CUDA Deformers for Model Reduction

BOHAN WANG, University of Southern California, USA
JERNEJ BARBIČ, University of Southern California, USA

Real-time deformable object simulation is important in interactive applications such as games and virtual reality. One common approach to achieve speed is to employ model reduction, a technique whereby the equations of motion of a deformable object are projected to a suitable low-dimensional space. Improving the real-time performance of model-reduced systems has been the subject of much research. While modern GPUs play an important role in real-time simulation and parallel computing, existing model reduction systems typically utilize CPUs and seldom employ GPUs. We give a method to efficiently employ GPUs for vertex position computation in model-reduced simulations. Our CUDA-based algorithm gives a substantial speedup compared to a CPU implementation, thanks to our system architecture that employs a memory layout friendly to GPU memory, reduces the communication between the CPU and GPU, and enables the CPU and GPU to work in parallel.

CCS Concepts: • **Computing methodologies** → **Physical simulation**.

Additional Key Words and Phrases: model reduction, real-time, CUDA, GPU, deformation, botanical

ACM Reference Format:
Bohan Wang and Jernej Barbič. 2020. CUDA Deformers for Model Reduction. In *Motion, Interaction and Games (MIG '20), October 16–18, 2020, Virtual Event, SC, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3424636.3426895

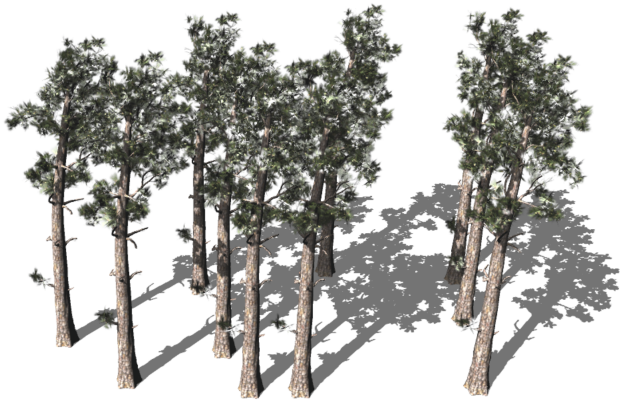


Fig. 1. Simulation of this plant forest runs at 50 FPS, thanks to our CUDA modal displacement deformer. Each branch and leaf is a separate model-reduced object. Calculating the reduced coordinates $q^{(i)}$ for all the objects (6,440) takes 20 msec per timestep. Observe that we can compute vertex positions and transformations on the GPU in parallel with timestepping $q^{(i)}$; hence we can achieve the 50 FPS frame rate = 1 / (20 msec). For comparison, the CPU system runs at 12 FPS.

1 INTRODUCTION

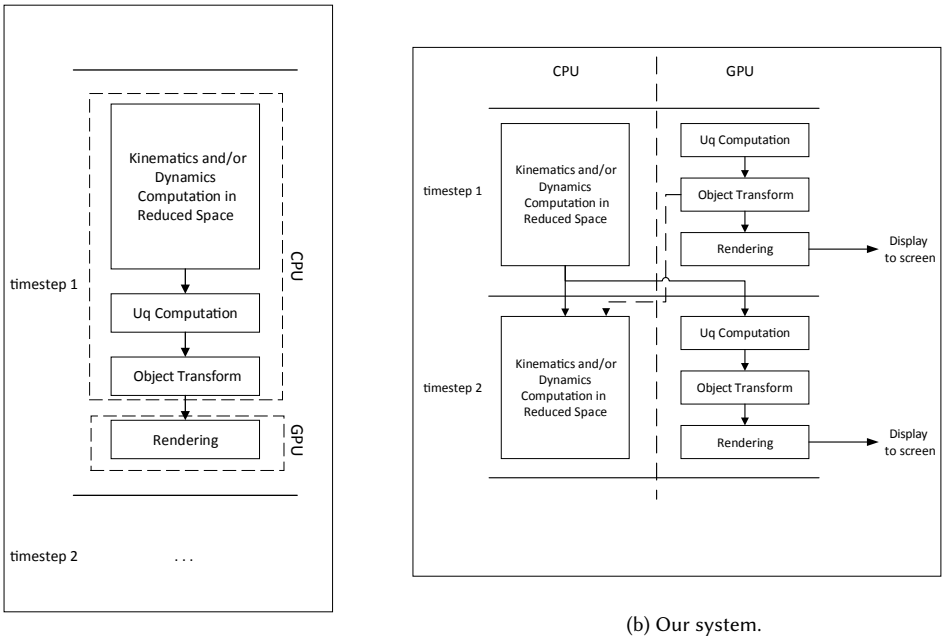
Model reduction is commonly used to accelerate physically based simulations in computer graphics and animation. It can be applied to 3D solids, cloth, plants, soft body characters, fluids and other systems. We propose a novel CUDA architecture for improving the runtime performance of simulation systems that use model reduction. We demonstrate our technique on multibody dynamics simulations whereby objects undergo large rotations and large model-reduced deformations, such as complex plant systems. The runtime computation of a typical model reduction timestep consists of two parts: (1) calculation of the current reduced coordinates q by timestepping a system of ODEs or some other simulation model, followed by (2) calculation of the mesh vertex displacements away from the neutral pose, $u(t) = Uq$, where $u = u(t) \in \mathbb{R}^{3n}$ are the displacements of the n mesh vertices, $U \in \mathbb{R}^{3n \times r}$ is the *modal matrix* and $q = q(t)$ are the reduced coordinates. In model reduction systems, step (1) typically entails timestepping a low-dimensional dynamical system and is commonly performed very rapidly and independently of the mesh complexity, e.g., in times on the order of (or under) 1 millisec. Step (2) depends on the mesh complexity and is often the computational bottleneck of model reduction systems at runtime. In our work, we give a CUDA approach to greatly speed up step (2). Model reduction systems typically perform step (2) on the CPU (Figure 2a). Our system, however, performs step (2) using an efficient CUDA algorithm on the GPU. Our experiments demonstrate that step (2) is often the bottleneck of the entire runtime computation and therefore our method gives substantial overall system speedups compared to a CPU implementation. We achieve the speedups by grouping CUDA multiplications to avoid bank conflicts, maximizing CUDA thread utilization, re-shaping and pooling modal matrices of multiple objects, and optimizing the memory data layout for maximum performance. To the best of our knowledge, no prior work has explored how to rapidly deform many model-reduced meshes using the GPU. Our method enables the CPU and GPU to work simultaneously and asynchronously, and reduces the amount of communication between the CPU and GPU. Together, these improvements greatly decrease the overall time cost of each model reduction simulation timestep. An overview of our system is shown in Figure 2b.

2 RELATED WORK

In computer graphics, early applications of model reduction to deformable object simulation were for linear systems [Hauser et al. 2003; James and Pai 2002; Pentland and Williams 1989]. The modal

MIG '20, October 16–18, 2020, Virtual Event, SC, USA

ACM ISBN 978-1-4503-8171-0/20/10…$15.00
https://doi.org/10.1145/3424636.3426895



(a) Classical system.

(b) Our system.

Fig. 2. The workflows of two timesteps in a CPU-based "classical system" that does not use CUDA computation (a), and in our CUDA-enabled system (b).

matrix U in these methods is computed using *linear modal analysis*; the resulting simulations are fast but only useful under small deformations. Large deformation modeling can be achieved using nonlinear systems. Our method is agnostic of the deformation magnitude and works both for linear and nonlinear systems. Model reduction of nonlinear systems has been used for fast simulation of deformable solids [An et al. 2008; Barbič and James 2005; Kaufman et al. 2008; Kim and James 2009; Metaxas and Terzopoulos 1992] and fluids [Treuille et al. 2006; Wicke et al. 2009], and for fast control of such systems [Barbič et al. 2009]. It has also been used to simulate the soft-body dynamics of characters [Kim and James 2011; Xu and Barbič 2016], for fast sound simulation [Chadwick et al. 2009; James et al. 2006] and for frictional contact between deformable objects [Kaufman et al. 2008] and for plant dynamics [Barbič and Zhao 2011; Wang et al. 2017; Zhao and Barbič 2013]. Yang et al. investigated how to accelerate the model reduction preprocessing [Yang et al. 2015]. Hildebrandt and colleagues [Hildebrandt et al. 2011] used model reduction for shape interpolation, whereas Harmon and Zorin [Harmon and Zorin 2013] showed how to efficiently handle localized deformations. Model reduction can also be combined with boundary component mode synthesis [Yang et al. 2013], full-space simulation [Teng et al. 2015] and projective dynamics [Brandt et al. 2018].

To date, there has been little work on computing modal deformations on GPUs. James and Pai [2002] gave a system whereby the Uq multiplication is performed in a fragment shader using GPU assembly language instructions, whereas Barbic [2007] used pbuffers and the Cg shading language to achieve similar results. These technologies are considered obsolete by modern standards. Furthermore, these previous papers simply calculated the matrix vs vector product on the GPU directly, without any consideration for memory storage, memory bank conflicts or pooling of computation for better performance. They also only addressed a single object. In our work, we address both single and multiple objects, using a modern GPU programming technology (CUDA), and we extensively discuss how to lay the data in GPU memory to maximize memory throughput. CUDA is widely used in physically based modeling [PhysX 2008; Tang et al. 2018; Zhang and Shen 2013]; but we are unaware of any previous application to computing modal deformations in model reduction.

3 BACKGROUND: MODEL REDUCTION

Model reduction starts with a high-dimensional dynamical system such as FEM deformable dynamics for a 3d solid object with $n >> 1$ vertices, whereby each mesh vertex has three deformable degrees of freedom (DOF). It then projects the dynamics to a suitable low-dimensional space spanned by the columns of a matrix $U \in \mathbb{R}^{3n \times r}$, and then the vertex displacements $u \in \mathbb{R}^{3n}$ are approximated as $u = Uq$, for some time-varying vector of reduced coordinates $q = q(t) \in \mathbb{R}^r$. As commonly done, we treat U as a dense matrix; different objects can have different matrices U. In real-time applications, the dimension r is typically small, i.e., common choices are $r = 5, 10, 20, 30$ or similar. The vector q satisfies some low-dimensional ODE, which, in a generic form, can be expressed as

$$\ddot{q} + A(q)\dot{q} + f(q) = g(t), \tag{1}$$

where A and f are nonlinear functions of q. Our work is agnostic of the specific format of this equation; and as a matter of fact, the values $q = q(t)$ can come from any of the model reduction variants, e.g., a basis from data [Krysl et al. 2001], modal derivatives [Barbič and James 2005], boundary modes [Yang et al. 2013], nonlinear inertia derivatives [Yang et al. 2015], multi-domain dynamics [Wang et al. 2017], domain decomposition [Kim and James 2011], etc. We use the CPU to timestep Equation 1. The specifics of this integration depend on the employed model reduction method; we use implicit Newmark [Wriggers 2002]. At the end of each timestep, new reduced coordinates q, velocities $\dot{q}$ and accelerations $\ddot{q}$ become available. Our CUDA kernels then calculate new vertex displacements u in the local frame of the object; and optionally if the object is undergoing rigid body motion, also the world-coordinate position of each vertex. The computed positions are then used for rendering and optionally contact handling. We did not pursue contact handling in our system. Figure 2b gives a block diagram of the computation in our system. Compared to our system, CPU-only methods are simpler (Figure 2a), but do not utilize the GPU (except for rendering). This means that when the CPU is working, the GPU may be idle, and vice versa. In our system, the GPU uses the most recently CPU-computed reduced coordinates q and can compute the displacements u in parallel with CPU's other work. Furthermore, our method reduces the CPU-GPU data transfer and as such benefits rendering. This is because only q and the 3×4 transformation of each object need to be passed from the CPU to GPU at each frame, as opposed to vertex positions.

4 CUDA DEFORMERS

We now describe our CUDA deformer to calculate the mesh vertex displacements $u = Uq$. We do not use 8-bit or 16-bit floating points because they often lead to visible quantization errors. We use 32-bit single-precision floating-point computation as opposed to 64-bit double-precision. There are three reasons for this. First, 32-bit single-precision floating point operations are much faster than 64-bit operations: the Nvidia manual [Nvidia 2020] (5.4.1 Arithmetic Instructions, Table 3) lists 32-bit floating point as 16x to 64x faster than 64-bit on a typically Nvidia GTX/RTX GPU. Second, the memory fetch and store instructions for a 4-byte datatype ("a 32-bit word") are faster than for a 8-byte datatype because of the data size. In interactive applications, the final vertex displacements are used only for rendering and collision detection; and therefore high precision is not required. Especially for rendering, double-precision is rarely used in OpenGL and its performance is significantly worse than single-precision. We note that Nvidia provides a matrix multiplication library cuBLAS. However, the matrices in modal reduction have a special property, namely they are thin. With multiple objects, our method is on average 29× faster than cuBLAS (see Table 4), as cuBLAS simply cannot do a large number of thin matrix-vector multiplications efficiently. Even for the single-object scenario, we observed a 20% speedup.

4.1 Computing displacements of a single object

If there is only one model-reduced object in the system, there is only one U matrix and one q vector to be multiplied in each timestep. For all vertices, the displacement u can be computed by $u = Uq$, where U is a modal matrix with $3n$ rows and r columns. The GPU memory storage and layout of the matrix U and vector q are critical for the algorithm performance. The displacement of degree of freedom ℓ of the mesh is

$$u_\ell = \sum_{j=1}^{r} U_{\ell,j}\, q_j, \tag{2}$$

where $U_{\ell,j}$ is the element in row ℓ and column j of matrix U. As follows from Equation 2, u_ℓ is computed as a series of multiplications followed by a summation. The multiplications are independent of each other and can be performed in parallel. We distribute the multiplications into separate CUDA threads. Note that on modern GPUs, the GPU always launches CUDA threads in "batches" called warps, and the number of threads in a warp is 32. In each thread, the program fetches a part of the U matrix and a part of the q vector and multiplies them together. We limit the dimension of q to be under 32, which is a common choice in real-time applications where system speed is more important than accuracy. Therefore, the calculation of each degree of freedom of u can be computed within 32 threads, i.e., one CUDA warp. Finally, the computed degree of freedom u_ℓ is stored into global GPU memory.

We now describe this process in greater detail, as the specific data memory organization makes a large difference in the resulting system performance. Our first observation is that in order to compute each degree of freedom of u one does not need to just multiply the entries of U with entries of q, but also perform the summation of the products. Whereas the multiplications are very parallel, the summations are not and can easily decrease the parallelism. Suppose that, for the sake of illustration, we have $r = 30$. A naive algorithm performs the 30 multiplications in parallel, and then a single thread adds the resulting products together. This leads to GPU under-utilization and a decrease in performance. We speedup the process by performing four multiplications per thread, and also having each thread sum the four products. Let ℓ be a degree of freedom of the mesh, $0 \le \ell < 3n$. Denote the k-th 4-tuple of q, U_ℓ and u_ℓ by q^k, U_ℓ^k and u_ℓ^k, respectively. Then, Equation 2 is rewritten as follows:

$$q^k = (q_{4k},\, q_{4k+1},\, q_{4k+2},\, q_{4k+3})^T, \tag{3}$$

$$U_\ell^k = \left(U_{\ell,4k},\, U_{\ell,4k+1},\, U_{\ell,4k+2},\, U_{\ell,4k+3}\right)^T, \tag{4}$$

$$u_\ell^k = U_\ell^k \cdot q^k, \quad u_\ell = \sum_k u_\ell^k. \tag{5}$$

After the 4-vector multiplication, the local summation of the four elements is performed by each thread, improving concurrency. In our $r = 30$ example, we now only have to add 8 floating point values for each ℓ, whereas we previously needed 30. In this way, we can utilize four times more active threads per warp to perform the summation. Because the multiplications are grouped into batches of four, each warp can compute more than one u_ℓ. The number of threads needed for computing one u_ℓ is $N = \left\lceil \frac{r}{4} \right\rceil$, and therefore the number of degrees of freedom of u that can be computed in a warp is $\left\lfloor \frac{32}{N} \right\rfloor \approx \frac{128}{r}$. Another reason to process 4 multiplications at a time (as opposed to a higher number) is that higher numbers require loading more U_ℓ^k entries per warp from global GPU memory. This increases the overhead of global memory operations. Quantities u_ℓ^k

are stored into shared memory as well. The final summation result u_ℓ is stored into global GPU memory.

We also considered splitting vectors q into sizes other than 4 (the "batch size"). In order to evaluate how the batch size affects the computation cost, we performed our CUDA kernel computation on an object with 1,000,000 vertices using $r = 16$ and $r = 32$, and using several batch sizes. The experiment demonstrates that our choice of batching q into arrays of 4-vectors is optimal (Table 1).

Table 1. The time cost (in [ms]) of the Uq computation under different batch sizes k on a model with 1,000,000 vertices, under two basis sizes, $r = 16$ and $r = 32$. We choose $k = 4$ in our paper.

r \ k	1	4	8	16	32
16	0.426	**0.138**	0.248	0.228	0.232
32	0.936	**0.255**	0.274	0.434	0.807

Accessing GPU memory is often the bottleneck of CUDA programs. Designing a good memory layout is therefore crucial for performance. Because there is only one object, the computation of each u_ℓ uses the same q. To avoid loading q multiple times, each block caches q explicitly from global memory to shared memory once at the beginning of the computation, using the __shared__ CUDA keyword. In all CUDA programs, The CUDA computation (the "grid") is organized into blocks, each of which contains a certain fixed number of threads, typically a multiple of the warp size (32). In our system, the number of threads in a block is at least 512. We view the threads of a GPU block as logically organized into a 2D array; call its dimensions $I_{max} \times J_{max}$, and denote the GPU threads by thd_{IJ}, where $0 \le I < I_{max}$ and $0 \le J < J_{max}$. We set J_{max} to the number of the threads in a warp (32), and therefore, for 512 threads, we have $I_{max} = 16$. We do this because a warp is the basic execution unit and because this avoids bank conflicts.

The shared GPU memory is organized into banks. Each CUDA block has its own shared memory. A bank conflict in GPU shared memory occurs when two distinct threads try to access distinct elements in the same bank. Such conflicts greatly slow down the GPU performance, and our method avoids them as follows. Note that a bank conflict does *not* occur if two threads access the *same* 32-bit word in a bank; only if different bank words are accessed. Denote the shared memory base address that stores q by $\text{addr}(q)$. We align this address to a 128-byte boundary for improved performance. To avoid bank conflicts, we store q_i at

$$\text{addr}(q_i) = \text{addr}(q) + \left((i \bmod 4) \times 32 + \left\lfloor \frac{i}{4} \right\rfloor\right) \times 4. \qquad (6)$$

The storage layout of q is shown in Figure 3. Note that even though q is stored sequentially in shared memory, the bank conflicts are avoided. To load q into shared memory, threads thd_{IJ} where $4J+I \le r$ first load q_{4J+I} into the corresponding shared memory address; the other threads do nothing at this stage. Each thread $\text{thd}_{IJ}, 0 \le I < I_{max}, 0 \le J < J_{max}$, can then fetch q^k from shared memory and the corresponding U_ℓ^k. The value q^k can be read from the shared memory because q has already been loaded from the global GPU memory to the GPU shared memory as described above. In contrast,

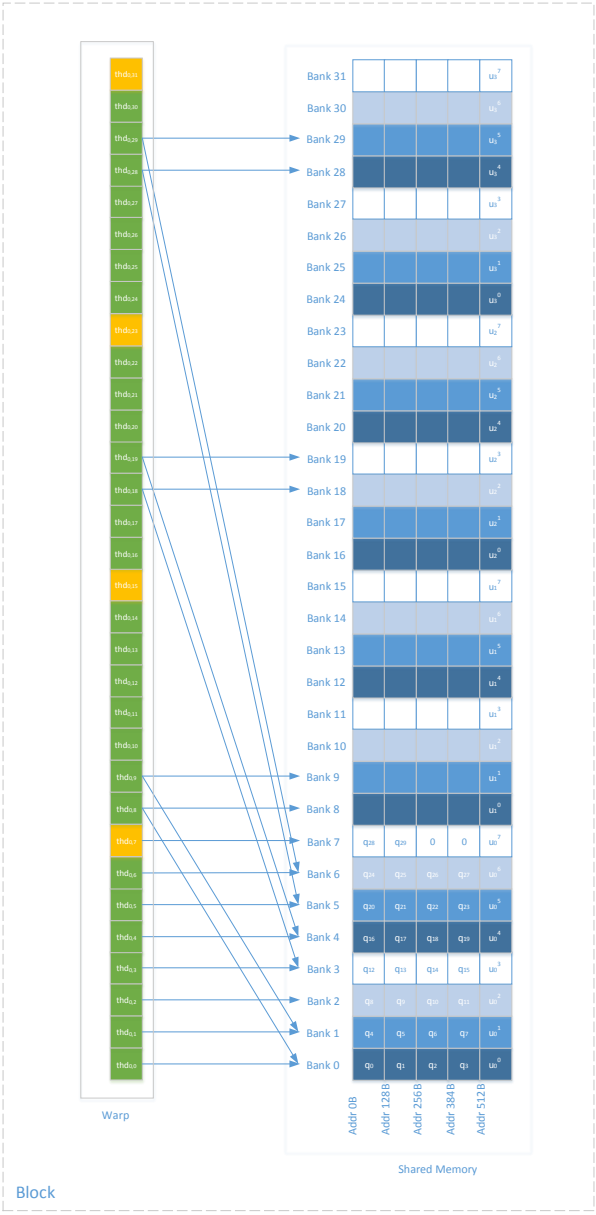


Fig. 3. Shared memory layout and thread access overview. In this figure, $r = 30$. A warp can compute four u_ℓ values. The arrows indicate the shared memory banks accessed by individual threads. The summations are performed by the threads highlighted in yellow.

since U_ℓ^k is used only once per Uq computation, we do not load U into the shared memory in advance; instead, U_ℓ^k is loaded directly from the global memory. To enhance the loading performance, U_ℓ are stored in row-major layout in GPU memory. Since one warp can compute several u_ℓ, we pack a series of continuous U_ℓ into one larger vector. We 128-byte align the base address, as shown

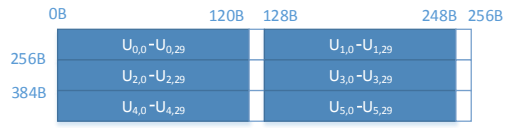


Fig. 4. The layout of matrix U in global memory for $r = 30$. Each row is 128-byte aligned.

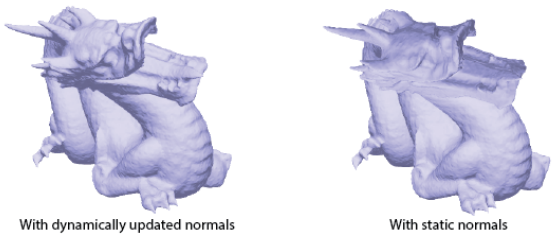


Fig. 5. Rendering with and without dynamically updated normals.

in Figure 4. Denote the base shared memory address for storing the intermediate result u_ℓ^k by addr(u_{shared}). Thread thd$_{IJ}$ computes $u_\ell^{J \bmod N}$ and stores it at

$$\text{addr}(u_\ell^{J \bmod N}) = \text{addr}(u_{\text{shared}}) + (I \times 32 + J) \times 4. \quad (7)$$

Figure 3 shows the memory layout of addr(u_ℓ^k). After all u_ℓ^k are computed, u_ℓ is finally computed using a summation, by a subset of the threads in a warp (shown in yellow in Figure 3). Specifically, it is the threads thd$_{IJ}$ with $J \bmod N = 0$ that perform the summation. For convenience of final vertex world position computation, u_ℓ is stored as a 4-element vector rather than 3-element vector, and is stored sequentially in global memory.

4.2 Computing dynamic normals

Using static normals produces visible artefects with deformable objects (Figure 5), and therefore real-time dynamic normals are highly desirable. To compute dynamic normals of each vertex, we first compute the area-weighted face normals of the neighboring faces, followed by summation and normalization. Table 2 gives the dynamic normal computation time. The normals are computed by executing a CUDA kernel that computes the face normal of each face and stores it into GPU memory. Next, we issue a second kernel that sums the face normals for each vertex, followed by normalization. Alternatively, one can employ a single CUDA kernel that computes the face normals of the neighboring faces of each vertex and sums them to the vertex normal. The second method has fewer memory operations but performs more duplicated arithmetic operations. We choose the first method to compute dynamic normals, as we experimentally measured it to be much faster than the second method on our dragon example (Table 2).

Table 2. **The time cost (milliseconds) of computing dynamic normals for a single object.** When using the CPU, one does not only need to compute the normals, but also needs to transfer the data to the GPU memory for rendering. Accordingly, we give two numbers for the CPU: the computation time and the CPU-GPU data transferring time. On the other hand, GPU does not need to transfer the data between CPU and GPU memory.

CPU	single-stage GPU	**two-stage GPU**
7.1 + 4.7 = 11.8	10.5	**6.1**

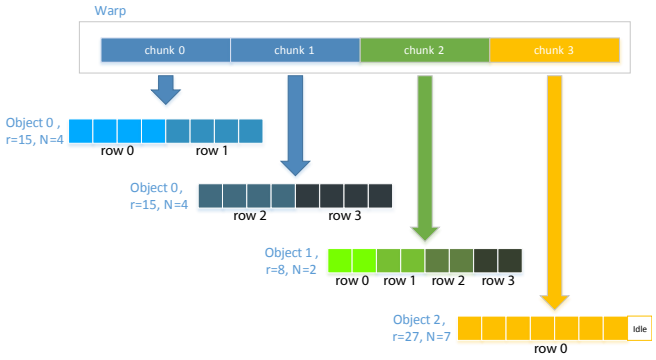


Fig. 6. Addressing objects with different dimensions r. In this example, the chunk size is $C = 8$, and there are four chunks in a warp. For object 0, $N = 4$ threads are needed to compute each vertex displacement DOF. Therefore each chunk can process two rows. In this example, two chunks are allocated to the first object. For object 1, $N = 2$ and we allocated one chunk to it. For object 2, $N = 7$ and we also allocated one chunk to it. Therefore, the warp computes 4, 4, 1 displacement DOFs of objects 0, 1, 2, respectively.

4.3 Computing displacements of multiple objects

Simulating multiple model-reduced objects poses several challenges compared to single-object simulation. Different objects i have different modal matrices $U^{(i)}$ and reduced coordinate vectors $q^{(i)}$. One does not just have different entries of $U^{(i)}$ and $q^{(i)}$, but instead different objects i have different dimensions of $U^{(i)}$ and $q^{(i)}$. Although one could simply use the single-object algorithm of Section 4.1 on each object, doing so is extremely inefficient when there are many objects present. We now give a parallel GPU algorithm to accelerate the $U^{(i)}q^{(i)}$ computations for multiple objects.

Consider first the simple case where the dimension r of each object is the same. This case is easy to address and can be solved by adapting the single-object algorithm. Each thread block now needs to load more than one q because vectors q are different for different objects. Because the number of columns is the same for all objects, we can stack the individual U matrices into one large matrix U. Each thread performs the same work as before, except the memory addresses of U and q are different for each object.

To address the general case where r is different for different objects, we "uniformize" the different values of $r^{(i)}$. The key idea is that we can reshape matrix $U^{(i)}$, as follows. Pick an integer $Q^{(i)} \geq 1$ and append rows $2, \ldots Q^{(i)}$ to row 1; append rows $Q^{(i)}+2, \ldots Q^{(i)}+Q^{(i)}$ to row $Q^{(i)}+1$, etc. This produces a reshaped matrix whose number

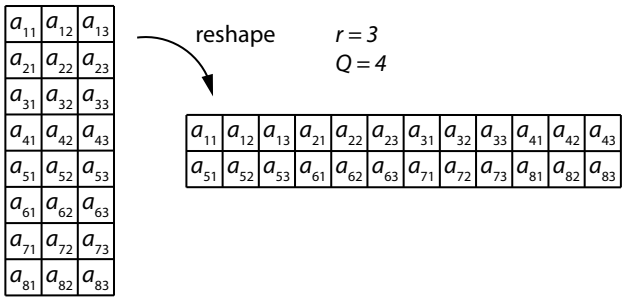


Fig. 7. **Reshaping the matrix to minimize the number of CUDA kernel invocations.**

of rows and columns is $Q^{(i)}$ times smaller, and larger, respectively (see Figure 7). We can now also replicate the reduced coordinate vector $Q^{(i)}$ times. We then multiply the reshaped matrix with the replicated reduced coordinate vector, except that we only perform the multiplications and defer the summation, as in the single-object case. By employing dedicated summation threads, we can compute the DOFs of the displacement vector $u^{(i)}$. The advantage of reshaping is that it minimizes the number of CUDA kernel invocations, and therefore speeds up the CUDA computation. This is because the reshaping uniformizes the number of columns of U across all the objects. Therefore, we can perform the $u = Uq$ computation using a single (or a few) number of kernels, as opposed to having a dedicated CUDA kernel for each value of r. In our examples, our method is on average 4× faster than processing each distinct r individually, and 25× faster than processing each object using a single-object method of Section 4.1.

The first "naive" approach to setting $Q^{(i)}$ is to set it so that we have $Q^{(i)}N^{(i)} \approx 32$. The logic behind this equation is that we need $N^{(i)}$ threads to process each DOF, hence each warp can process $Q^{(i)} = \left\lceil \frac{32}{N^{(i)}} \right\rceil$ DOFs. Hence, we need to arrange $Q^{(i)}$ rows in the column direction. We tried this approach and it is inefficient due to too many idle threads. Therefore, we need a finer-granularity control over $Q^{(i)}$. We achieve this by dividing the 32 threads of a warp into equally sized "chunks" of size $C^{(i)} \leq 32$. A chunk computes several vertex displacement DOFs of the same object, but not of different objects. The entire chunk must be processed by a single warp. We first group four consecutive entries in each row of $U^{(i)}$ into a single unit, to be processed by a single thread, as in the single object algorithm. Matrix $U^{(i)}$ can now be seen as a "condensed" $3n \times N^{(i)}$ matrix whereby each entry is a 4-tuple. The number of vertex displacement DOFs that a chunk can compute is $Q^{(i)} = \left\lfloor \frac{C^{(i)}}{N^{(i)}} \right\rfloor$. We therefore reshape the modal matrix $U^{(i)}$ and vector $q^{(i)}$ using the factor $Q^{(i)}$ as described above. The reshaped matrix has $C^{(i)}$ columns and $\left\lceil \frac{3n^{(i)}}{Q^{(i)}} \right\rceil$ rows. The process is illustrated in Figure 6.

Because there are many chunks in each block, a block may process one or more objects. Therefore, a block needs to access the reduced coordinates for those objects. Because the threads in a warp may load the $q^{(i)}$ vector for different objects, we do not cache $q^{(i)}$ in shared

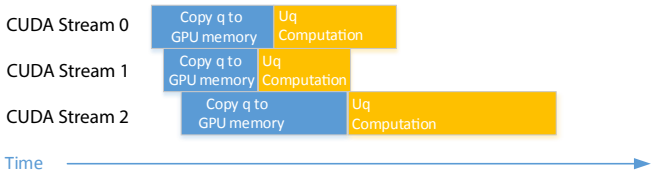


Fig. 8. Asynchronous kernel execution. In this example, there are three separate chunk sizes; each chunk size is given its own kernel and memory copy and is sent to its own stream. In our actual system, we only use two chunk sizes (6 and 8), and hence we only employ 2 streams.

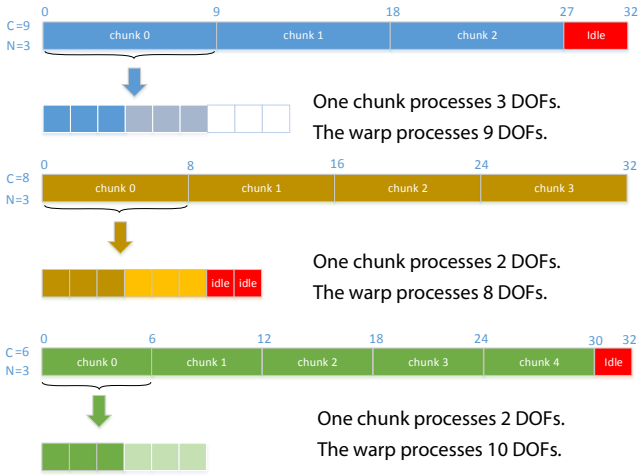


Fig. 9. Effect of changing the chunk size C. In this example, $N = 3$ threads are needed to compute each vertex displacement DOFs. When $C = 9$, each chunk can process $C/N = 3$ DOFs. Because there are 3 chunks in a warp, a warp can process $3 \times 3 = 9$ DOFs. Similar calculations are performed for $C = 8$ and $C = 6$. In this example, $C = 6$ is the best design.

GPU memory. Instead, we directly load them from global GPU memory. This is because the re-usability of $q^{(i)}$ is much lower than with single-object computation. When $U^{(i)}q^{(i)}$ is computed, threads need to know which $q^{(i)}$ should be loaded, where the compute displacement DOFs should be stored, and $N^{(i)}$. Accordingly, we store this information into a meta-data vector to assist with the $U^{(i)}q^{(i)}$ computation for each thread. The output displacements u are stored in the same way as in the single object simulation, namely sequentially in memory for all objects.

We now describe how we select $C^{(i)}$. This choice is very important as it influences the performance of the algorithm and the utilization of the GPU. Values of $C^{(i)}$ greater than 8 are not recommended because then a warp can contain at most 3 chunks which leads to many idle threads. Figure 9 illustrates this problem. In contrast, too small values $C^{(i)}$ cannot accommodate many different values of $N^{(i)}$. For efficient thread utilization, we use a different C for different objects. Because $r^{(i)} \leq 32$, we have $1 \leq N^{(i)} \leq 8$. To fully occupy the chunks, we use $C = 8$ when N equals 4 and 8, and $C = 6$ when N equals 1, 2, 3 or 6. For $N = 5$ and $N = 7$, we first attempted to use $C = 5$ and $C = 7$, respectively, leading to four

different chunk sizes. This strategy guarantees that there are no idle threads in any chunk. Note that it is possible to use $C = 8$ when N equals 1 or 2. However, we experimentally determined that this choice is not as efficient as $C = 6$ (Table 3). Each chunk size requires writing a separate CUDA kernel for efficiency. We can employ the CUDA streaming technology to parallelize kernel execution and data transfer (Figure 8). However, having more kernels comes at an increased time cost of preparing the data and launching the kernels. We experimentally determined that using 4 chunk sizes does not pay off (Table 3). Instead, using a single chunk size $C = 8$, or 2 distinct chunk sizes $C = 8$ and $C = 6$, gives the best performance over all examples. Employing 2 distinct chunk sizes $C = 8$ and $C = 6$ is more compact and better accommodates all values of N (especially $N = 5$). Therefore, we ultimately converged on using $C = 6$ when $N = 5$ and $C = 8$ when $N = 7$.

Table 3. **Evaluation of performance under various chunk sizes $C^{(i)}$.** Table gives the time cost of the Uq computation in milliseconds.

species \ $C^{(i)}$	8	6,8[1]	6,8[2]	5,6,7,8
Conifer (single)	0.18	0.16	0.18	0.20
Peach tree	0.38	0.61	0.62	0.73
Broad-leaved tree	0.29	0.29	0.72	0.32
Treesketch	0.29	0.29	0.30	0.30
Eastern hemlock	0.47	0.47	0.60	0.52

[1] : For $N^{(i)} = 1, 2, 3, 5, 6$, we have $C^{(i)} = 6$. For $N^{(i)} = 4, 7, 8$, we have $C^{(i)} = 8$.
[2] : For $N^{(i)} = 3, 5, 6$, we have $C^{(i)} = 6$. For $N^{(i)} = 1, 2, 4, 7, 8$, we have $C^{(i)} = 8$.

4.4 Multi-object transformations

The displacements u computed in Sections 4.1 and 4.3 are expressed in the local coordinate frame of each object. We now need to compute the vertex positions and normals in the world coordinate system, by taking into account the global position and orientation of each object. The resulting vertex positions and normals can then be used for rendering.

Triangle meshes are commonly used to represent objects in computer graphics. A triangle mesh is given by a list of triangles. The triangles are represented by their vertices. Denote the local coordinate frame displacement of vertex j in object i by $\mathbf{u_j^{(i)}}$, the vertex's world-coordinate position by $x_j^{(i)}$. Denote the rest (neutral mesh) position by $\overline{x}_j^{(i)}$. Also, denote the object's rigid rotation by $R^{(i)}$ and translation by $p^{(i)}$. The vertex's position is

$$x_j^{(i)} = R^{(i)} \left(\overline{x}_j^{(i)} + u_j^{(i)} \right) + p^{(i)}. \tag{8}$$

The rotation matrix $R^{(i)}$ is a 3×3 matrix. We can use a 4×4 homogeneous matrix packing both rotation and translation. The equation of updating $x_j^{(i)}$ becomes

$$x_j^{(i)} = T^{(i)} \left(\overline{x}_j^{(i)} + u_j^{(i)} \right) \tag{9}$$

$$T^{(i)} = P^{(i)} R^{(i)}, \tag{10}$$

where $P^{(i)}$ is a 4×4 translation matrix. Accordingly, there will only be one matrix-vector multiplication and one vector addition for each vertex. Using a similar idea as in Uq computation, we load commonly used data to shared GPU memory. Transformation matrix $T^{(i)}$ and the undeformed mesh positions are loaded into shared memory. Positions are represented as 4D vectors in homogeneous coordinates. Each matrix-vector multiplication therefore consists of 4 dot products. Each thread computes one dot product, i.e., one DOF of a vertex position. Therefore, each warp calculates eight vertices.

We assume that each object has at least 4 vertices, so that each warp processes at most two objects; this avoids bank conflicts. The transformation matrix is stored row major and sequentially in the shared memory, so there are not any read overheads. Additionally, there aren't any bank conflicts when reading the matrices. In addition to the matrix, the rest position of each vertex is loaded into shared memory before the computation, because they are frequently used by matrix multiplication. They are separately stored in shared memory so that each thread in a warp can read the data from the corresponding bank without any conflicts. The shared memory layout is shown in Figure 10. Every time the computation starts, the program first loads the data to shared memory. Thread thd_{IJ} loads the $(J \bmod 16)$-th element in $\left(2I + \left\lfloor \frac{J}{2} \right\rfloor\right)$th transformation matrix of the block. Because each block may load different transformation matrices, we precompute (during initialization) the specific data to be loaded by each thread. After matrix is loaded into shared memory, we load the position vectors into shared memory. Thread thd_{IJ} in block k loads the $(J \bmod 4)$-th element of the $\left(Nk + 8I + \left\lfloor \frac{J}{4} \right\rfloor\right)$-th vertex, where N is the number of vertices each block can compute; this value is constant for all the blocks. When the positions are fetched from global memory, we add the displacements to them before storing them to shared memory. Before the matrix-vector multiplication, we need to synchronize all threads inside the block to guarantee that everything the computation needs has been successfully loaded into the shared memory. After the data is loaded into shared memory, each thread calculates one degree of freedom of a vertex and stores the result back to global memory. After the vertex position are calculated, we dynamically update the normals of all vertices in all objects using the algorithm of Section 4.2.

4.5 Data communication

Figure 11 shows the data communication between and within devices. Packed U matrices and vertex rest positions and normals are constant at runtime so they are copied from the CPU memory into the GPU memory when the system is initialized. Quantities q, u, T and the vertex position buffer are passed between the CPU and the GPU at each timestep. To facilitate the data transfer, we use page-locked memory allocated by cudaHostAlloc. In each timestep, CPU computes new q, packs it for CUDA computation and stores it to the host (CPU) q buffer. CPU only writes to the q buffer and therefore we designate the host q buffer as write-combine memory [Intel 1998]. Because of the write-combining ability, write performance is greatly improved. After the CPU q buffer is updated, the data will be transferred to GPU memory. In our system, there are multiple Uq kernel executions and buffer copy operations. To improve the concurrency, we perform kernel executions and asynchronous memory copies using cudaStream, which parallelizes them (Figure 8). After the Uq computation, the resulting u is placed into the GPU u buffer.

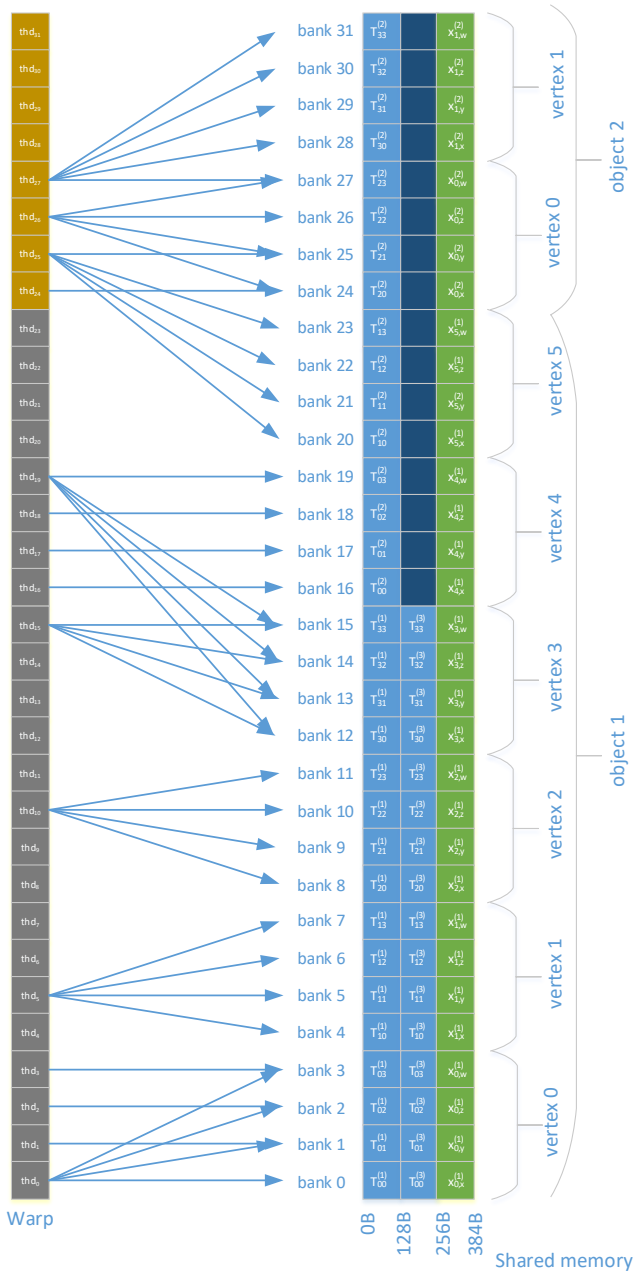


Fig. 10. The layout of the shared memory and the bank accesses of threads for computing vertex world-coordinate positions. Differently colored threads are processing different objects. Different shared memory colors denote different types of data: blue = transformation matrices, green = vertex positions. The x,y,z,w subscripts denote the components of the homogeneous 4-vectors.

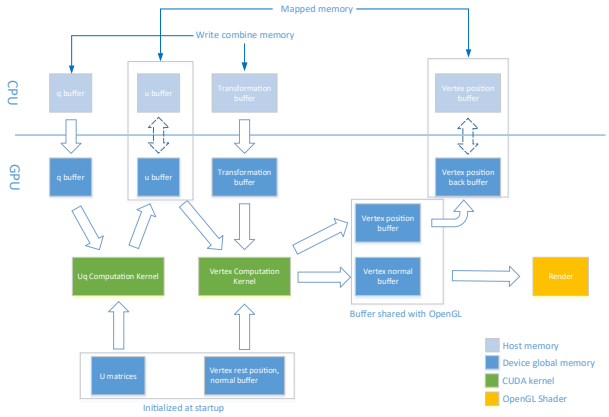


Fig. 11. Data communication between buffers and devices.

One only needs to copy this buffer back to the CPU if performing collision detection; otherwise, there is no need to copy because the computed u can be directly used for the subsequent rendering pipeline.

After the Uq computation, we compute the world-coordinate quantities for each vertex (Section 4.4). The deformation u is already stored in GPU memory. As per the transformation, we treat it in the same way as q, and therefore we designate the CPU transformation buffer to also be write-combined memory. Additionally, we update the GPU transformation buffer after the start of the asynchronous copying of the q buffer to the GPU. This ensures that transferring the transformation matrix data proceeds in parallel with the Uq computation. The world-coordinate positions and normals are shared by CUDA and OpenGL. CUDA gains control over the OpenGL buffer objects before writing the positions and normals to them. After the kernel executions are completed, CUDA unmaps the OpenGL buffers and then the rendering begins. While the GPU computes world-coordinate quantities, CPU cannot yet retrieve data from the GPU vertex position buffer. To make vertex positions viable during the GPU computation, we use two buffers for vertex positions. Swapping the buffers is fast because both buffers are in the same GPU memory. One of the position buffers is used to communicate with the CPU, and we designated it as mapped portable page-locked memory [Nvidia 2020]. In this way, the data is automatically synchronized to the CPU (without explicitly calling cudaMemcpy), and can be accessed by all the CPU cores.

5 RESULTS

We used an Intel Xeon W-3275 workstation (28 physical cores @ 2.5 GHz), with 192GB of RAM, and Nvidia RTX 2080 Ti. Tables 5 and 4 compare the performance of our system against a CPU implementation for a single object and multiple objects, respectively.

Because the floating-point multiplication and addition operations are not greatly affected by the data specifics, we use randomly generated inputs to test the Uq computation algorithm for a single object. The CPU algorithm uses Intel MKL with 56 threads (all CPU cores are used). For each (n, r) pair, the algorithm was tested via

Table 4. **Multi-object performance:** Table gives the time to perform a single dynamic reduced timestep to update the q values (q), the Uq computation time (Uq or cuBLAS Uq for cuBLAS library; Section 4.3), time for transformations (T; Section 4.4), frames per second (FPS), GPU memory usage (mem) and #kernels (#krnl) used in Uq computation. Note that only one of Uq or cuBLAS Uq is needed; we give both to provide a performance comparison. Conifer is shown in Figure 1 (here we give statistics for one conifer), and Eastern hemlock is shown in Figure 12. We give two numbers for the CPU Uq and T computations. The first denotes the unrealistic situation (performed for experiment only) whereby the CPU does not need to dynamically update the q values. The second denotes the actual time cost during the simulation that has to update q; as can be seen, the CPU time is heavily affected. Column #objects gives the number of deformable objects in each example; the number in parenthesis is the total number of objects, including deformable and rigid objects.

Species	geometry			CPU				GPU					
	#objects	total n	total r	q (ms)	Uq (ms)	T (ms)	FPS	Uq (ms)	cuBLAS Uq (ms)	T (ms)	FPS	mem (MB)	#krnl
Conifer (single)	43 (644)	7,543	360	16.73	0.11/12.84	0.36/3.15	35	0.16	0.46	0.26	60	5.8	1
Peach tree	237 (22,659)	273,003	2,950	29.41	0.70/34.92	0.85/36.88	4	0.61	2.52	0.53	20	119.6	2
Broad-leaved tree	419 (7,003)	288,542	3,613	35.01	0.76/36.29	1.68/18.66	6	0.29	4.41	0.75	29	150.9	2
Eastern hemlock	2,866 (27,778)	190,466	16,793	71.43	3.13/34.79	5.47/37.51	4	0.47	21.24	1.02	15	165.2	2
Treesketch	2,875 (2,875)	44,404	21,178	11.34	3.75/43.08	2.62/6.76	11	0.29	22.90	0.33	70	29.7	2

Table 5. Ratios of CPU to GPU running times for the single object Uq computation, under varying numbers of vertices (n) and modal dimensions (r).

r \ n	10^3	10^4	10^5	10^6	10^7
5	47.7	18.0	17.8	7.2	8.0
10	18.5	18.8	15.5	5.5	6.5
15	19.9	20.2	11.2	4.9	6.5
20	19.9	18.9	9.5	6.2	6.0
25	20.4	20.8	9.2	6.3	6.1
30	19.8	18.3	9.8	6.2	6.1

Fig. 12. Real-time simulation of this complex plant (27,778 deformable objects) at 14 FPS, thanks to our CUDA modal displacement deformer. Timestepping the reduced coordinates $q^{(i)}$ takes 71 msec. Observe that this corresponds to 14 FPS = 1 / (71 msec); this is possible because our method calculates Uq and the transformations in parallel with the CPU timestepping of $q^{(i)}$. For comparison, a CPU system runs at 4 FPS.

100 random iterations using the CPU method, and our CUDA GPU method. Table 5 gives the ratios of the average running times; it can be seen that the CUDA implementation is substantially faster than the CPU method. We also tested our dynamic normal CUDA implementation on our dragon example (Figures 5, 13). It is 2x faster than the multi-threaded CPU implementation as shown in Table 2.

We illustrate our multi-object technique on botanical model-reduced simulations on 5 different examples (Figures 1,12,14). Every part of tree morphology such as a leaf or a branch is a separate object. Substructuring is used to compute their model-reduced dynamics and world-coordinate rotations and translations [Barbič and Zhao 2011]. Results are given in Table 4. Because there are many objects (leaves, branches) in each example, the calculation of reduced coordinates q requires a significant amount of the CPU resources. As a result, there are few resources left for the Uq computation and object transformations; and hence, our CUDA deformer greatly speeds up the system. With our method, we observed an average speedup of 97x in the Uq computation, and 33x in transforming the objects. To remove the effect of dynamic timestepping of q, we also performed an experiment whereby only the Uq computation and object transformations were performed. Despite the fact that such a scenario rarely exists in practice, our method outperformed the CPU implementation on average by 5x in Uq computation, and 4x in transforming the objects. We also compared the time cost of our Uq computation with CUDA built-in cuBLAS, and measured an average speedup of 29x over cuBLAS in our multi-object examples. Moreover, as the number of objects and the reduced dimension increase, our time costs remain within the same order of magnitude. In contrast, the time cost of cuBLAS increases significantly.

ACKNOWLEDGEMENTS

This research was sponsored in part by NSF (IIS-1911224), USC Annenberg Fellowship to Bohan Wang, Bosch Research and Adobe Research.

6 CONCLUSION

Modal displacements are an important bottleneck in model-reduced systems. We gave a method to calculate modal displacements using CUDA, and demonstrated that this substantially accelerates real-time simulations that use model reduction. Our method works by carefully organizing the computation into larger chunks that better fit the architecture and memory characteristics of modern GPUs. We accelerated both individual model-reduced objects and multi-body dynamics simulations involving multiple model-reduced objects. Our method is limited to modal size of 32 and less. While this can accommodate most real-time model-reduced systems in

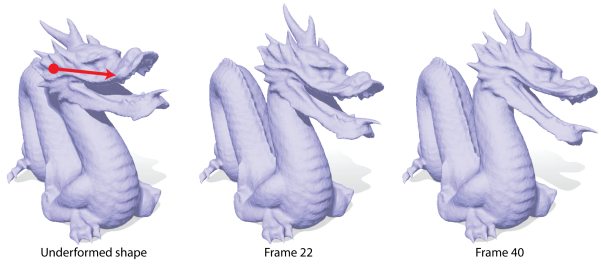


Fig. 13. Real-time simulation of a single model-reduced dragon with reduced dimension $r = 15$ and 455,568 vertices. The user pulled a single vertex (red dot) to the right, shown in the red arrow. Our CUDA-enabled simulation runs at 52 FPS, whereas the CPU system runs at 16 FPS. During each timestep, it only takes 0.5ms to compute the reduced displacements q. In addition, our method takes 1.0ms to compute Uq. For comparison, the CPU system takes 3.1ms to compute Uq.

Fig. 14. Broad leaf model (left), peach tree model (middle) and treesketch model (right).

practice, future work could explore how to relax this requirement. We only calculated modal displacements on the GPU, whereas the low-dimensional dynamics is still performed on the CPU. Although this is typically not an issue, it would be interesting to explore how to timestep low-dimensional dense dynamical systems using CUDA. We did not investigate collision detection and response in our work. For CPU collision detection and response, the computed vertex positions must be copied back to the CPU. Although this incurs data transfer costs, our method greatly accelerates the computation of vertex positions, which would otherwise need to be done on the CPU. GPU collision detection and response has been widely studied [Tang et al. 2011], and can benefit from our method because the vertex positions are already on the GPU, similarly to rendering.

REFERENCES

Steven S. An, Theodore Kim, and Doug L. James. 2008. Optimizing Cubature for Efficient Integration of Subspace Deformations. *ACM Trans. on Graphics* 27, 5 (2008), 165:1–165:10.

Jernej Barbič. 2007. *Real-time Reduced Large-Deformation Models and Distributed Contact for Computer Graphics and Haptics.* Ph.D. Dissertation. Carnegie Mellon University.

J. Barbič, M. da Silva, and J. Popović. 2009. Deformable Object Animation Using Reduced Optimal Control. *ACM Trans. on Graphics* 28, 3 (2009).

J. Barbič and D. L. James. 2005. Real-time subspace integration for St. Venant-Kirchhoff deformable models. *ACM Trans. on Graphics* 24, 3 (2005), 982–990.

J. Barbič and Y. Zhao. 2011. Real-time Large-deformation Substructuring. *ACM Trans. on Graphics (SIGGRAPH 2011)* 30, 4 (2011), 91:1–91:7.

Christopher Brandt, Elmar Eisemann, and Klaus Hildebrandt. 2018. Hyper-Reduced Projective Dynamics. *ACM Trans. Graph.* 37, 4, Article 80 (2018), 13 pages.

Jeffrey N. Chadwick, Steven S. An, and Doug L. James. 2009. Harmonic Shells: A practical nonlinear sound model for near-rigid thin shells. *ACM Transactions on Graphics (SIGGRAPH Asia 2009)* 28, 5 (2009), 1–10.

D. Harmon and D. Zorin. 2013. Subspace integration with local deformations. *ACM Trans. on Graphics (SIGGRAPH 2013)* 32, 4 (2013), 107:1–107:9.

Kris K. Hauser, Chen Shen, and James F. O'Brien. 2003. Interactive Deformation Using Modal Analysis with Constraints. In *Proc. of Graphics Interface.* 247–256.

Klaus Hildebrandt, Christian Schulz, Christoph von Tycowicz, and Konrad Polthier. 2011. Interactive Surface Modeling using Modal Analysis. *ACM Trans. on Graphics* 30, 5 (2011), 119:1–119:11.

Intel. 1998. Write Combining Memory Implementation Guidelines. http://download.intel.com/design/PentiumII/applnots/24442201.pdf

Doug L. James, Jernej Barbič, and Dinesh K. Pai. 2006. Precomputed Acoustic Transfer: Output-sensitive, accurate sound generation for geometrically complex vibration sources. *ACM Transactions on Graphics (SIGGRAPH 2006)* 25, 3 (2006).

Doug L. James and Dinesh K. Pai. 2002. DyRT: Dynamic Response Textures for Real Time Deformation Simulation With Graphics Hardware. *ACM Trans. on Graphics (SIGGRAPH 2002)* 21, 3 (2002), 582–585.

Danny M. Kaufman, Shinjiro Sueda, Doug L. James, and Dinesh K. Pai. 2008. Staggered Projections for Frictional Contact in Multibody Systems. *ACM Transactions on Graphics (SIGGRAPH Asia 2008)* 27, 5 (2008), 164:1–164:11.

Theodore Kim and Doug James. 2009. Skipping steps in deformable simulation with online model reduction. *ACM Trans. on Graphics (SIGGRAPH Asia 2009)* 28, 5 (2009), 123:1–123:9.

Theodore Kim and Doug L. James. 2011. Physics-Based Character Skinning Using Multi-Domain Subspace Deformations. In *Symp. on Computer Animation (SCA).* 63–72.

P. Krysl, S. Lall, and J. E. Marsden. 2001. Dimensional model reduction in non-linear finite element dynamics of solids and structures. *Int. J. for Numerical Methods in Engineering* 51 (2001), 479–504.

Dimitri Metaxas and Demetri Terzopoulos. 1992. Dynamic deformation of solid primitives with constraints. In *Computer Graphics (Proc. of SIGGRAPH 92).* 309–312.

Nvidia. 2020. CUDA C programming guide. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

Alex Pentland and John Williams. 1989. Good Vibrations: Modal Dynamics for Graphics and Animation. *Computer Graphics (Proc. of ACM SIGGRAPH 89)* 23, 3 (1989), 215–222.

PhysX. 2008. Nvidia. CUDA Fluid Simulation. www.nvidia.com/object/physx_new.html.

Min Tang, Dinesh Manocha, Jiang Lin, and Ruofeng Tong. 2011. Collision-Streams: Fast GPU-based collision detection for deformable models. In *Proc. of ACM Symp. on Interactive 3D Graphics and Games (I3D).* 63–70.

Min Tang, Tongtong Wang, Zhongyuan Liu, Ruofeng Tong, and Dinesh Manocha. 2018. I-Cloth: Incremental Collision Handling for GPU-Based Interactive Cloth Simulation. *ACM Transaction on Graphics (Proceedings of SIGGRAPH Asia)* 37, 6 (2018), 204:1–10.

Yun Teng, Mark Meyer, Tony DeRose, and Theodore Kim. 2015. Subspace Condensation: Full Space Adaptivity for Subspace Deformations. *ACM Trans. Graph. (SIGGRAPH 2015)* 34, 4, Article 76 (2015), 9 pages.

Adrien Treuille, Andrew Lewis, and Zoran Popović. 2006. Model reduction for real-time fluids. *ACM Trans. on Graphics (SIGGRAPH 2006)* 25, 3 (2006), 826–834.

Bohan Wang, Yili Zhao, and Jernej Barbič. 2017. Botanical Materials Based on Biomechanics. *ACM Trans. on Graphics (SIGGRAPH 2017)* 36, 4 (2017).

Martin Wicke, Matt Stanton, and Adrien Treuille. 2009. Modular Bases for Fluid Dynamics. *ACM Trans. on Graphics (SIGGRAPH 2009)* 28, 3 (2009), 39:1–39:8.

Peter Wriggers. 2002. *Computational Contact Mechanics.* John Wiley & Sons, Ltd.

Hongyi Xu and Jernej Barbič. 2016. Pose-Space Subspace Dynamics. *ACM Trans. on Graphics (SIGGRAPH 2016)* 35, 4 (2016).

Yin Yang, Dingzeyu Li, Weiwei Xu, Yuan Tian, and Changxi Zheng. 2015. Expediting Precomputation for Reduced Deformable Simulation. *ACM Trans. Graph. (SIGGRAPH Asia 2015)* 34, 6, Article 243 (2015), 13 pages.

Y. Yang, W. Xu, X. Guo, K. Zhou, and B. Guo. 2013. Boundary-Aware Multidomain Subspace Deformation. *IEEE Trans. on Visualization and Computer Graphics* 19, 10 (2013), 1633–1645.

J. Zhang and D. Shen. 2013. GPU-Based Implementation of Finite Element Method for Elasticity Using CUDA. In *2013 IEEE 10th Int. Conf. on High Performance Computing and Communications.* 1003–1008.

Yili Zhao and Jernej Barbič. 2013. Interactive Authoring of Simulation-Ready Plants. *ACM Trans. on Graphics (SIGGRAPH 2013)* 32, 4 (2013), 84:1–84:12.