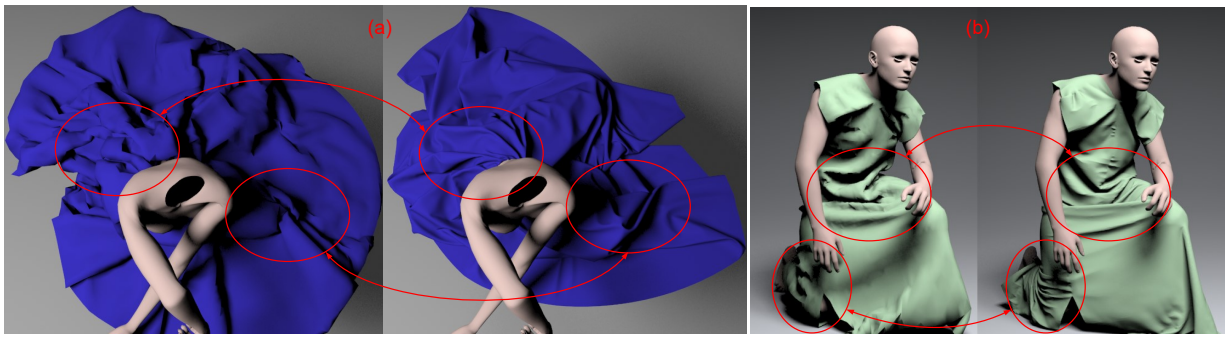

Fig. 16. **Resolution Comparison:** We use different resolutions for the two benchmarks, and highlight the detailed wrinkles and folds for the complex cloth with the higher resolution. The Princess (a) is with $10K$ and $510K$ triangles, and the Kneel (b) is with $3.1K$ and $1.65M$ triangles.

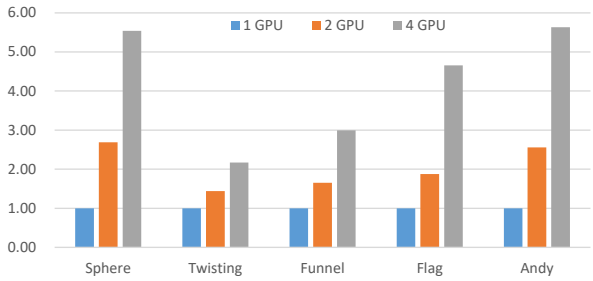

Fig. 17. **Parallelization Speedups:** We highlight the speedups using 2 or 4 GPUs for different benchmarks using our P-Cloth algorithm. We observe almost linear speedups, i.e., up to $1.4-2.6X$ on 2 GPUs and $2.1-5.6X$ on 4 GPUs. The superlinear speedups are due to better memory performance and the reduced working set size.

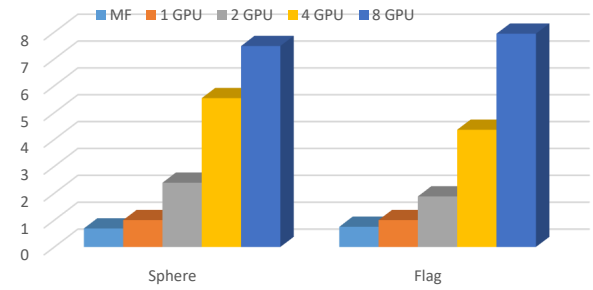

Fig. 18. **Performance Comparison with Matrix-free Method:** We compare the performance of implicit time integration using a matrix-free method based implementation on a single GPU (MF) with our matrix-assembly based implementation with different number of GPUs (1 GPU/2 GPUs/4 GPUs/8 GPUs). The matrix-free implementation is about $25-40\%$ slower than our 1 GPU-based implementation due to redundant memory accesses and computational overhead on these benchmarks.

Our approach has several limitations. For cloth configurations with folds or self-collisions, collision detection remains a bottleneck. Although our Pipelined SpMV algorithm is a general solution for all distributed systems, our work queue and data transfer algorithms are limited to fat-tree topologies for GPU interconnection and assume that the topology is known apriori. It will be useful to evaluate the performance on other topologies. The overall speedup can vary depending on the cloth configuration and data synchronization overhead. The robustness of the system is governed by contact force computation and a non-linear impact zone solver for penetration handling. Our current parallel algorithm is based on implicit time integrator, and other methods based on projective dynamics, Anderson acceleration, or ADMM may offer better stability.

Our cloth simulator can be used not only for VFX application, but also is applicable for CAD and gaming/VR applications. The parallel SpMV algorithm is also useful for other scientific applications.

There are many avenues for future research. In addition to overcoming the limitations, we need to evaluate the scalability of our approach on workstations with higher number of GPUs and different interconnect topologies. It may be possible to improve the performance by exploiting the memory hierarchy and cache layouts of modern GPUs. Our novel parallel algorithms for SpMV, implicit integration and collision detection can also be used to accelerate other simulations on multi-GPU systems. In particular, SpMV is a fundamental computation for many GPU-based scientific applications [Filippone et al. 2017] and it may be useful to apply our Pipelined SpMV algorithm (Section 3.1) and matrix assembly (Section 3.2) to other applications like finite-element simulation (FEM), material point method (MPM), etc.

ACKNOWLEDGMENTS

This work is supported in part by the National Key R&D Program of China under Grant No.: 2017YFB1002703, and the National Natural Science Foundation of China under Grant No.: 61972341, Grant No.: 61832016, Grant No.: 51775496, and Grant No.: 61732015. We would like to thank Zhiyu Zhang and Xiaorui Chen for helping on the benchmarks, Momo Inc. for the Benchmark Kimono, Zhijiang Lab for the 8-GPU workstation, and the anonymous referees for their valuable comments and helpful suggestions.

REFERENCES

Marco Ament, Gunter Knittel, Daniel Weiskopf, and Wolfgang Strasser. 2010. A Parallel Preconditioned Conjugate Gradient Solver for the Poisson Problem on a Multi-GPU Platform. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. 583–592.

David Baraff and Andrew Witkin. 1998. Large Steps in Cloth Simulation. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques (SIGGRAPH '98)*. ACM, New York, NY, USA, 43–54.
David Baraff, Andrew Witkin, and Michael Kass. 2003. Untangling Cloth. In *SIGGRAPH '03*. ACM, New York, NY, USA, 862–870.
Nathan Bell and Michael Garland. 2008. *Efficient Sparse Matrix-vector Multiplication on CUDA*. Technical Report. Technical Report NVR-2008-004, Nvidia Corporation.
Sofien Bouaziz, Sebastian Martin, Tiantian Liu, Ladislav Kavan, and Mark Pauly. 2014. Projective Dynamics: Fusing Constraint Projections for Fast Simulation. *ACM Trans. Graph.* 33, 4, Article 154 (July 2014), 11 pages.
Robert Bridson, Ronald Fedkiw, and John Anderson. 2002. Robust Treatment of Collisions, Contact and Friction for Cloth Animation. *ACM Trans. Graph.* 21, 3 (2002), 594–603.
Tyson Brochu, Essex Edwards, and Robert Bridson. 2012. Efficient Geometrically Exact Continuous Collision Detection. *ACM Trans. Graph.* 31, 4 (2012), 96:1–96:7.
Juan J. Casafranca, Gabriel Cirio, Alejandro Rodríguez, Eder Miguel, and Miguel A. Otaduy. 2020. Mixing Yarns and Triangles in Cloth Simulation. *Computer Graphics Forum* 39, 2 (2020), 101–110.
Ali Cevahir, Akira Nukada, and Satoshi Matsuoka. 2009. Fast Conjugate Gradients with Multiple GPUs. In *Inter. Conf. on Computational Science*. Springer, 893–903.
Jieyu Chu, Nafees Bin Zafar, and Xubo Yang. 2017. A Schur Complement Preconditioner for Scalable Parallel Fluid Simulation. *ACM Trans. Graph.* 36, 5 (2017), 163.
Gabriel Cirio, Jorge Lopez-Moreno, David Miraut, and Miguel A. Otaduy. 2014. Yarn-level Simulation of Woven Cloth. *ACM Trans. Graph. (SIGGRAPH Asia)* 33, 6, Article 207 (Nov. 2014), 11 pages.
David Eberle. 2018. Better Collisions and Faster Cloth for Pixar's Coco. In *ACM SIGGRAPH 2018 Talks (SIGGRAPH '18)*. Article 8, 2 pages.
Iman Faraji, Seyed H Mirsadeghi, and Ahmad Afsahi. 2016. Topology-aware GPU Selection on Multi-GPU Nodes. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 712–720.
Salvatore Filippone, Valeria Cardellini, Davide Barbieri, and Alessandro Fanfarillo. 2017. Sparse Matrix-vector Multiplication on GPGPUs. *ACM Transactions on Mathematical Software (TOMS)* 43, 4 (2017), 30.
Jiaquan Gao, Yu Wang, and Jun Wang. 2017. A Novel Multi–graphics Processing Unit Parallel Optimization Framework for the Sparse Matrix-vector Multiplication. *Concurrency and Computation: Practice and Experience* 29, 5 (2017), e3936.
Serban Georgescu and Hiroshi Okuda. 2010. Conjugate Gradients on Multiple GPUs. *International Journal for Numerical Methods in Fluids* 64, 10-12 (2010), 1254–1273.
Dominik Göddeke, Robert Strzodka, Jamaludin Mohd-Yusof, Patrick McCormick, Sven HM Buijssen, Matthias Grajewski, and Stefan Turek. 2007. Exploring Weak Scalability for FEM Calculations on a GPU-enhanced Cluster. *Parallel Comput.* 33, 10-11 (2007), 685–699.
Naga K. Govindaraju, David Knott, Nitin Jain, Ilknur Kabul, Rasmus Tamstorf, Russell Gayle, Ming C. Lin, and Dinesh Manocha. 2005. Interactive Collision Detection between Deformable Models using Chromatic Decomposition. *ACM Trans. Graph. (SIGGRAPH)* 24, 3 (July 2005), 991–999.
Roger G. Grimes, David R. Kincaid, and David M. Young. 1979. ITPACK 2.0 User's Guide. Report No. CNA-150. *Center for Numerical Analysis, University of Texas at Austin* (1979).
Ping Guo and Changjiang Zhang. 2016. Performance Optimization for SpMV on Multi-GPU Systems Using Threads and Multiple Streams. In *2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*. IEEE, 67–72.
Qi Guo, Xuchen Han, Chuyuan Fu, Theodore Gast, Rasmus Tamstorf, and Joseph Teran. 2018. A Material Point Method for Thin Shells with Frictional Contact. *ACM Trans. Graph.* 37, 4 (2018), 147:1–147:15.
Dongsoo Han. 2015. Interleaved Cloth Simulation. In *Workshop on Virtual Reality Interaction and Physical Simulation*, Fabrice Jaillet, Florence Zara, and Gabriel Zachmann (Eds.). The Eurographics Association. https://doi.org/10.2312/vriphys.20151330
David Harmon, Etienne Vouga, Rasmus Tamstorf, and Eitan Grinspun. 2008. Robust Treatment of Simultaneous Collisions. *ACM Trans. Graph.* 27, 3 (2008), 23:1–23:4.
Marco Hutter, Martin Knuth, and Arjan Kuijper. 2014. Mesh partitioning for parallel garment simulation. In *Proceedings of WSCG*. 125–133.
Chenfanfu Jiang, Theodore Gast, and Joseph Teran. 2017. Anisotropic Elastoplasticity for Cloth, Knit and Hair Frictional Contact. *ACM Trans. Graph.* 36, 4 (2017), 152:1–14.
Ye Juntao, Ma Guanghui, Jiang Liguo, Chen Lan, Li Jituo, Xiong Gang, Zhang Xiaopeng, and Tang Min. 2017. A Unified Cloth Untangling Framework Through Discrete Collision Detection. *Computer Graphics Forum* 36, 7 (2017), 217–228.
Hyung-Jin Kim. 2011. GPU Performance of Conjugate Gradient Solver with Staggered Fermions. In *The XXVIII International Symposium on Lattice Field Theory*. SISSA Medialab, 028.
Martin Komaritzan and Mario Botsch. 2019. Fast Projective Skinning. In *Motion, Interaction and Games*. Association for Computing Machinery, New York, NY, USA, Article 22, 10 pages.
Aimei Kutt. 2018. Art-directed Costumes at Pixar: Design, Tailoring, and Simulation in Production. In *SIGGRAPH Asia 2018 Courses (SA '18)*. Article 2, 102 pages.
Yongjoon Lee, Sung-Eui Yoon, Seungwoo Oh, Duksu Kim, and Sunghee Choi. 2010. Multi-Resolution Cloth Simulation. *Comp. Graph. Forum* 29, 7 (2010), 2225–2232.
Junbang Liang and Ming C. Lin. 2018. Time-Domain Parallelization for Accelerating Cloth Simulation. *Computer Graphics Forum* 37, 8 (2018), 21–34.
Haixiang Liu, Nathan Mitchell, Mridul Aanjaneya, and Eftychios Sifakis. 2016. A Scalable Schur-complement Fluids Solver for Heterogeneous Compute Platforms. *ACM Trans. Graph.* 35, 6, Article 201 (Nov. 2016), 12 pages.
Tiantian Liu, Adam W. Bargteil, James F. O'Brien, and Ladislav Kavan. 2013. Fast Simulation of Mass-Spring Systems. *ACM Trans. Graph.* 32, 6 (2013), 209:1–7.
Tiantian Liu, Sofien Bouaziz, and Ladislav Kavan. 2017. Quasi-Newton Methods for Real-Time Simulation of Hyperelastic Materials. *ACM Trans. Graph.* 36, 3, Article 23 (May 2017), 16 pages.
Nathan Luehr. 2016. Fast Multi-GPU Collectives with NCCL. https://devblogs.nvidia.com/fast-multi-gpu-collectives-nccl/
Michael Malahe. 2016. *PDE Solvers for Hybrid CPU-GPU Architectures*. Ph.D. Dissertation. The University of North Carolina at Chapel Hill.
Dinesh Manocha. 1998. Solving Polynomial Equations. *Applications of Computational Algebraic Geometry: American Mathematical Society Short Course, January 6-7, 1997, San Diego, California* 53 (1998), 41.
Dinesh Manocha and James Demmel. 1994. Algorithms for Intersecting Parametric and Algebraic Curves I: Simple Intersections. *ACM Trans. Graph.* 13, 1 (Jan. 1994), 73–100.
Eike Müller, Xu Guo, Robert Scheichl, and Sinan Shi. 2013. Matrix-Free GPU Implementation of a Preconditioned Conjugate Gradient Solver for Anisotropic Elliptic PDEs. *Comput. Vis. Sci.* 16, 2 (April 2013), 41–58. https://doi.org/10.1007/s00791-014-0223-x
Eike Hermann Müller, Robert Scheichl, and Eero Vainikko. 2014. Petascale Elliptic Solvers for Anisotropic PDEs on GPU Clusters. *arXiv preprint arXiv:1402.3545* (2014).
Matthias Müller, Nuttapong Chentanez, Tae-Yong Kim, and Miles Macklin. 2015. Air Meshes for Robust Collision Handling. *ACM Trans. Graph.* 34, 4, Article 133 (July 2015), 9 pages.
Rahul Narain, Armin Samii, and James F. O'Brien. 2012. Adaptive Anisotropic Remeshing for Cloth Simulation. *ACM Trans. Graph.* 31, 6, Article 152 (Nov. 2012), 10 pages.
Xiang Ni, L.V. Kale, and R. Tamstorf. 2015. Scalable Asynchronous Contact Mechanics Using Charm++. In *IEEE Parallel and Distributed Processing Symposium (IPDPS)*. 677–686.
NVIDIA. 2017. DIGITS DEVBOX User Guide. https://docs.nvidia.com/dgx/pdf/DIGITS_DEVBOX_User_Guide.pdf
NVIDIA. 2020. NVIDIA DGX-2 System User Guide. https://docs.nvidia.com/dgx/pdf/dgx2-user-guide.pdf
Miguel A. Otaduy, Rasmus Tamstorf, Denis Steinemann, and Markus Gross. 2009. Implicit Contact Handling for Deformable Objects. *Computer Graphics Forum* 28, 2 (2009), 559–568.
Matthew Overby, George E. Brown, Jie Li, and Rahul Narain. 2017. ADMM ⊇ Projective Dynamics: Fast Simulation of Hyperelastic Models with Dynamic Constraints. *IEEE Transactions on Visualization and Computer Graphics* 23, 10 (2017), 2222–2234.
Simon Pabst, Artur Koch, and Wolfgang Straßer. 2010. Fast and Scalable CPU/GPU Collision Detection for Rigid and Deformable Surfaces. *Comp. Graph. Forum* 29, 5 (2010), 1605–1612.
Yue Peng, Bailin Deng, Juyong Zhang, Fanyu Geng, Wenjie Qin, and Ligang Liu. 2018. Anderson Acceleration for Geometry Optimization and Physics Simulation. *ACM Trans. Graph.* 37, 4, Article 42 (July 2018), 14 pages.
Bhagyashree C. Prabhune and Krishnan Suresh. 2020. A Fast Matrix-free Elasto-plastic Solver for Predicting Residual Stresses in Additive Manufacturing. *Comput. Aided Des.* 123 (2020), 102829.
Xavier Provot. 1995. Deformation Constraints in a Mass-spring Model to Describe Rigid Cloth Behavior. In *Proc. of Graphics Interface*. 147–154.
Xavier Provot. 1997. Collision and Self-collision Handling in Cloth Model Dedicated to Design Garments. In *Graphics Interface*. 177–189.
Andrew Selle, Jonathan Su, Geoffrey Irving, and Ronald Fedkiw. 2009. Robust High-Resolution Cloth Using Parallelism, History-Based Collisions, and Accurate Friction. *IEEE Trans. Vis. Comp. Graph.* 15, 2 (March 2009), 339–350.
Wenyi Shao and William McCollough. 2017. Multiple-GPU-based Frequency-dependent Finite-difference Time Domain Formulation using MATLAB Parallel Computing Toolbox. *Progress In Electromagnetics Research* 60 (2017), 93–100.
Eftychios Sifakis, Sebastian Marino, and Joseph Teran. 2008. Globally Coupled Collision Handling using Volume Preserving Impulses. In *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. 147–153.
Mohammed Sourouri, Johannes Langguth, Filippo Spiga, Scott B Baden, and Xing Cai. 2015. CPU+ GPU programming of Stencil Computations for Resource-Efficient use of GPU Clusters. In *2015 IEEE 18th International Conference on Computational Science and Engineering*. IEEE, 17–26.

Min Tang, Zhongyuan Liu, Ruofeng Tong, and Dinesh Manocha. 2018a. PSCC: Parallel Self-Collision Culling with Spatial Hashing on GPUs. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 1, 1 (2018), 18:1–18.

Min Tang, Ruofeng Tong, Zhendong Wang, and Dinesh Manocha. 2014. Fast and Exact Continuous Collision Detection with Bernstein Sign Classification. *ACM Trans. Graph. (SIGGRAPH Asia)* 33 (November 2014), 186:1–186:8. Issue 6.

Min Tang, Huamin Wang, Le Tang, Ruofeng Tong, and Dinesh Manocha. 2016. CAMA: Contact-Aware Matrix Assembly with Unified Collision Handling for GPU-based Cloth Simulation. *Computer Graphics Forum* 35, 2 (2016), 511–521.

Min Tang, Tongtong Wang, Zhongyuan Liu, Ruofeng Tong, and Dinesh Manocha. 2018b. I-Cloth: Incremental Collision Handling for GPU-Based Interactive Cloth Simulation. *ACM Trans. Graph.* 37, 6 (November 2018), 204:1–10.

Mickeal Verschoor and Andrei C Jalba. 2012. Analysis and Performance Estimation of the Conjugate Gradient Method on Multiple GPUs. *Parallel Comput.* 38, 10-11 (2012), 552–575.

Etienne Vouga, David Harmon, Rasmus Tamstorf, and Eitan Grinspun. 2011. Asynchronous Variational Contact Mechanics. *Computer Methods in Applied Mechanics and Engineering* 200 (June 2011), 2181–2194.

Huamin Wang. 2014. Defending Continuous Collision Detection Against Errors. *ACM Trans. Graph. (SIGGRAPH)* 33, 4, Article 122 (July 2014), 10 pages.

Huamin Wang. 2018. Rule-Free Sewing Pattern Adjustment with Precision and Efficiency. *ACM Trans. Graph.* 37, 4, Article 53 (July 2018), 13 pages.

Huamin Wang, James F. O'Brien, and Ravi Ramamoorthi. 2011. Data-Driven Elastic Models for Cloth: Modeling and Measurement. *ACM Trans. Graph.* 30, 4, Article Article 71 (July 2011), 12 pages.

Xinlei Wang, Minchen Li, Yu Fang, Xinxin Zhang, Ming Gao, Min Tang, Danny M. Kaufman, and Chenfanfu Jiang. 2020a. Hierarchical Optimization Time Integration for CFL-Rate MPM Stepping. *ACM Trans. Graph.* 39, 3, Article 21 (April 2020), 16 pages.

Xinlei Wang, Yuxing Qiu, Stuart R. Slattery, Yu Fang, Minchen Li, Song-Chun Zhu, Yixin Zhu, Min Tang, Dinesh Manocha, and Chenfanfu Jiang. 2020b. A Massively Parallel and Scalable Multi-GPU Material Point Method. *ACM Trans. Graph. (SIGGRAPH* 39, 5, Article 1 (Aug. 2020), 12 pages.

Zhendong Wang, Longhua Wu, Marco Fratarcangeli, Min Tang, and Huamin Wang. 2018. Parallel Multigrid for Nonlinear Cloth Simulation. *Computer Graphics Forum* 37, 7 (2018), 131–141.

Audrey Wong, David Eberle, and Theodore Kim. 2018. Clean Cloth Inputs: Removing Character Self-intersections with Volume Simulation. In *ACM SIGGRAPH 2018 Talks (SIGGRAPH '18)*. Article 42, 2 pages.

Ichitaro Yamazaki, Stanimire Tomov, and Jack Dongarra. 2015. Mixed-precision Cholesky QR factorization and its case studies on multicore CPU with multiple GPUs. *SIAM Journal on Scientific Computing* 37, 3 (2015), C307–C330.

Florence Zara, François Faure, and Vincent Jean-Marc. 2004. Parallel Simulation of Large Dynamic System on a PCs Cluster: Application to Cloth Simulation. *International Journal of Computers and Applications* 26, 3 (March 2004).

Dongliang Zhang. 2005. Cloth Design and Application. In *ACM SIGGRAPH 2005 Courses*. ACM, New York, NY, USA, 5–37.

Jun Zhou, Yifeng Cui, Efecan Poyraz, Dong Ju Choi, and Clark C Guest. 2013. Multi-GPU Implementation of a 3D Finite Difference Time Domain Earthquake Code on Heterogeneous Supercomputers. *Proc. of Computer Science* 18 (2013), 1255–1264.