# CUDA Deformers for Model Reduction

**Bohan Wang**

Department of Computer Science
University of Southern California

Advisor: Prof. Jernej Barbič

This dissertation is submitted for the degree of
*Master of Science*

Viterbi School of Engineering May 2015

I would like to dedicate this thesis to my loving parents for support and encouragement, my advisor Prof. Jernej Barbič for inspiration and guidance and Yili Zhao for the assistance and friendship.

# Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this thesis are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This thesis is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This thesis contains fewer than 65,000 words including references, tables and equations and has fewer than 150 figures.

Bohan Wang
May 2015

# Acknowledgements

This research was sponsored in part by the National Science Foundation (CAREER-1055035, IIS-1422869), the Sloan Foundation, and a donation of two workstations by the Intel Corporation.

# Abstract

The finite element method (FEM) with model reduction is widely used to simulate deformable objects. Improving the performance and rendering quality of deformable objects simulated using FEM with model reduction has been a critical problem for several years. As computer technologies rapidly evolve, GPUs have seen significant improvement. It is clear that modern graphics processing units (GPUs) play an important role in real-time rendering and parallel computing. The existing FEM model reduction rendering systems, however, utilize a large amount of CPU resources but relatively little GPU resources. Our experiments show that vertex position computations and mesh rendering account for a large percentage of the CPU's overhead in the existing system. Therefore, this work investigates how to efficiently employ GPUs for vertex position computation ($Uq$ computation) and rendering of deformable objects simulated using model reduction.

We first survey the related GPU technologies. Then we give a CUDA-based algorithm for $Uq$ computation and vertex global position computation, which we demonstrate to remarkably speed up the $Uq$ computation. We then give a new system architecture to reduce the communication between the CPU and GPU. Meanwhile, the system performance is further improved by making the CPU and GPU work in parallel. Additionally, a new rendering system based on modern OpenGL features is developed. It not only enhances the rendering performance but also improves the rendering quality. Finally, our experiments show that the proposed system outperforms the previous systems in terms of the computational cost as well as the rendering quality.

# Table of contents

# List of figures

# List of tables

# Chapter 1

# Introduction

This thesis is about accelerating elastic deformable object model reduction simulations using GPUs. In physics, partial differential equations are used to simulate deformations of solid objects. The finite element method (FEM) is a common discretization method to simulate deformable objects. Detailed meshes, however, result in a large number of deformable degrees of freedom. Suppose there are $n$ vertices in the volumetric mesh. Let $u$ be the displacement of all vertices of the volumetric mesh. Here $u$ is a vector whose size is $3n$. Therefore, there are $3n$ unknown degrees of freedom in the system. As the accuracy of FEM increases, the volumetric mesh becomes finer and thereby $n$ increases. As a result, for each time step, a large number of equations must be solved. Furthermore, the differential equations that describe the deformable object are nonlinear. Although the general FEM model is reasonably accurate, the simulation system is slow. To address this, such general FEM models can be simplified using dimensional model reduction. The basic idea is that the $u$ vector is transformed into the space with a much lower dimensionality. Denote the displacements in the reduced space by $q$. Full-space displacements $u$ are computed as $u = Uq$, where $U$ is an $3n \times r$ matrix and $q$ is a r-length vector. In general, $r$ is much smaller than $3n$. In our system, it is less than 32. Therefore, reduced deformable models can be simulated with little computational effort, but are not as accurate as the general FEM models. Every time after the equations are solved, $u$ is calculated by $u = Uq$. Next, any translations and rotations of the reduced deformable object are taken into account, producing global vertex positions. These positions are then used to render the mesh on the screen.

Suppose the mesh contains a large number of vertices and triangles. Using model reduction, we can greatly reduce the time cost of physical simulation. However, when we try to render the mesh on the screen, rendering alone can take a lot of time. This is because we need to first

compute (update) positions of all vertices, and then transfer them to the GPU at every time step. Without the consideration of the capacity of the main memory and the GPU memory, if a rigid mesh is rendered, we only need to transfer all vertices from the main memory to the GPU memory once, during the initialization of the rendering system. On the other hand, if a deformable object is rendered, we have to compute and update the vertex position in the GPU memory at each time step. This imposes a huge amount of communication for the system PCI-E bus between the main memory and GPU memory. Consequently, this leads to a decrease in the rendering speed. A similar phenomenon can be observed in computer games, where rigid objects are widely used rather than deformable objects, commonly due to the overheads associated with using deformable objects.

Graphics workstations with high-end CPU and GPU are widely used in graphics industries and laboratories. The $u = Uq$ computation and the vertex updating formulas have a highly parallel structure. By using the GPU, we can make the computation more efficient. Moreover, if $Uq$ computation and vertex updating are performed by the GPU, the amount of communication between the main memory and the GPU memory is greatly reduced. If all computation is performed by the CPU but only rendering is processed by the GPU, CPU may still suffer from a heavy workload. In contrast, GPU often has less tasks assigned to it and often stays idle. Accordingly, by moving some parts of the computation from CPU to GPU, the CPU overhead can be reduced and GPU can be better utilized.

As technologies rapidly develop in computer science, GPUs have seen significant improvement. Compute Unified Device Architecture (CUDA) makes it possible to manipulate GPUs more flexibly [15]. By using CUDA, we first move $Uq$ computation and vertex updating from the CPU to the GPU. It reduces not only the CPU workload but also the communications between the CPU and GPU. Additionally, by using the core OpenGL profile, we redesigned the rendering system to further improve the performance [16]. Finally, we provide a new simulator system architecture that makes CPU and GPU work synchronously. The experimental results show that our proposed system can accelerate the overall frame rate by about 5x in many cases. Furthermore, the quality of rendering is greatly improved.

The remainder of the thesis is organized as follows. Chapter 2 gives related work and preliminaries to the thesis. Next, the architecture of our simulation system and the corresponding algorithms are described (Chapter 3). Chapter 4 presents the experimental results. Finally, Chapter 5 provides a conclusion and future work.

# Chapter 2

# Related Work

## 2.1 Real-time Rendering

### 2.1.1 Shadows

Shadows greatly improve the scene realism and are very important for 3D rendering. Nonetheless, shadows consume a lot of GPU computation recourses. Shadow maps and shadow volumes are two classic algorithms for rendering shadows [5, 24]. Shadow volumes is a geometry-based algorithm. It is accurate but slow for very complex meshes such as trees. On the other hand, shadow mapping is one of the most popular algorithms for casting shadows in real-time applications due to fast computation. The classical shadow map algorithm can be considered as the father of several diverse modern shadow mapping algorithms. On the downside, shadow maps suffer from the problems of inaccuracy and aliasing. Figure 2.1a gives an example of aliasing along the shadow outline. Many researchers have tried to improve the shadow accuracy as well as performance.

Under different rendering conditions and environments, the algorithms behave differently. Daytime outdoor scenes are a common scene in practice. They have only one strong light source (the sun). The sun is a strong directional light that always casts clear shadows onto the scene objects. A critical issue of casting shadows in outdoor scenes is that the large scene area limits the accuracy of the depth map. In the shadow mapping algorithm, one needs to first generate a depth map from the light location – this is called a shadow map. The size of the shadow map is limited by the hardware's specification. Moreover, as the size of the shadow map increases, performance drops. To alleviate the problem, Stamminger et al. proposed perspective shadow maps (PSM) [22]. In PSM, the depth map changes according to the distance between

(a) Standard shadow maps. (b) Variance shadow maps.

Figure 2.1 Comparison between standard shadow maps and variance shadow maps.

(a) Standard shadow maps. (b) Parallel-split shadow maps.

Figure 2.2 Comparison between standard shadow maps and parallel-split shadow maps [12].

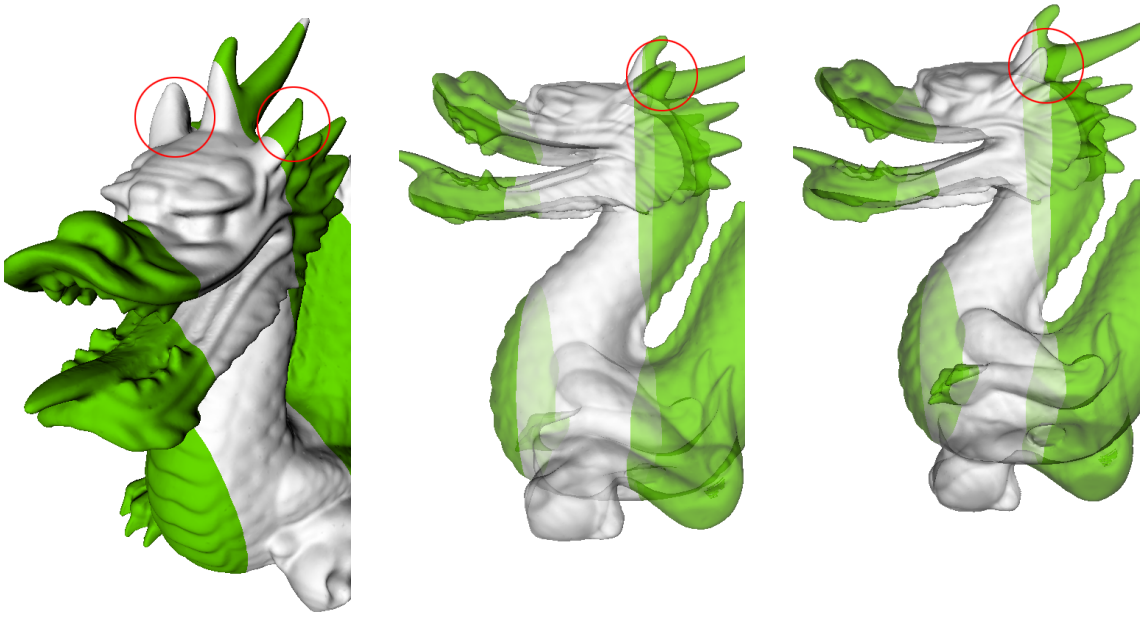
the objects and camera. If there are many objects that can be seen from the camera, the depth map may suffer from low resolution which can cause aliasing. To solve this problem, Zhang et al. proposed the idea that the scene can be unevenly split into several parts according to the distance to the camera, and the depth map is then generated separately, as illustrated in Figure 2.2 [26]. This approach is called parallel-split shadow maps (PSSM). The depth map of each part can now be more detailed. With equal size of depth maps, the smaller a part is, the higher resolution the depth map has. The closest part to the camera often occupies most of user's view, and needs the highest resolution of the depth map. The rendering performance decreases as the number of parts increases, because the scene is rendered more times to generate depth maps for each part. Related to this idea, if there is a key character in the scene we can generate another depth map specifically for the character. This depth map will give the character's shadows high quality. As the character moves, the depth map will follow its movement. In this case, PSM may further improve the quality of character's shadows.

High resolution of the depth map makes the shadows more accurate. In addition to accuracy, researchers tried to make shadows more realistic. The sun can be considered to emit directional light. Since it is not a point light, there is no penumbra (partial shadow) around umbra (complete shadow). Still, even in a sunny day the outline of shadows in outdoor scenes is not sharp due to the diffraction of light rays. Percentage closer filtering (PCF) provides an approach to soften the shadow boundary [20]. The insight is that it filters the results of depth comparisons instead of filtering the depths. It randomly samples the depth map in a small area. By depth comparisons, the percentage of how many points are unblocked can be calculated and is used to compute the shadow factor. The drawback is that it randomly samples the shadow map rather than sample it only once. Moreover, this approach cannot utilize the filtering functionality of the graphics hardware, because it needs to do depth comparisons before filtering. Additionally, many sample points are required in order to eliminate aliasing. Variance shadow maps (VSM) addresses the problem of filtering. It estimates the percentage by Chebychev's inequality. The algorithm is hardware-friendly. Additionally, it uses less computation effort but produces good soft shadows. Figure 2.1 gives a comparison between the standard shadow maps and variance shadow maps.

### 2.1.2 Transparency

The most common approach to rendering transparent objects is to render them back to front, using traditional alpha-blending equation

$$C = A_{src}C_{src} + (1 - A_{src})C_{dest}. \tag{2.1}$$



(a) Horn positions relationship. (b) Rendered with depth sorting. (c) Rendered without depth sorting.

Figure 2.3 Problems with alpha-blending.

This is also known as the **over** operator [19]. To improve the efficiency, we can pre-multiply the alpha value with the source color [11]. We do not need to store the alpha value for each intermediate pixel. Each iteration relies on current pixel's alpha value. Another common way is rendering objects from front to back. Then, the equations are

$$\begin{aligned} C_{dest} &= A_{dest}\,(A_{src}C_{src}) + C_{dest} \\ A_{dest} &= (1 - A_{src})\,A_{dest}. \end{aligned} \tag{2.2}$$

The initial value of $A_{dest}$ is 1.0. Each time a new pixel is encountered, we update the $C_{dest}$ and $A_{dest}$. Therefore, alpha value is stored with the RGB color in each iteration. Compared to back-to-front method, the advantage of this method is that we know the previous iteration's alpha value. If the alpha value is close to 0, we don't need to further process the remaining pixels, because they won't be seen any more. By using opacity thresholding, we can control how deep we want to reach for each screen position [23].

The alpha-blending algorithms mentioned above must impose the condition that the pixels at any screen position must be depth-sorted. If the front-to-back order of two objects is clear, we can render the objects either from back to front or from front to back. However, if the front-

to-back relationship of two objects is vague, we need to resolve this in the rasterization stage, checking pixel by pixel. Otherwise, the blending results are incorrect, as shown in Figure 2.3c. To address this problem, a different algorithm can be used, called order independent transparency (OIT). For each screen position, an array is allocated. All the pixels that are rasterized on the same screen position are stored in the array. After all pixels are rendered, we sort all pixels in the arrays and then render the color by the traditional algorithms mentioned above. There are ways to sort the pixels on the GPU, known as "A-buffer" and "k-buffer" [9]. The correct result is shown in Figure 2.3b. The drawback of the OIT algorithm is that we need to store a long list of pixels for one screen position in case of many overlapping pixels. Moreover, since this is an image-based algorithm, the resolution significantly affects the memory storage and computational speed.

## 2.2 General-Purpose Computation on GPU

### 2.2.1 GPU

GPUs are popular for their high parallelism and the speeds of the single-precision floating-point computations, as shown in Figure 2.4. GPUs are designed for compute-intensive and highly parallel computations. In the past two decades, GPUs have been greatly developed. Originally, GPU was designed only for graphical rendering. Figure 2.5 shows the block diagram of NVIDIA GeForce 6 series published in 2004-2005. The cores of the GPU have specific functionalities and therefore, the usage of such a GPU is limited. Modern GPU cores are more general. Each core is a general-purpose arithmetic unit that can perform the computations traditionally handled by the CPU. With its parallel architecture, modern GPUs offer great advantages compared to CPUs when the process of large data blocks is performed in parallel. These advantages also make the topic of this thesis feasible. Take NVIDIA Kepler GPU architecture as an example (Figure 2.6). It consists of several multiprocessors, an L2 cache, PCI-E bus interface and memory controllers. For each multiprocessor, it contains hundreds of CUDA cores. They work similarly to the CPU. Tasks are distributed among multiprocessors and handled concurrently. In addition to the parallel structure, discrete GPUs are always equipped with high-bandwidth independent memory on boards. Compared to CPU conventional memory, its bandwidth (30–192 GB/s) is several times faster than conventional memory bandwidth (12–30 GB/s) [4]. GPU memory access is much faster than CPU memory access.

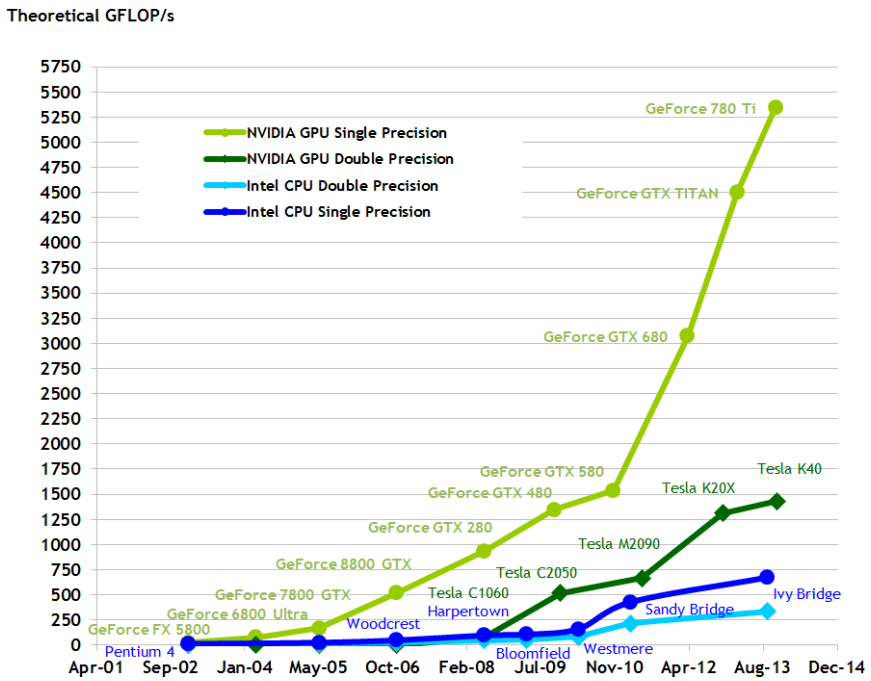



Figure 2.4 Giga floating-point operations per second (GFLOPS) comparison between CPUs and GPUs [14].

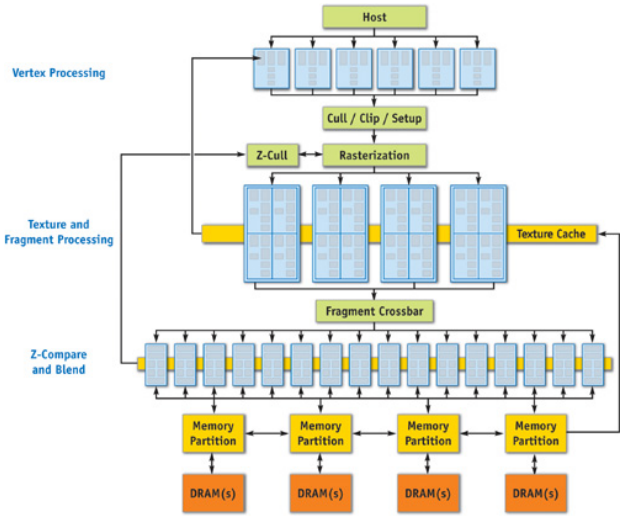



Figure 2.5 NVIDIA GeForce 6 series block diagram [17].

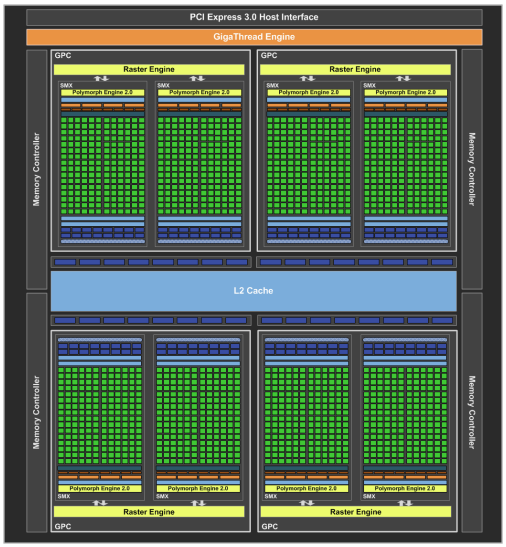



Figure 2.6 NVIDIA GeForce GTX 680 block diagram [13].

### 2.2.2 CUDA

CUDA is a parallel computing platform and programming model invented by NVIDIA. It is used for manipulating GPUs to perform general-purpose computation. CUDA enables dramatic increases in computing performance by harnessing the power of the GPU. With millions of CUDA-enabled GPUs sold to date, software developers, scientists and researchers are finding broad-ranging uses for GPU computing with CUDA [15]. In the CUDA model, GPU physical architecture is carefully abstracted at a logic level. Generally, graphics cards consist of hundreds of arithmetic cores and memory. Arithmetic cores basically load data from memory, perform computation following the instructions in the GPU programs and store the data into memory. In the CUDA model, the graphics card is described hierarchically, as illustrated in Figure 2.7. The basic arithmetic unit is called a thread. For each thread, the same CUDA kernel function is executed simultaneously. Thread has its own local memory and registers. The storage size is small, typically tens of kilobytes. Additionally, each thread cannot access the memory of the other threads. Still, the speed of the memory is the fastest in the hierarchy. Threads are grouped into blocks. Each block consists of hundreds of threads and a shared memory. Shared memory is slower and larger than local memory. Every thread in the same block can access the shared memory of the block. Threads, however, cannot access other blocks' shared memory. Blocks are grouped into a grid. A grid contains thousands of blocks. The grid can be considered as the top level of the hierarchy. Correspondingly, another type of memory is defined at this level.

There are three types of memory: constant memory, texture memory and global memory. All of them are globally accessible by all threads. The speeds of these memory types are the slowest in the hierarchy. A fast CUDA program should completely utilize the parallel structure and the hierarchy, and avoid too many global memory accesses. CUDA specifications under different compute capabilities differ from each other. The work of this thesis is based on compute capability 3.0. Accordingly, the following description is under compute capability 3.0 and may not be valid for other compute capabilities.

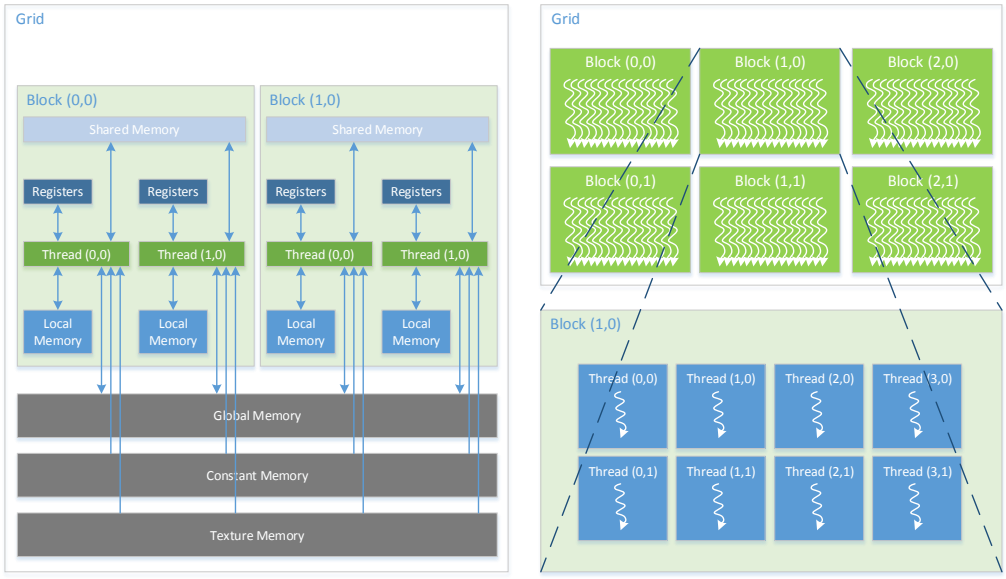



Figure 2.7 CUDA hierarchy diagram.

### CUDA Instruction Execution

All threads in a CUDA program execute the same kernel function. Suppose there are no flow control instructions. Then, at each moment of time, all threads execute the same instruction. The instructions are executed as warps by multiprocessors. Warp is an execution unit defined as a group of threads. It consists of 32 threads. GPU uses the warp as the basic execution unit. How many warps are executed per cycle in one multiprocessor is limited by the instruction type as well as the number of operations per cycle. The specifics depends on the GPU specification.

The GPU specification provides the information about the number of multiprocessors. It essentially describes how powerful a GPU is. Besides the GPU specification, the types of

instructions heavily affect the number of operations per cycle. Table 2.1 lists the number of operations per clock cycle that one multiprocessor can execute for common native arithmetic instructions. Executing 32-bit floating-point add, multiply, multiply-add instructions is typically 24 times faster than 64-bit instructions of the same kind. It is also faster than integer instructions. This means that 32-bit floating-point instruction is the best choice in a CUDA program. GFLOPS illustrated in Figure 2.4 also illustrates the same idea.

Table 2.1 Number of operations per clock cycle per multiprocessor for native arithmetic instructions [14].

| Instruction Type | Compute Capability | | |
|---|---|---|---|
| | 2.0 | 3.0 | 5.x |
| 32-bit floating-point add, multiply, multiply-add | 32 | 192 | 128 |
| 64-bit floating-point add, multiply, multiply-add | 4 | 8 | 1 |
| 32-bit floating-point reciprocal, reciprocal square root, base-2 logarithm (__log2f), base-2 exponential (exp2f), sine (__sinf), cosine (__cosf) | 4 | 32 | 32 |
| 32-bit integer add, extended-precision add, subtract, extended-precision subtract | 32 | 160 | 128 |
| 32-bit integer multiply, multiply-add, extended-precision multiply-add | 16 | 32 | Multiple instructions |
| 24-bit integer multiply (__[u]mul24) | Multiple instructions | Multiple instructions | Multiple instructions |
| 32-bit integer shift | 16 | 32 | 64 |
| compare, minimum, maximum | 32 | 160 | 64 |
| 32-bit integer bit reverse, bit field extract/insert | 16 | 32 | 64 |

Besides the type of arithmetic instructions, flow control instructions could significantly impact the effective instruction throughput. If there are different execution paths in the same warp, those paths will be serialized. If there is only one execution paths in the same warp, but the number of valid threads is less than 32 (e.g. a *if* statement without *else*), the concurrency of the warp is worse and the utilization of the GPU is lower. Therefore, flow control instructions should be used as little as possible. Since CUDA is a parallel computing platform, there are some synchronization instructions. Function __syncthreads is widely used in CUDA to

synchronize all threads in a block. However, it should be avoided as much as possible, because it costs 128 cycles in compute capability 3.0 [14].

### CUDA Memory Access

Memory access is a critical part of the CUDA program. It significantly affects the performance. As described above, CUDA has a hierarchical memory structure. The fastest but the smallest memory is the thread local memory and thread registers. A slower memory is shared memory/L1 cache that exists in each block. It is relatively larger and typically has about 64KB. Actually 64KB is program-configurable in between shared memory and L1 cache. The slowest memory is device global memory that contains constant memory, texture memory and global memory. Constant memory is the smallest memory at that level, but it is the fastest one if it is cached. Constant memory is read-only during the GPU computation. Typically, we store small parameters into the constant memory. They are frequently used during execution and will be cached. On the other hand, texture memory is used for storing the texture and cached in the read-only cache of each multiprocessor. This operation is efficient when users want to sample the texture, because GPU provides various kinds of memory fetching and data filtering functionalities. Furthermore, the cache of texture memory provides a higher hit rate if the data has good spatial locality [7]. Global memory is a basic type of memory. It is similar to the usage of main memory. Users can store scalars, vectors and arrays into global memory. Unlike texture memory, it does not have many ways to sample and fetch data. Still, global memory is also cached in one unique L2 cache that can be accessed by all multiprocessors, as shown in Figure 2.6. From the hierarchy, we can see that we should carefully design the data storage to improve the performance. Too many accesses to global memory should be avoided. If the data is frequently used, we should fetch them from global memory to a shared memory or even lower levels.

The global memory data is align-accessed by the GPU. The size of a GPU memory transaction is typically 32, 64 or 128 bytes. Memory accesses of the warp are coalesced into one or more of these transactions. The more bytes the warp needs, the more transactions are needed. Additionally, if the accessed data address is not aligned, extra transactions are needed and thereby unused data is accessed. This is shown in Figure 2.8. If data is sparsely stored in memory, it will cause a lot of unused data accesses. Therefore, if the address of data that GPU accessed is aligned and the accesses of a warp are sequential, the memory access time will be optimized. The L2 cache size is usually smaller than 1.5MB, so the size of L2 cache is much smaller than the sum of shared memory in all thread blocks [25]. Therefore, it is better to

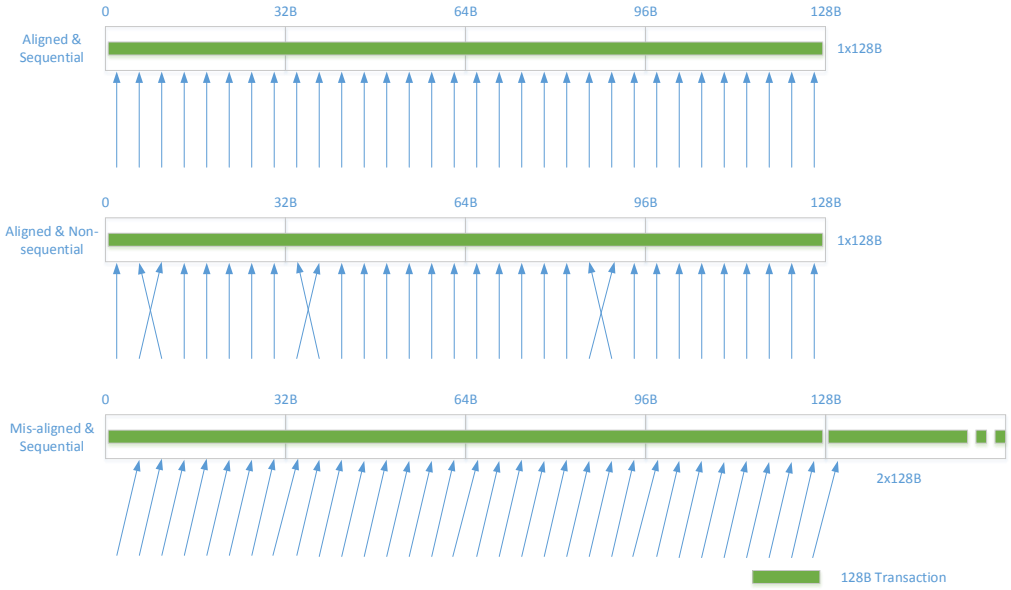



Figure 2.8 Examples of global memory accesses by a warp. Each thread accesses one 4-byte word. Corresponding memory transactions are displayed in green.

cache data into shared memory. If a data block is frequently used, it should be loaded to shared memory of the corresponding thread block first.

Shared memory in each block is composed of banks. If two threads in the same warp access the same bank, but not the same 64-bit word, bank conflicts will occur. Figure 2.9 illustrates the concept of a bank conflict. If a bank conflict happens, memory transactions will be serialized, costing more time to access the data in the shared memory. The data layout in shared memory can greatly impact the memory access time.

### CUDA Computation

In order to use CUDA parallel architecture, the matrix multiplication is decomposed as follows:

$$C = AB = \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_n \end{bmatrix} \begin{bmatrix} B_1 & B_2 & \dots & B_n \end{bmatrix} = \begin{bmatrix} A_1B_1 & A_1B_2 & \dots & A_1B_n \\ A_2B_1 & A_2B_2 & \dots & A_2B_n \\ \vdots & \vdots & \ddots & \vdots \\ A_nB_1 & A_nB_2 & \dots & A_nB_n \end{bmatrix} \tag{2.3}$$

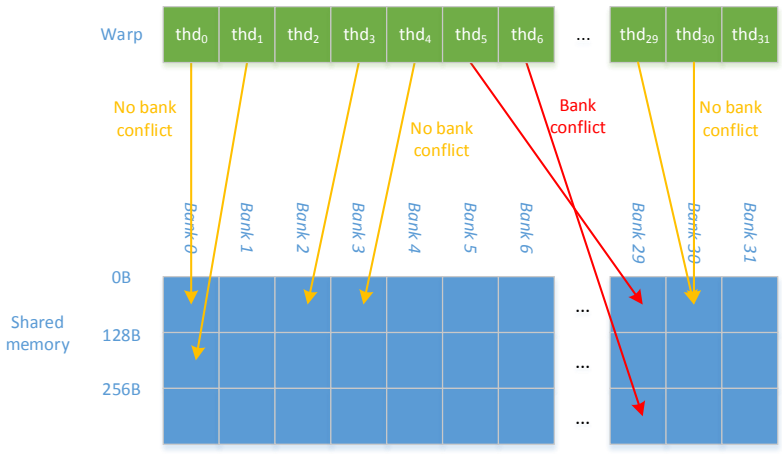



Figure 2.9 Shared memory structure and bank conflicts.

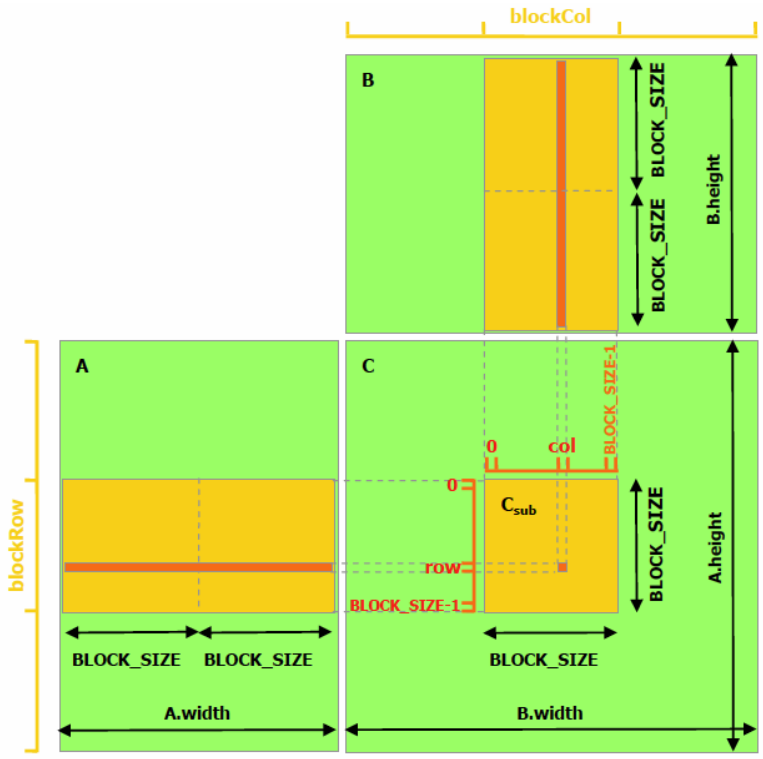



Figure 2.10 Matrix multiplication using CUDA [14].

Each sub-multiplication can be addressed separately. If the data required for each sub-multiplication is less than the shared memory of each block, one or more sub-multiplications can be calculated inside one thread block. Otherwise, sub-multiplications can be further decomposed, as illustrated in Figure 2.10. The algorithm is described in the NVIDIA CUDA programming guide [14]. Each time we read a pair of sub-matrices from $A_i$ and $B_j$, and store them to shared memory. Each thread takes one row and one column from the sub-matrices, calculates their dot product, records the result and waits for computing another pair of sub-matrices. After all sub-matrices of $A_i$ and $B_j$ are processed, the corresponding sub-matrix in $C$ is computed. By using shared memory, we avoid repetitive loading from global memory, which saves a lot of time.

Matrix-vector multiplication can be considered as one kind of matrix multiplication. Therefore, we can use the same idea to compute it. The algorithm is a general method for matrix multiplication, so it may not be the best algorithm for all cases. Different approaches exist for specific cases. Bell and Garland provided an efficient method for sparse matrix-vector multiplication [2]. They provide several CUDA kernel functions that are suitable for different representations of sparse matrices. $Uq$ computation is a specific kind of matrix-vector multiplication. The $U$ matrix has a very large number of rows but few columns. Additionally, $q$ is typically a small vector, with dimension less than 32. In the following chapter, we describe how we perform the $Uq$ computation.

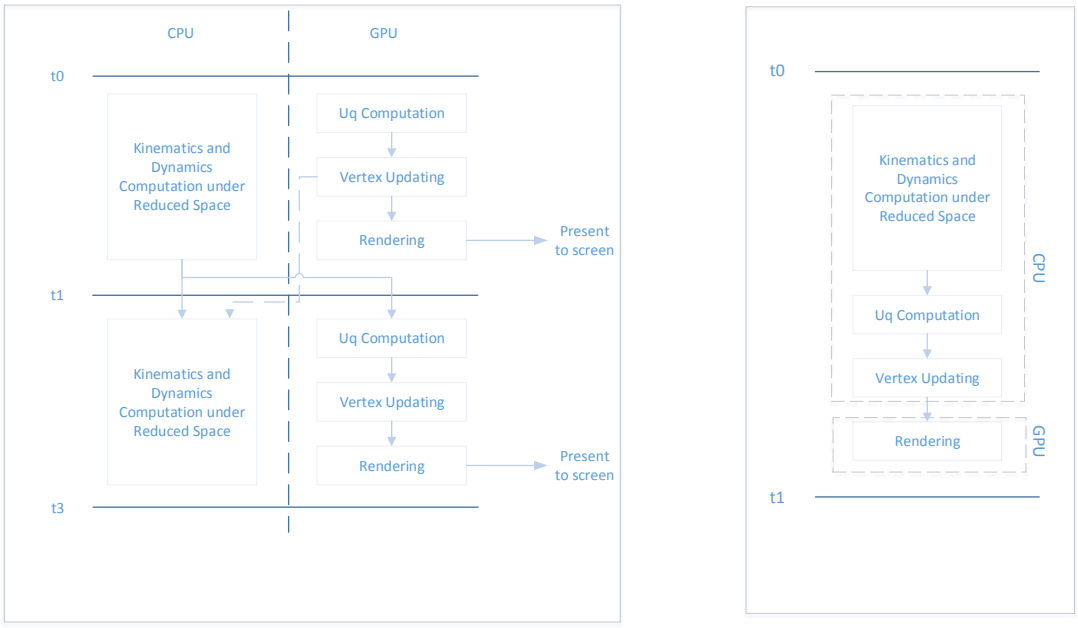
# Chapter 3

# Simulation System

## 3.1 System Overview

Based on the GPU structure, we propose a novel architecture and CUDA approach to improving the performance of the simulation system that has one or more deformable domains simulated by FEM and model reduction. The system consists of a physically based simulation and rendering. For physically based simulation, it consists of the computation of kinematics and dynamics in the reduced space, the $u = Uq$ computation and vertex updating. The overall workflow of our system is shown in Figure 3.1a. Compared to the workflow of previous systems, our system enables the CPU and GPU to work at the same time, and thereby decreases the time cost of each time step. Moreover, our system not only reduces the CPU load, but also reduces the amount of communication between the CPU and GPU.

When our system is running, CPU only computes the kinematics and dynamics based on the data from the previous time step. At the end of the current time step, new coordinates, velocities and accelerations in the reduced space are computed. Based on the reduced coordinates $q$ and rigid transformations computed by previous time step, GPU calculates new $u$, vertex global positions, and eventually renders the mesh on the screen. Therefore, in the same time step, the tasks on CPU and GPU are independent. Compared to our system, previous architectures are much simpler, as shown in Figure 3.1b. However, previous architectures do not fully utilize the GPU. When CPU is working, GPU may be in an idle state, and vice versa.




(a) Our system. (b) Previous system.

Figure 3.1 The workflows of two time steps in different systems.

## 3.2 CUDA Computation

### 3.2.1 Single Domain Uq Computation

If there is only one deformable domain in the system, there is only one $U$ matrix and one $q$ vector to be multiplied in each time step. For all vertices, the displacement $u$ can be computed by $u = Uq$. Due to the properties of GPU, the $U$ matrix and $q$ vector storage layouts are critical for the algorithm performance. $U$ is a modal matrix with $3n$ rows and $r$ columns. For each vertex $i$, the displacement $u_i$ is computed by

$$u_i = \begin{pmatrix} \sum_j^r U_{3i,j} q_j \\ \sum_j^r (U_{3i+1,j} q_j) \\ \sum_f^r (U_{3i+2,j} q_j) \end{pmatrix} \tag{3.1}$$

where $U_{i,j}$ is the element in row $i$ and column $j$. Denote $u_i$ by $(u_{i1}, u_{i2}, u_{i3})^T$. From Equation 3.1, $u_{ij}$ is computed as a sum of a series of multiplications. The multiplications are independent of each other and can be executed in parallel. We distribute each multiplication into dedicated

GPU threads. In each thread, the program fetches a part of the $U$ matrix and a part of the $q$ vector from global memory and multiplies them together. Since the size of $q$ is usually less than 32, in the computation of each $u_{ij}$, the summation of the results of the multiplications can be computed within 32 threads, i.e., one warp. Finally, the result of $u_{ij}$ is stored into the global memory.

For each GPU block that has $w$ columns and $h$ rows, we allocate $T$ threads. Denote a GPU thread by $thd_{ij}$, where $0 \leq i < h$, $0 \leq j < w$ and $h \times w = T$. We set $w$ to the number of the threads in a warp. The reason for this is that a warp is the basic execution unit. This can also avoid bank conflicts. Take $r = 30$ as an example. With the basic idea of computing $Uq$ above, there is only one active thread per warp when the final summation is computed. As a result, the GPU may be under-utilized and the performance decreases. To avoid this, we put four multiplications into one thread. Then the multiplications are grouped into groups of four. There are two reasons for grouping. First, we can utilize four times more active threads per warp during the final summation. Moreover, in GPUs, the vector multiplication is more efficient than the scalar multiplication. GPUs can calculate the four multiplications as one operation. After vector multiplication, the local summation of the four elements in the result vector is performed by each thread, improving concurrency. The summation result is stored into shared memory for later usage. Denote the $k$-th group of $q$ by $q^k$, the $k$-th group of $U_{3i+j}$ by $U^k_{3i+j}$ and the $k$-th group of the sub-summation of $u_{ij}$ by $u^k_{ij}$. Then, Equation 3.1 is rewritten as follows:

$$\begin{aligned} u_{ij} &= \sum_k u^k_{ij} \\ u^k_{ij} &= q^k \cdot U^k_{3i+j} \\ q^k &= (q_{4k}, q_{4k+1}, q_{4k+2}, q_{4k+3})^T \\ U^k_i &= \left(U_{i,4k}, U_{i,4k+1}, U_{i,4k+2}, U_{i,4k+3}\right)^T \end{aligned} \tag{3.2}$$

Shared memory data storage layout is critical for the time cost of the $Uq$ computation. Before describing the shared memory layout, we need to discuss how floating point values are represented and managed by GPUs. We use 32-bit single-precision floating-point computation instead of 64-bit double-precision which are more commonly used by CPU simulation programs. There are three reasons for this. First of all, the 32-bit single-float operations are the fastest in its class among all other data types in the GPU. Additionally, the fetch and store for 4-byte data type are faster than for the 8-byte data type. Moreover, the final displacement $u_i$ is used only for rendering and collision detection. They do not need high precision as in physical based

simulation. Especially for rendering, OpenGL only accepts single-precision values in most cases anyway. In the following sections, we use 32-bit single-precision.

Since the multiplications are grouped by four, each warp can compute more than one $u_{ij}$ at the same time. Explicitly, we can compute more than three $u_{ij}$ at the same time. Denote the size of $q$ vector by $r$. After $q$ is grouped by four, the number of threads that are used for computing one $u_{ij}$ is $n_{vr} = \lceil \frac{r}{4} \rceil$. Then the number of $u_{ij}$ that can be computed at the same time in a warp is $n_g = \left\lfloor \frac{32}{n_{vr}} \right\rfloor$. Since there is only one deformable domain, there is only one $q$ vector. This also means that the computation of each $u_{ij}$ uses the same $q$ in one time step. To avoid loading $q$ multiple times, each block loads $q$ from global memory to shared memory once at the beginning of the computation. We only need to load one $q$ into shared memory per block. Even if two threads in a warp access the same 32-bit word, there won't be any bank conflicts. Denote the base address of shared memory for storing $q$ by $addr_q$. To avoid a bank conflict, $q_i$ will be stored in

$$addr_{q_i} = addr_q + \left( (i \mod 4) \times 32 + \left\lfloor \frac{i}{4} \right\rfloor \right) \times 4. \tag{3.3}$$

The $q$ storage layout is shown in Figure 3.2. Note that even though $q$ is stored sequentially in the shared memory, the bank conflict won't happen if $r$ is less or equal to 32. Still, by our method there won't be any bank conflict even if $q$ is larger than 32. To load the $q$ into shared memory, $thd_{ij}$ will load $q_{4j+i}$ into the corresponding memory address. After the loading process, each thread $thd_{ik}$ fetches $q^k$ and the corresponding $U_i^k$. The value $q^k$ can be read from the shared memory, because $q$ has already been loaded from the global memory to the shared memory as described above. On the other hand, since $U_i^k$ is used only once per $Uq$ computation, we do not need to load it into the shared memory in advance. Instead, $U_i^k$ is directly loaded from the global memory. To enhance the loading performance, $U_i$ is stored sequentially in the GPU memory. Since one warp can compute several $u_{ij}$, we pack a series of continuous $U_i$ into one larger vector. The head address of it is 128-byte aligned, as shown in Figure 3.3. Denote the base shared memory address for storing result $u_{ij}^k$ by $addr_{usub}$. In thread $thd_{ik}$, $u_{ij}^k$ is computed and stored in

$$addr_{u_{ij}^k} = addr_{usub} + (i \times 32 + k) \times 4. \tag{3.4}$$

Figure 3.2 shows the memory layout of $addr_{u_{ij}^k}$. After all $u_{ij}^k$ are computed for $u_{ij}$, $u_{ij}$ are finally computed by selected threads in a warp. The yellow threads in Figure 3.2 calculate the summation. We can see that the thread $thd_{ik}$ with $k \mod n_{vr} = 0$ computes the summation. As described before, $r$ is usually less equal to 32 in most situations. In the case that $k$ is larger than $n_{vr}$, $thd_{ik}$ will compute another $u_{i'j'}$, by using $q^{k \mod n_{vr}}$ and $U_{i'}^{k \mod n_{vr}}$.

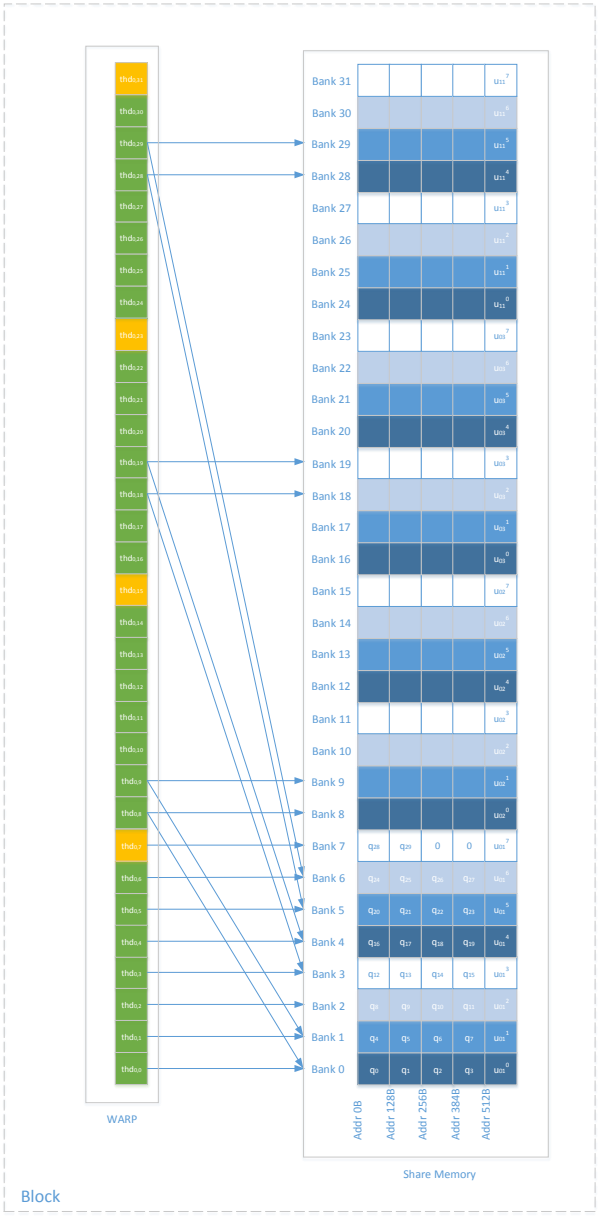



Figure 3.2 Shared memory layout and thread access overview. In this figure, $r = 30$. A warp can compute four $u_{ij}$ at the same time. The banks the arrows point to are accessed by the corresponding threads. Threads highlighted in yellow compute the summations.

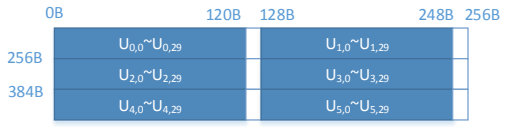



Figure 3.3 $U$ matrix layout in global memory.

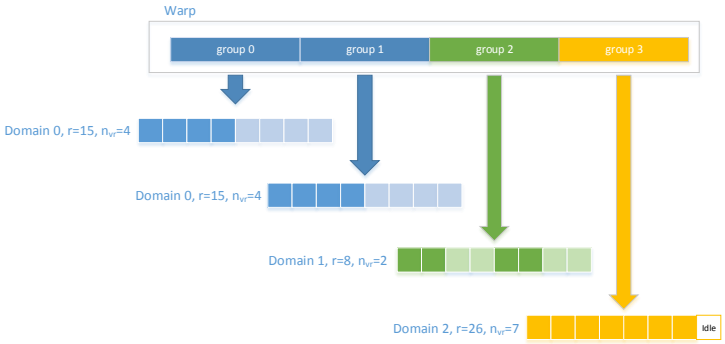



Figure 3.4 To address domains with different dimensions $r$, we use the group concept.

For convenience of final vertex world position computation, $u_i$ is stored as a 4-element vector rather than 3-element vector and sequentially stored in global memory. Additionally, $u_i$ is stored in the order of $i$. To enable CPU access to $u$ stored in the GPU global memory, we store $u$ by page-locked host memory. Main memory for storing $u$ is mapped to GPU memory, and can be accessed by CPU and GPU at the same time. This also avoid too many memory copy operations and thereby decreases the data communication time. The details will be described later.

### 3.2.2 Multi-domain Uq Computation

With multiple domains or objects, there are several differences compared to single-domain simulation. First, there is more than one independent deformable part in the system. Moreover, different domains may have different $U$ matrices and $q$ vectors. One does not just have different data in $U$ and $q$; $U$ and $q$ may have different dimensions for different domains. We give a GPU parallel algorithm to accelerate multiple $Uq$ computation in this setting. We use an idea similar to single $Uq$ computation. However, the single-domain computation idea cannot be completely adopted, because $U$ matrices and $q$ vectors are different in each domain.

Consider a relatively simple situation where the dimension $r$ of each domain is the same. This situation is easy to address and can be solved by adapting the algorithm of the previous section. We need to load more than one $q$ for each block rather than only one $q$ because vectors $q$ are different for the different domains. For $U$ matrices, due to the same number of columns, we can combine them together into one large $U$ matrix. The work for each thread is the same as before, except the storing addresses of $U$ and $q$. In fact, $r$ of all domains may not be the same. Accordingly, we propose an approach to "uniformize" the different values of $r$. To do this, we define a new concept called "group". Threads of a warp are divided into a number of groups. A group computes several $u_{ij}$ of the same domain, but not of different domains. Different groups can compute $u_{ij}$ in the same or different domains. A group cannot be processed by two warps at the same time. Denote the group width by $w_g$. Group width means the number of threads in a group. The number of groups in a warp is $n_g = \left\lfloor \frac{32}{w_g} \right\rfloor$. For a domain whose dimension is $r$, we first group $U_i$ by four as in the single domain algorithm. The $U$ matrix now can be considered as a $3n \times n_{vr}$ matrix and each element inside the matrix is a four-element vector. We denote it by $U'$. Now the number of $u_{ij}$ that a group of threads can address concurrently becomes $n_{gg} = \left\lfloor \frac{w_g}{n_{vr}} \right\rfloor$. To accommodate the group width, we reshape the $U'$ matrix to make the number of columns equal to $w_g$. New matrix will have $w_g$ columns and $n'$ rows. $n'$ is computed by $n' = \left\lceil \frac{3n}{n_{gg}} \right\rceil$. Define $U'^k$ as a sub-matrix of the $U'$ matrix. $U'^k$ has $n_{vr}$ columns and $n'$ rows. $U'$ can be represented as

$$U' = \begin{bmatrix} U'^1 \\ U'^2 \\ \dots \\ U'^{n_{gg}} \end{bmatrix} \tag{3.5}$$

Then, $U'$ is reshaped to

$$U'' = \begin{bmatrix} U'^1 & U'^2 & \dots U'^{n_{gg}} \end{bmatrix} \tag{3.6}$$

Conceptually, $q$ is reshaped by the same idea as $U$. The $q$ vector is first grouped by four components to $q'$. Then $q'$ is duplicated $n_{gg}$ times. Denote it by $q''$ The data actually used by one group is one row of the $U''$ matrix and $q''$ vector. When threads in a group compute $Uq$, they perform the same calculation as in a single-domain $Uq$ computation. Note that a group will also compute one or more $u_{ij}$ rather than only one at the same time. This is shown in Figure 3.4. In other words, when the final summation is calculated, there will be one or more active threads inside one group.

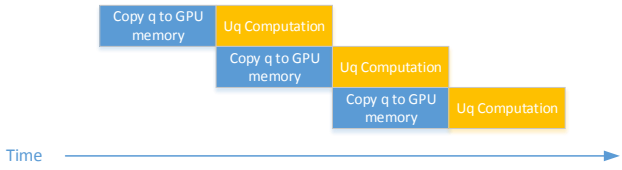



Figure 3.5 Kernel function execution.

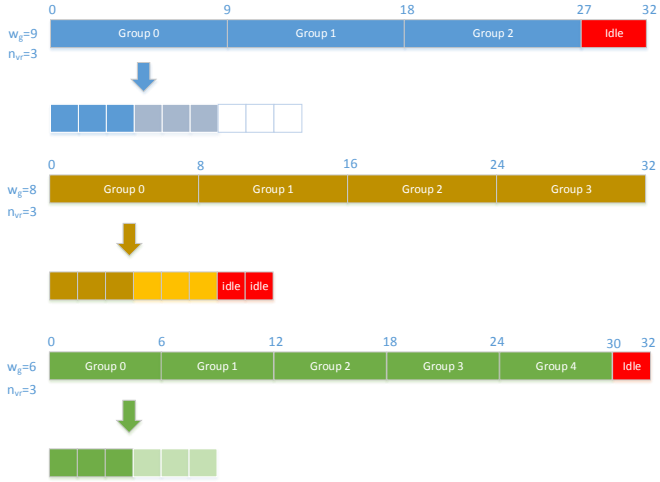



Figure 3.6 Thread states in a warp, under varying $W_g$.

Since a block has many threads, it will contain a lot of groups. As a result, a block may process one or more domains. Therefore, a block needs to load one or more domains' $q$. Threads in a warp may use a different $q$. This time, we don't cache $q$ in shared memory. Instead, we directly load them to the threads. This is because $q$ is not reused as much as it in the single-domain execution in each block. Additionally, one or more vectors $q$ needs to be loaded into a block, so synchronizing threads costs more time than before. Therefore, when $Uq$ is computed, threads need to know which part of $q$ should be loaded, where $u_{ij}$ should be stored, and the current domain's dimension $r$. Accordingly, an extra meta-data vector is required for each thread to assist with the $Uq$ computation. For each thread, we use a 4-element vector to store the required data. It contains the address of $u_{ij}$ for global memory storage, the address of $q$ in shared memory and $n_{vr}$. The reason we use 4-element vectors to store three elements is that this improves memory access performance.

When there is more than one domain, we can use the grouping method to "uniformize" all domains. Group width $w_g$ is an important parameter that influences the performance of the algorithm and the utilization of the GPU. Suppose any domain's dimension $r$ is less or equal to 32. Group width $w_g$ must be greater than or equal to 8 in order to compute one $u_{ij}$ of the domains whose maximal dimension is 32. Admittedly, we can set $w_g$ to a number over 8. However, there will be more idle threads in a warp because a warp can only contain three such groups. Undefined threads that are outside the groups are not used during the computation. Moreover, there will be more idle threads in a group. Figure 3.6 illustrates these problems. To utilize threads efficiently, we provide different sizes of groups, in order to be compatible with various dimensions $r$. We found that there are a total of eight different $n_{vr}$, i.e., from 1 to 8. For $n_{vr}$ that equals to 1, 2, 4 and 8, we can use $w_g = 8$. For $n_{vr}$ that equals to 3 and 6, we can use $w_g = 6$. For $n_{vr}$ that equals to 5 and 7, we can choose to address them separately and there will be 4 different groups. This strategy can guarantee that there is no idle thread for each group size. Still, there are four kernel executions in each time step. CUDA streaming technology provides us with the functionality to parallelize kernel function execution and data transmission, as shown in Figure 3.5. However, as the number of kernel function executions increases, the time cost becomes more expensive. The technology cannot guarantee absolute parallelism of the kernel execution and data transmission. Additionally, for each kernel execution, CPU and GPU take a long time to prepare for it. Consequently, more kernel function calls will slow down the computation. According to the experiments, we found that using 2 different sizes of groups has the best performance as well as utilization rate of GPU threads. In this situation, domains whose $n_{vr}$ is equal to 5 is merged into group with $w_g = 6$. Domains whose $n_{vr}$ is equal to 7 is merged into the group with $w_g = 8$.

For multi-domain $Uq$ computation, $u$ is stored in the same way as in the single domain situation. Vector $u$ of each domain is stored sequentially in the memory.

### 3.2.3 Multi-domain Vertex Updating

After the displacement $u$ is computed for each domain, new vertex position is computed, by taking into account the domain's global position and orientation. We call this process "vertex updating". The result of the vertex updating are the global vertex positions which can then be used to render the mesh.

Triangle meshes are commonly used to represent objects in computer graphics. A triangle mesh is given by a list of triangles. The triangle are represented by their vertices. Meshes can

be rendered either as an array of triangle vertices or as an array of vertices with an array of vertex indices. In the first approach, the number of vertices in the vertex array is a multiple of three, because each three vertices in the array represent a triangle. In the second approach, the triangles are denoted by an index array. Each three indices in the array account for a triangle. The index is the vertex index in the vertex array. The second approach is more compact if there are many duplicated vertices. In our system, we choose the first approach. Therefore, each mesh triangle is treated and updated separately. Denote the displacement of domain $i$ by $\mathbf{u_i}$. $\mathbf{u_i}$ has

$$\begin{aligned} \mathbf{u_i} &= \begin{bmatrix} u_1^i \\ u_2^i \\ \dots \\ u_n^i \end{bmatrix} \\ u_j^i &= \left(u_{j1}^i, u_{j2}^i, u_{j3}^i\right)^T \end{aligned} \tag{3.7}$$

When the system is initialized, we first unfold the mesh into separate triangles for domain $i$. Denote the position of a triangle vertex by $x_j^i$, normal by $n_j^i$ and tangent by $t_j^i$. Denote the rest position of a triangle vertex by $x_{0j}^i$, rest normal by $n_{0j}^i$ and rest tangent by $t_{0j}^i$. Also, denote the rigid rotation by $R_i$ and translation by $p_i$. A vertex is updated by

$$\begin{aligned} x_j^i &= R_i\left(x_{j0}^i + u_k^i\right) + p_i \\ n_j^i &= R_i n_{j0}^i \\ t_j^i &= R_i t_{j0}^i \end{aligned} \tag{3.8}$$

We do not recompute the normal and tangent after the object is deformed, because doing so would add additional cost to the CPU and GPU. Often, for reasonable deformations, it may be difficult to recognize the error of normals and tangents. Consequently, normals and tangents are only rigidly transformed.

The rotation matrix $R_i$ is a $3 \times 3$ matrix. We can use a $4 \times 4$ homogeneous matrix packing both rotation and translation. The equation of updating $x_j^i$ becomes

$$\begin{aligned} x_j^i &= T_i\left(x_{j0}^i + u_k^i\right) \\ T_i &= P_i R_i \end{aligned} \tag{3.9}$$

where $P_i$ is a $4 \times 4$ translation matrix. Accordingly, there will only be three matrix-vector multiplications and one vector addition for each vertex. Using a similar idea as in $Uq$ computation, we load commonly used data to shared memory. Transformation matrix $T_i$ and rest configurations are loaded into shared memory. Positions, normals and tangents are represented as 4-element vectors, since the transformation matrix is a $4 \times 4$ matrix. Additionally, the warp size is a multiple of four. As a result, a warp can calculate eight vertices at the same time. Each thread inside a warp only computes one element of a vertex position, normal or tangent vector. Each four threads compute one vertex.

For each three vertices, the same transformation matrix is used. We assume that each domain has at least has three triangles, so each warp needs at most two matrices at the same time. The transformation matrix is stored row major in the shared memory sequentially, so there is not any overload for reading them. Additionally, there is not any bank conflicts when matrices are read. In addition to the matrix, the rest position, normal and tangent of each vertex are loaded into shared memory before the computation, because they are frequently used by matrix multiplication. They are separately stored in shared memory because each thread in a warp can exactly read data from the corresponding bank without any bank conflicts. Finally, the shared memory layout is shown in Figure 3.7. Every time the computation starts, the program loads the data to shared memory first. For $thd_{ij}$, it loads the $(j \mod 16)$-th element in $\left(2i + \left\lfloor \frac{j}{2} \right\rfloor\right)$th transformation matrix of the block. Since each block may load different transformation matrices, what a thread should load is precomputed when the system is initialized. After matrix is loaded into shared memory, we load the position, normal and tangent vectors into shared memory. For $thd_{ij}$ in block $k$, it loads the $(j \mod 4)$-th element of the $\left(n_{vb}k + 8i + \left\lfloor \frac{j}{4} \right\rfloor\right)$-th vertex, where $n_{vb}$ is the number of vertices each block can compute. It is the same for all blocks. When the positions are fetched from global memory, we add the displacements to them before storing them to the shared memory. Before the matrix-vector multiplication, we need to synchronize all threads inside the block to guarantee that everything the computation needs is successfully loaded into the shared memory. After the data is loaded into shared memory, each thread calculates one element of a vertex and stores the result back to global memory directly.

### 3.2.4 Data communication

Figure 3.8 shows data communication between and within devices. Packed $U$ matrices and vertex rest positions, normals and tangents are stored into GPU global memory when the system is initialized.

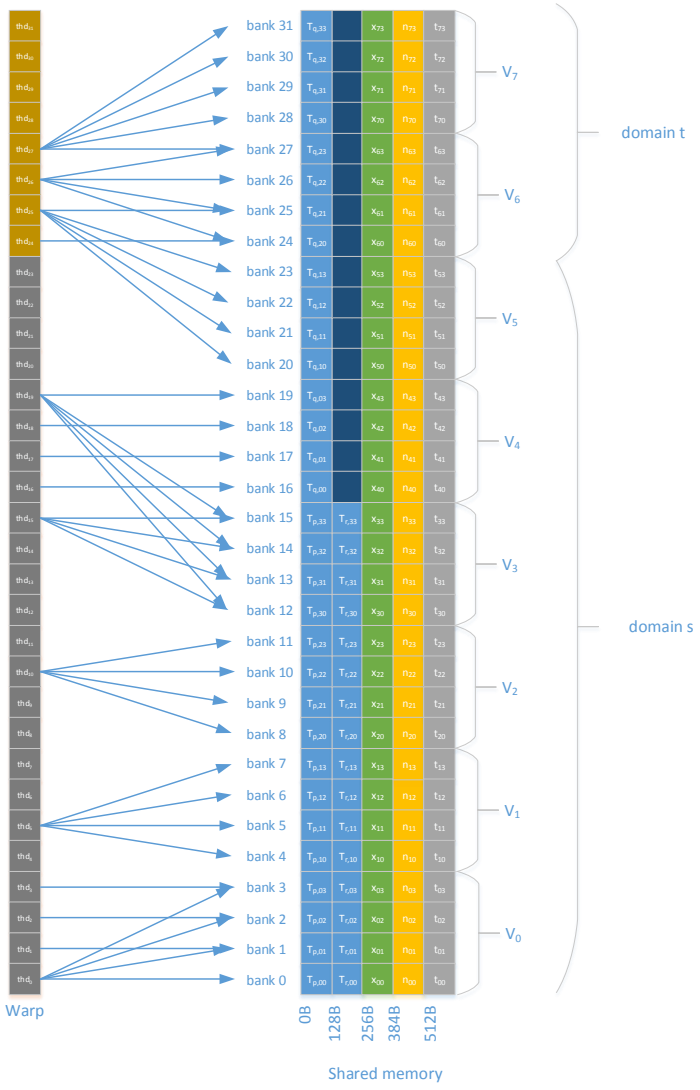



Figure 3.7 The layout of the shared memory and the bank accesses of threads for vertex updating. Different colors of threads mean that they address different domains. Areas of different colors of the shared memory mean that they store different kinds of data. Blue represents transformation matrices. Green represents vertex positions. Yellow represents normals. Gray represents tangents.

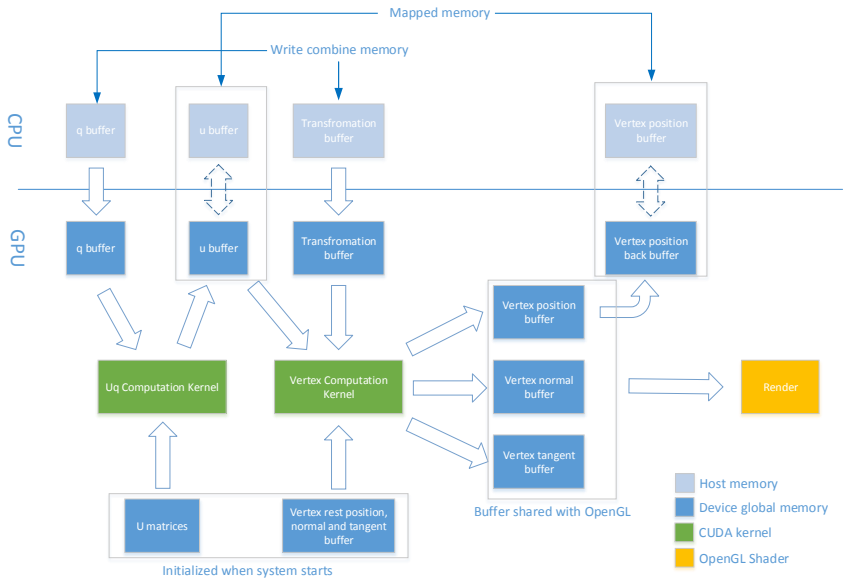



Figure 3.8 Data communications between buffers and devices.

In each time step, CPU computes new $q$, packs it for CUDA computation and flushes it to the host $q$ buffer. CPU only writes to the $q$ buffer so the host $q$ buffer can be a write-combine memory [8]. It is page-locked host memory, which accelerates memory transactions [14]. Furthermore, it is write-combining which greatly improves the write performance. After the CPU $q$ buffer is updated, the data will be transferred to GPU memory. Since there is more than one CUDA $Uq$ computation kernel, kernel executions are streaming. To improve the concurrency, we use asynchronous memory copy operations to copy data from the $q$ buffer in the CPU to the $q$ buffer in the GPU. Consequently, data copying and kernel execution can be parallelized. After the $Uq$ computation, result $u$ is placed into $u$ buffer in the GPU. Vector $u$ is stored in a mapped memory which is one of the page-locked host memories. GPU memory is mapped into CPU main memory. The advantage here is that mapped memory can reduce many unnecessary copy operations between the CPU and GPU, accelerating the performance. Typically, $u$ is not used by CPU because vertex updating is performed by the GPU. Therefore, copying between the CPU and GPU is avoided.

After the $Uq$ computation, vertex updating begins. First, we acquire $u$ from the $u$ buffer in the GPU and the transformation matrices from the transformation buffer in the GPU. The transformation buffer is also write-combined memory. Additionally, the transformation buffer in the CPU is updated after asynchronous $q$ buffer copying starts. This ensures that transferring

the transformation matrix data proceeds in parallel with the $Uq$ computation. Until the $u$ buffer and transformation buffer in the GPU are updated, we do not start the vertex updating. The output of vertex updating is memory-shared by CUDA and OpenGL. Before writing data to memory, CUDA needs to set flags in order to gain control over the buffer. The output memory is unmapped until vertex updating is completed. After that, the rendering can start. During the vertex updating, CPU cannot retrieve data from the GPU vertex position buffer because the operation has not been completed. To make vertex position viable during the computation, we provide double buffers for vertex position buffer. During the computation, the back buffer can be accessed by CPU. After vertex updating is done, the back buffer is updated. Since all buffer are in the same GPU memory, the memory copy speed is fast. We also use mapped memory for the back buffer to improve the CPU to GPU transfer speed. After vertex updating is done, the rendering starts. Since CUDA and OpenGL both utilize the same GPU resources, the parallel work is not significant. Therefore, in every timestep, CUDA computation is performed before rendering begins.

Buffers in the GPU memory are not only write-combined or mapped, but are also portable memory. Therefore, all buffers can be accessed by multiple threads. With this property, a simulation program that runs on another thread can read and write the mapped memory.

## 3.3 Rendering

After the vertices are updated, they are rendered to the screen using the OpenGL core profile [16]. The overall pipeline is shown in Figure 3.9. The rendering process is divided into three procedures. The first step is to generate a depth map for later shadow mapping. The second step is to generate G-buffer (position and normal) and vertex index buffer as well as to shade the objects. The last step is to post-process the color image and display it on the screen. The goal of the rendering system is not only to increase the overall system frame rate, but also to improve rendering quality. Since our system mainly focuses on simulation, we do not introduce a lot of workload for rendering. The three steps are introduced in the following text.

### 3.3.1 Shadow Maps

Plants are our main simulating objects. Since they are complex meshes, shadow maps offer advantages over shadow volumes. Nonetheless, traditional z-buffer shadow mapping has the obvious aliasing problem if the depth map is not filtered properly. To render realistic shadows

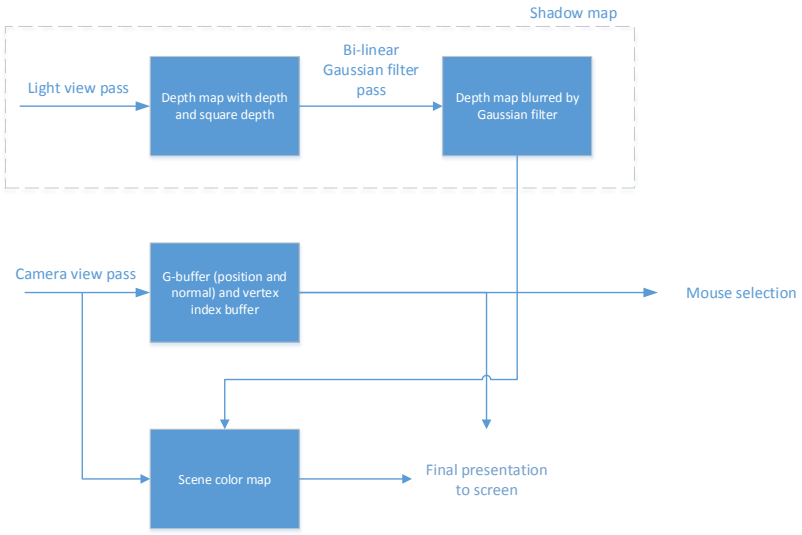



Figure 3.9 Overview of the rendering pipeline.

with a low cost, we choose to use variance shadow maps (VSM) [6]. The basic idea originates from the following inequality and equations:

$$\begin{aligned} P(x \leq t) \leq p_{max}(t) &\equiv \frac{\sigma^2}{\sigma^2 + (t-\mu)^2} \\ \mu &= E(x) = M_1 \\ \sigma^2 &= E\left(x^2\right) - E(x)^2 = M_2 - M_1, \end{aligned} \tag{3.10}$$

where $M_1$ is the mean of the depth and $M_2$ is the mean of the squared depth. Quantity $P(x \leq t)$ gives the percentage of the surface that is in shadow. We use $p_{max}(t)$ to approximate this value, because $p_{max}(t)$ is a good approximation of $P(x \leq t)$. Quantity $p_{max}(t)$ can be computed by Equation 3.10. To implement this algorithm, we first need to compute $M_1$ and $M_2$. Instead of generating a traditional shadow map via a z-buffer from the light direction, we render the depth value to the color framebuffer. The depth value is stored in one channel of each pixel. In addition to the depth value, we also compute the square of the depth value and store it into another channel of the pixel. This procedure is the "light view pass" in Figure 3.9. Another issue which should be considered is that there may be transparent objects in the scene. The transparent property is specified in object's material, usually via the alpha value of its diffuse color (may also be texture color). In real life, the shadows of transparent objects are lighter

than the shadows of opaque objects. In the VSM algorithm, there is not a way to discriminate whether a shadow is cast by a transparent object or by an opaque object. Therefore, we decide that an object is considered as opaque if the transparency of it is higher than a pre-defined threshold $t_{aplha}$. Otherwise, it is considered as a totally transparent object that doesn't cast any shadows.

A general depth map is generated by rendering the scene from the light viewpoint. We assume that the sun is the default light source. A virtual camera is placed at some high position and aimed in the direction opposite to light direction. The projection matrix used is the orthographic matrix for the directional light. To further increase the shadow map quality, we try to remove areas where no shadow is cast. To do this, we first compute an axis-aligned bounding box (AABB) for the objects in their local space. If the objects are rigidly transformed into world space, we only need to rigidly transform the bounding box accordingly. This means that we do not need to re-compute the bounding box in each time step. Since the object is deformable, we scale the bounding box by some factor to prevent the object from escaping the bounding box in most situation. This works well for plants. After the bounding box in the world space is calculated, we transform the bounding box to light view space. To do this, we need to compute the light view coordinate system in the world space. Denote the center of world-space bounding box by $O_{bb}$. Let $v_l$ be the sun direction vector. Denote the maximum distance between any point within the world space bounding box and $O_{bb}$ by $d_{bb}$. The origin $O_{lv}$ of the light view coordinate system expressed in the world space can be calculated by $O_{lv} = O_{bb} + d_{bb}v_l$. We use a right-hand coordinate system, so the z-axis of the light view space is $v_l$. The y-axis points up in the world. Then, the x-axis can be computed using a cross-product. After we compute the transformation matrix to the light view space, we can transform the world space bounding box into the light view space. Based on the bounding box in the light view space, we re-compute another AABB on it. This AABB is symmetric around xy face, xz face and yz face in the light view space. To ensure that the shadow is successfully cast on the ground, we let four facets of the AABB intersect with the ground plane. An orthographic matrix is computed to make the AABB exactly project on the screen. With this algorithm, the depth map is dynamically following the objects and they are always center in the depth map. Moreover, the object occupy most of the area of the depth map.

If the generated shadow map is directly used when the objects are shaded, the shadow boundary would not be smooth enough. To further improve the shadow quality, we apply Gaussian blur on the depth map before it is used. Traditional Gaussian blur is done in one drawing call. However, this costs a lot of GPU resources because one needs to sample tens

of two-dimensional neighboring pixels in order to reach a considerable quality. Instead, we use two-pass Gaussian blur to improve the performance. In the first pass, one-dimensional Gaussian blur is applied in the X direction. In the second pass, it is applied in the Y direction. The equations are as follows:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{\frac{x^2+y^2}{2\sigma^2}} = \left(\frac{1}{\sqrt{2\pi}\sigma} e^{\frac{x^2}{2\sigma^2}}\right)\left(\frac{1}{\sqrt{2\pi}\sigma} e^{\frac{y^2}{2\sigma^2}}\right) \tag{3.11}$$

This equation will produce similar quality as traditional Gaussian blur. If the blur window size is $w$, there are $O\left(w^2\right)$ pixels sampled by the traditional Gaussian blur. On the other hand, there are only $O(w)$ pixels sampled in the two-pass Gaussian blur. This procedure is performed after 'light view pass' and before object shading.

After we obtain the blurred shadow map, we are able to compute the shadow factor $s_f$ for each fragment. In the fragment shader, we read $M_1$ and $M_2$ from the blurred depth map. If the the depth of the current fragment in light view space is less than $\mu$ in Equation 3.10, the fragment is not in a shadow. Otherwise, we compute the variance by Equation 3.10 and compute the shadow factor $s_f$, i.e., $p_{max}$.

### 3.3.2 Object Shading

We use the Phong lighting model to shade the objects [18]. The color $C$ of a fragment is calculated by

$$\begin{aligned} C &= s_f C' \\ C' &= I_l \left(C_a + (n \cdot v_l) C_d + \left(v'_e \cdot v_l\right)^k C_s\right) \\ v'_e &= 2\,(v_e \cdot n)\, n - v_e \end{aligned} \tag{3.12}$$

where $v_e$ is the normal vector that points to the current fragment from the camera, $v'_e$ is the reflection vector $v_e$, $s_f$ is the shadow factor computed by the method in previous section, $C_a$, $C_d$ and $C_s$ are the material properties and $n$ is the normal of the fragment. Quantity $v'_e$ can be computed from the fragment world position, and $C_a$, $C_d$ and $C_s$ are sampled from material textures. In the next section, we give more detail on this procedure. Vector $n$ is calculated automatically by the mesh vertex normal. Additionally, if the object mesh contains the normal maps, we use them to re-compute the new normal. This approach is called bump mapping [3].

Normal maps represent the local normal variety of a more fine surface, while vertex normals represent coarse normals of the surface. By using normal maps, we can compute a more detailed and accurate normal of each fragment. Thereby, the surface can have rich detail without increasing the number of vertices. The tangent and the bi-tangent of a vertex are necessary to implement the bump mapping. They are used to represent the local coordinate system around the vertex, called tangent space. The normals in a normal map are in the tangent space. Normally, a mesh data seldom contains tangent vectors and bi-tangent vectors but only normals, texture coordinates and positions. To compute the tangent space for each fragment, we first compute the tangent and bi-tangent for each vertex. We apply the method of Eric Lengyel [10]. The inputs of the method are the vertex normals and texture coordinates. The outputs are the tangent and bi-tangent of each vertex. This is performed when the system is initialized.

In addition to shadows, we need to also determine the color for the opaque and transparent objects. For example, consider the rendering of tree leaves. In order to improve performance, the leaves are typically represented by a single quad polygon with a leaf texture image on it. In the image, the leaf itself is opaque while other areas are transparent. By using alpha blending technology, the transparent area is removed. Still, there are some areas that are translucent such as the boundaries of leaves. This is good as it decreases aliasing on the boundaries. When the translucent area is rendered, however, the final results is affected by the rendering order. Suppose there are two leaves A and B. If A is in front of B but A is rendered first, we cannot see B from the translucent area on A. The reason is that when A is rendered, there is nothing on the scene. Therefore, A cannot blend its color with B. Accordingly, the leaf that is furthest from the camera should be rendered first. The order issue cannot be solved at the geometric level in most cases, so we solve it at the fragment level. The OIT approach is an accurate approach to address this, but it may suffer from a performance loss. A botanical tree is not well-suited for the OIT approach, because there may be many leaves that overlap at the same screen position. Since the major parts of a leaf are opaque, we adapt an approximate way to address translucency. This is done in two passes. In the first pass, only opaque parts of the objects are rendered. During rendering, alpha blending is disabled and depth writing is enabled. Then, in the second the pass, only non-opaque objects are rendered. This time, depth writing is disabled and alpha-blending is enabled. The reason we do not write depths to the z-buffer is that the completely transparent area should not affect the values in the z-buffer. Since there are already colors in the background, the error caused by the rendering order becomes less obvious. Especially when translucent areas are very small, we can hardly see any errors. This approach guarantees that the color of the non-opaque area can be blended with the color of opaque area

successfully. Still, it cannot guarantees that the color of a non-opaque area can be blended with the color of other non-opaque areas correctly.

User interaction is an important part of our real-time simulation system. In our system, the user can apply a force to the object with his/her mouse. The brute force method to finding the vertex that the user wants to drag is to search all the triangles to find one that is closest to the mouse cursor. However, this is slow and not suitable for real-time simulation if the mesh contains a large number of vertices. By using core profile OpenGL, mouse vertex selection can be implemented with a small amount of overhead. Our method benefits from the flexibility of manipulating the color frame buffer in OpenGL. In OpenGL, we can simultaneously write to at most eight color attachments for one framebuffer. The format of each color attachment is not limited to 3/4 channels and one byte per channel [21]. To implement the mouse selection, we do not render directly to the screen. Instead, we create a framebuffer with two color attachments to store the intermediate results. For the first color attachment, we store the rendered scene colors. For the second color attachment, we store the world space 3D positions of each fragment and an index. The index gives the triangle that the fragment is in. When the user selects one pixel on the screen, we can get the triangle index and pixel's 3D position in world space by reading pixel values from the second color attachment of the frame buffer. The second color attachment is a byproduct of the object shading process, and does not cost many extra resources. Even though we need to transfer the color attachment from the GPU back to the CPU in order to read it in each time step, we only need to transfer a small part of it, containing the pixel we want. After we have successfully read the pixel values from the frame buffer, we can directly find the triangle that was selected. This completes the mouse selection.

In summary, the object shading process is illustrated in Figure 3.10. After shading, we obtain a rendered scene color texture and an auxiliary texture for mouse selection. We use another rendering pass to place the color texture on the screen. To do this, only one triangle is drawn on the screen. The normalized device coordinates (NDC) of its vertices are $(3,1,0)$, $(-1,-3,0)$ and $(-1,1,0)$. The relationship between the triangle and the screen in the xy plane is shown in Figure 3.11. We display the color by mapping the color texture to the triangle.

### 3.3.3 Mesh Assembly

Domains are independent of each other, because they always have different rendering properties. However, rendering them sequentially is not the best choice, as this would generate too many draw calls in each timestep. To avoid this, we integrate the meshes of all do-

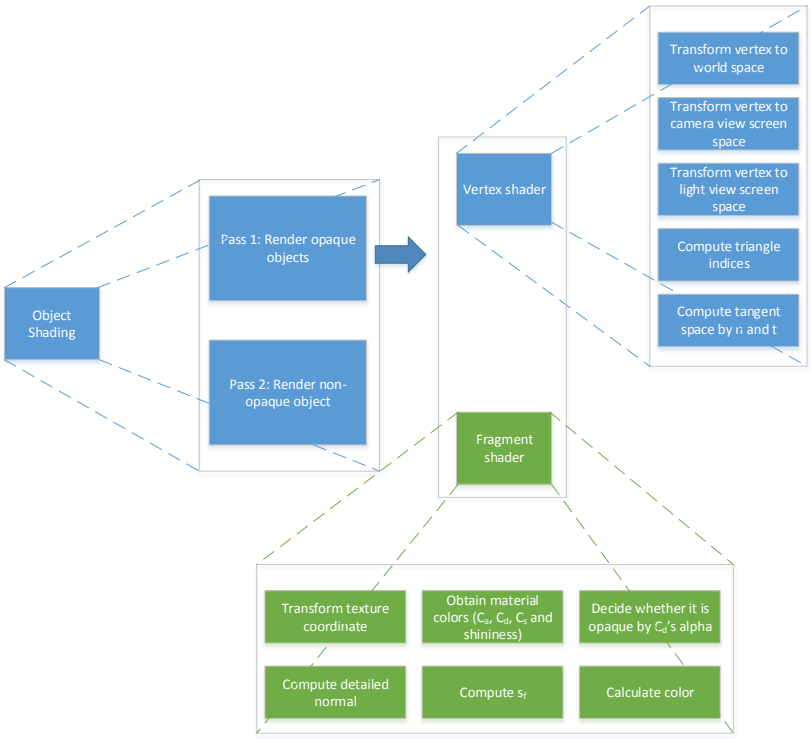



Figure 3.10 Object shading details.

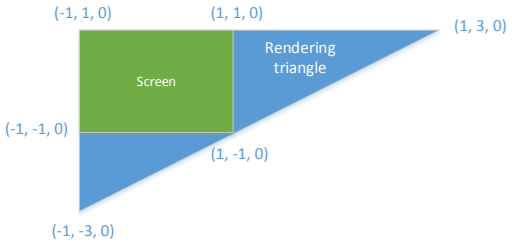



Figure 3.11 The relationship between the triangle and the screen in NDC space.

mains into one giant mesh. As mentioned in the previous section, the faces of every domain are unfolded to separate triangles, each containing tree vertices. These vertices are gathered together when they are updated by the CUDA kernels. When rendering, the program can directly use the vertex buffer generated by the GPU to draw all domains. However, a problem that impedes rendering is that domains have different rendering properties, making them hard to be "uniformized". In our system, we pack all the rendering materials of the different domains together. First of all, each triangle vertex is defined as follows:

```
struct vertex
{
    vec4 position;
    vec4 normal;
    vec4 tangent;
    vec4 transformed texture coordinate (TTC);
    vec4 material coordinate (MC);
};
```

Here, 'position', 'normal' and 'tangent' in the data block are 4-element vectors that store the world position, normal and tangent of the vertex, respectively. The 'tangent' is used for computing the tangent space and 'TTC' represents the texture coordinate. For convenience, it is also called the *uv* coordinate. The first two elements of 'TCC' store the original *uv* coordinates. The last two elements store a scale vector, used by the fragment shader. It is used to scale the *uv* coordinate to correctly sample the texture image. The last field 'MC' is the material coordinate. It denotes whether the vertex needs texture mapping, the position in the texture that stores the material information and the index of the texture that is sampled. Combined with 'TTC' and 'MC', we can compute the correct color of each vertex.

First, the textures used by all domains are packed into one texture array. In our system, we assume that each domain can have at most one texture image. Additionally, to speed up rendering, we assume that all domains share at most 512 separate texture images. Generally, most mesh models do not have that many different texture images. Moreover, if there was a mesh model that has lots of different texture images, they could be combined into one or several larger texture images. The most common case is that a mesh has a lot of separate groups, but only a few texture images. We use a texture array to pack those images together. When they are packed together, it is necessary to "uniformize" them to the same resolution. The maximum width and height among all the images is chosen as the final packed texture array's width and height. Denote width by $w_t$ and height by $h_t$. For each 2D texture in the texture array, the

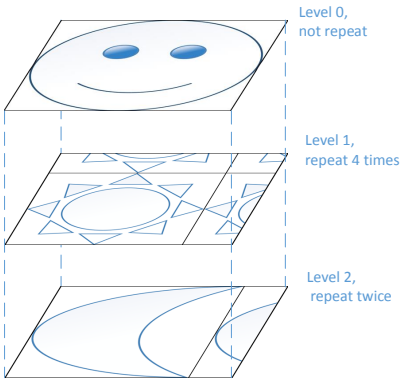



Figure 3.12 Texture images are packed into one texture array.

sizes of them become the same. If an image is smaller than $w_t \times h_t$, there is a blank area in the texture. We fill the blank area with the same image repeatedly, as illustrated in Figure 3.12. Still, for any image $i$, a scale 2D factor $s_i$ needs to transform the original texture coordinate $uv_i$, to accommodate the width and height of the texture array. Then the new $uv$ coordinate $uv_i'$ is computed as follows:

$$\begin{aligned} uv_i' &= (uv_i - \lfloor uv_i \rfloor)\, s_i \\ s_i &= \left( \frac{w_{t_i} - 1}{w_t - 1}, \frac{h_{t_i} - 1}{h_t - 1} \right)^T . \end{aligned} \tag{3.13}$$

The equation first wraps the $uv$ coordinate into $[0, 1)$. Then the wrapped $uv$ coordinate is scaled into the correct coordinate space. Accordingly, a 4-element vector needs to represent the $uv$ coordinate, which is the variable 'TTC'. If there are $n$ different images, the number of texture array's layers equals $n$. Additionally, another value is needed to denote the layer index of one texture image. Eventually, each vertex needs five values to sample the texture color. The fifth value, which is the layer index, is stored in the variable 'MC'. If normal texture images exist, they are packed in the same way.

Material properties of each domain are also packed. Typically, vertices in the same domain have the same material properties. In the Phong lighting model, the properties of a material are composed of ambient, diffuse, specular and shininess properties. Therefore, for a vertex, three 4-element vectors and one scalar are needed. Storing the material property for each vertex is not practical, as this generates duplicated data and unnecessarily consumes GPU memory. It also imposes extra burden on the vertex shader. To pack material properties of each domain

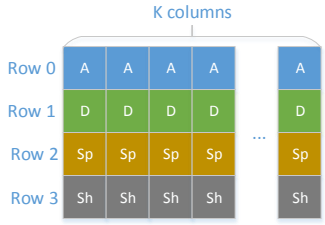



Figure 3.13 Materials are packed into one 2D texture. Blue is the ambient color, green is the diffuse color, yellow is the specular color, and gray is the shininess parameter.

compactly, we define a 2D texture image $M$ that stores the material properties. Image $M$ is a $K \times 4$ image where each pixel is represented by four floating-point numbers. Here, $K$ is the number of different materials. Each material property is stored in one column of the image, as shown in Figure 3.13. Given the column index, the material property of a vertex can easily be found in $M$ . One element of the variable 'MC' stores the column index of the material texture $M$. When a specific material is being fetched, the program does not use the default texture sampling function. Instead, the program just considers the image as a 2D array and directly loads data from $M$ without any sampling or filtering. As texture mapping is one of the bottlenecks of shader performance, this improve the rendering performance. The color of a vertex may come from either sampling a texture image, or material diffuse color. We also need a flag $s_{tex}$ to determine whether a vertex needs texture mapping or not. The flag is also stored in 'MC'. By using $s_{tex}$, the final diffuse color is computed by

$$C_d = s_{tex} M_d + (1 - s_{tex}) T_d, \tag{3.14}$$

where $C_d$ is the final diffuse color, $M_d$ is the material diffuse color and $T_d$ is the texture sampling color. Note that we don't use an *if* statement when $C_d$ is calculated, because the branch statement is slower than basic arithmetic operations.

# Chapter 4

# Results

This thesis work was performed in the environment listed in 4.1. We elaborate the testing environment in Tables 4.2 and 4.3. The experiments consist of two parts. The first part analyzes the CUDA algorithms and the new system architecture. The second part gives the rendering results.

Table 4.1 Our software environment.

| Name | Version |
|---|---|
| Operating system | Ubuntu 12.04 |
| Linux kernel | 3.2.0-74 |
| CUDA | 6.5 |
| OpenGL | 4.2 |
| g++ | 4.6.3 |

Table 4.2 Graphics card configuration (NVIDIA® GeForce® GTX 680).

| GPU Specifications | |
|---|---|
| CUDA cores | 1536 |
| Base clock | 1006 MHz |
| Boost clock | 1058 MHz |
| Texture fill rate | 128.8 billion/sec |
| **Memory Specification** | |
| Memory speed | 6.0 Gbps |
| Standard memory config | 2048 MB |
| Memory interface width | 256-bit GDDR5 |
| Memory bandwidth | 192.2 GB/sec |

Table 4.3 CPU and main memory configuration.

| CPU Specifications | |
|---|---|
| Number of CPUs | 2 |
| CPU model | Intel® Xeon® E5-2690 |
| Number of cores | 8 |
| Processor base frequency | 2.9GHz |
| Max turbo frequency | 3.8GHz |
| Max memory bandwidth | 51.2 GB/s |
| **Memory Specification** | |
| Memory size | 32 GB |
| Memory speed | 1333 MHz DDR3 |

## 4.1 CUDA Computation

### 4.1.1 Single Domain Uq Computation

Table 4.4 Statistics for the ratios of CPU to GPU running time for the single domain $Uq$ computation, under varying numbers of vertices ($n$) and modal dimensions ($r$).

| $r$ \ $n$ | $2^{14}$ | $2^{16}$ | $2^{18}$ | $2^{20}$ | $2^{22}$ |
|---|---|---|---|---|---|
| 10 | 2.53 | 0.68 | 1.44 | 1.6 | 1.77 |
| 20 | 6.47 | 2.76 | 5.01 | 6.54 | 7.04 |
| 30 | 5.32 | 3.67 | 6.60 | 8.02 | 9.69 |

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 domain. The CPU algorithm uses OpenMP, with 16 threads. The algorithm was tested by 50 random iterations and averages are reported ($t_{suq}$). The GPU running time ($t'_{suq}$) is measured in the same way. The floating point values of Table 4.4 are calculated as $t_{suq}/t'_{suq}$. From the table, we see that GPU has better performance in most cases. As the data size grows, GPU has better performance than the CPU. This demonstrates that our CUDA algorithm is more efficient than the traditional CPU algorithm.

### 4.1.2 Multi-domain Simulation

To measure the performance of multi-domain $Uq$ computation, we use 12 different botanical species as the testing data, listed in Table 4.5. The implementation is based on an existing multi-domain simulation system developed by Jernej Barbič and Yili Zhao [1, 27]. In this previous system, the $Uq$ computation and vertex updating are performed by the CPU. The rendering system used the compatibility profile OpenGL [16]. On the other hand, our system uses the core OpenGL profile and CUDA. The performance test is done by comparing our system with the previous system.

First, we test the memory usage of the mesh assembly. As described in Chapter 3, this will increase the memory usage. From Table 4.5, we can see that the number of vertices is quite close to the number of faces. This means that even though we unfolded the mesh into disconnected triangles, the memory usage does not increase significantly. Additionally, from the statistics in Table 4.6, we can see that the maximum GPU memory usage (Quercus Garryana) is much smaller than the total GPU memory size. Indeed, the GPU memory usage is related to the number of domains, the number of faces and the number of vertices. The number of domains determines how many transformation matrices there are. The number of faces determines the vertex buffer size. The number of vertices determines the dimension of the displacement vector $u$ and the dimension of the $U$ matrices.

Next, we test the our system performance. To measure the time accurately, we use a micro-second level timer during the experiment. We use "occupation" to measure the importance of a module in the simulation system. Occupation is defined as the time cost of a module divided by the the total frame time. Computed from data in Table 4.7, the average occupation of the $Uq$ computation is 2.62% for the previous system, and 1.11% for our system. We conclude that the $Uq$ computation time is not significant.

In contrast, the average occupation of vertex updating and rendering is quite high in the previous system, which is 77.0%, forming a bottleneck of the system. We therefore focused on optimizing this part. In our system, however, the average occupation of vertex updating and rendering is much smaller, i.e., 10.45%. The vertex updating and rendering are therefore not bottlenecks anymore. Other part of the computation, such as solving the model-reduced differential equation and the thread scheduling, become more critical for our system's performance.

Performance comparisons for each species are listed Table 4.7. For simple models our system is not as fast as the previous system, even though the FPS is high enough for real-time applications. The reason for this is that thread scheduling occurs more frequently, costing

us plenty of time. Conversely, the previous system is sequential for rendering and physical simulation. For complex models, our system is much faster (up to 5.33x).

For $Uq$ computation, our system is 1.71x-10.45x faster than the previous system, as illustrated in Table 4.7. This is a reasonable result, as the GeForce GTX 680 can compute 1536 floating-point operations per cycle. Additionally, for the vertex updating and rendering, our system is 1.15x-47.58x faster. The CUDA algorithm significantly improves efficiency. Moreover, Table 4.6 shows that the rendering time is always small, which is good for physical based simulation.

Table 4.5 Botanical model specifications. The table gives the #vertices ($v$), #faces ($f$), #vertices after unfolding ($v'$), #domains ($d$), #domains with $r > 0$ ($d'$), #modes ($r$) and total dimension of $u$ ($u$).

| Species | Geometric Info | | | Simulation Info | | | |
|---|---|---|---|---|---|---|---|
| | $v$ | $f$ | $v'$ | $d$ | $d'$ | $r$ | $u$ |
| Pansy | 7,987 | 15,372 | 46,476 | 22 | 22 | 324 | 106,882 |
| Conifer | 7,543 | 7,517 | 22,551 | 644 | 43 | 360 | 41,652 |
| Weed willow | 291,914 | 313,230 | 939,690 | 32,623 | 98 | 958 | 362,084 |
| Peach tree | 273,003 | 299,707 | 899,121 | 22,659 | 237 | 2,950 | 230,528 |
| Broad-leaved tree | 288,542 | 339,111 | 1,017,333 | 7,003 | 402 | 3,613 | 443,022 |
| Fir | 64,824 | 75,953 | 227,859 | 9,077 | 585 | 4,912 | 181,524 |
| Western red cedar | 40,480 | 55,810 | 167,430 | 4,408 | 800 | 5,987 | 205,112 |
| Incense cedar | 102,115 | 133,545 | 400,635 | 11,938 | 1,054 | 6,853 | 405,258 |
| Quercus Garryana | 586,668 | 224,926 | 674,778 | 120,871 | 871 | 9,520 | 1,000,860 |
| Blue spruce | 1,421,810 | 1,070,963 | 3,212,889 | 219,476 | 1,582 | 10,680 | 524,640 |
| Three firs | 194,472 | 227,859 | 683,577 | 27,232 | 1,755 | 14,736 | 544,572 |
| Eastern hemlock | 190,466 | 262,794 | 788,382 | 27,778 | 2,537 | 16,793 | 1,051,414 |

## 4.2 Rendering Effects

Figure 4.1 gives the final result of rendering a single tree (the FPS can be found in Table 4.7). Figure 4.2 shows the rendering results of multiple trees. Figure 4.3 gives the comparison between the tree trunk with bump mapping and without bump mapping. We can see the bump feeling of the truck texture. Figure 4.4 shows the comparison between VSM and standard shadow maps. The resolution of the depth map is $4096 \times 4096$. Compared to VSM, standard shadow maps cause aliasing on the shadow outlines. If we decrease the resolution, the aliasing problem is much more obvious, as illustrated in 4.5. Figure 4.6 shows the rendering with

Table 4.6 Statistic for $Uq$ computation time ($t_{uq}$), vertex updating time ($t_{vb}$), vertex updating with rendering time ($t_{vbr}$), frames per second (FPS), GPU memory usage ($s$) and #kernels of $Uq$ computation ($n_{uqk}$).

| Species | CPU | | | GPU | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | FPS | $t_{uq}$ (ms) | $t_{vbr}$ (ms) | FPS | $t_{uq}$ (ms) | $t_{vb}$ (ms) | $t_{vbr}$ (ms) | $s$ (MB) | $n_{uqk}$ |
| Pansy | 290 | 0.323 | 2.606 | 75 | 0.189 | 1.73 | 2.248 | 9.7 | 2 |
| Conifer | 60 | 0.261 | 5.886 | 55 | 0.165 | 1.059 | 1.693 | 5.8 | 1 |
| Weed willow | 3 | 8.066 | 261.926 | 16 | 1.12 | 5.766 | 6.483 | 129.8 | 2 |
| Peach tree | 5 | 1.978 | 150.619 | 10 | 0.713 | 6.949 | 9.692 | 119.6 | 2 |
| Broad-leaved tree | 10 | 3.341 | 80.008 | 25 | 0.391 | 6.417 | 7.01 | 150.9 | 2 |
| Fir | 10 | 1.949 | 63.729 | 28 | 0.329 | 1.776 | 2.335 | 35.5 | 2 |
| Western red cedar | 17 | 1.434 | 42.368 | 32 | 0.283 | 1.727 | 2.611 | 25.6 | 2 |
| Incense cedar | 7 | 3.299 | 120.238 | 20 | 0.476 | 4.132 | 4.817 | 59.7 | 1 |
| Quercus Garryana | 1 | 22.837 | 784.792 | 3 | 4.768 | 39.822 | 40.575 | 906.3 | 2 |
| Blue spruce | 1 | 12.152 | 1302.294 | 2 | 6.997 | 15.86 | 16.409 | 424.3 | 1 |
| Three firs | 3 | 3.268 | 261.394 | 15 | 0.714 | 4.978 | 5.494 | 103.6 | 2 |
| Eastern hemlock | 3 | 8.703 | 240.762 | 11 | 0.833 | 11.457 | 12.216 | 165.2 | 2 |

Table 4.7 Statistic for $Uq$ computation time comparison ($k_{uq} = t_{uq}/t'_{uq}$), vertex updating and rendering time comparison ($k_{vbr} = (t_{vbr}) \,/\, \left(t'_{vbr}\right)$), FPS comparison ($k_{fps} = FPS_{GPU}/FPS_{CPU}$), $Uq$ computation time occupation ($O_{uq}$) and vertex buffer generation and rendering time occupation ($O_{vbr}$). $t'$ is the time cost using new method.

| Species | Comparison | | | $O_{uq}$ | | $O_{vbr}$ | |
|---|---|---|---|---|---|---|---|
| | $k_{uq}$ | $k_{vbr}$ | $k_{fps}$ | CPU | GPU | CPU | GPU |
| Pansy | 1.71 | 1.16 | 0.26 | 9.37% | 1.42% | 75.57% | 16.86% |
| Conifer | 1.58 | 3.48 | 0.92 | 1.57% | 0.91% | 35.32% | 9.31% |
| Weed willow | 7.20 | 40.40 | 5.33 | 2.42% | 1.79% | 78.58% | 10.37% |
| Peach tree | 2.77 | 15.54 | 2.00 | 0.99% | 0.71% | 75.31% | 9.69% |
| Broad-leaved tree | 8.54 | 11.41 | 2.50 | 3.34% | 0.98% | 80.01% | 17.52% |
| Fir | 5.92 | 27.29 | 2.80 | 1.95% | 0.92% | 63.73% | 6.54% |
| Western red cedar | 5.07 | 16.23 | 1.88 | 2.44% | 0.91% | 72.03% | 8.36% |
| Incense cedar | 6.93 | 24.96 | 2.86 | 2.31% | 0.95% | 84.17% | 9.63% |
| Quercus Garryana | 4.79 | 19.34 | 3.00 | 2.28% | 1.43% | 78.48% | 12.17% |
| Blue spruce | 1.74 | 79.36 | 2.00 | 1.22% | 1.40% | 130.23% | 3.28% |
| Three firs | 4.58 | 47.58 | 5.00 | 0.98% | 1.07% | 78.42% | 8.24% |
| Eastern hemlock | 10.45 | 19.71 | 3.67 | 2.61% | 0.92% | 72.23% | 13.44% |

Figure 4.1 Rendering results of a single tree. (Top left: Broad-leaves tree. Top Right: Peach tree. Bottom left: Fir. Bottom right: Eastern hemlock)

Figure 4.2 Rendering results of multiple trees. (Top: Four firs. Bottom: Three species and 20 trees)

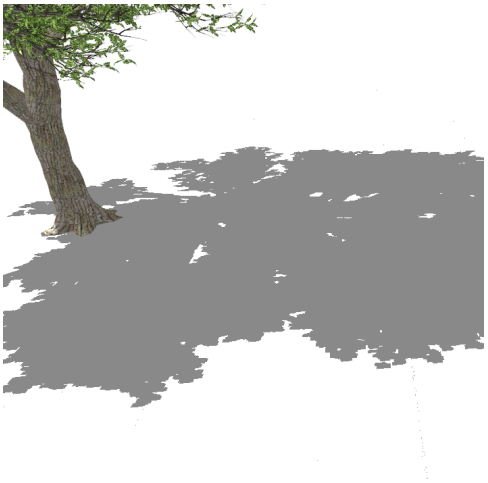
(a) Peach tree trunk with bump mapping

(b) Peach tree truck without bump mapping

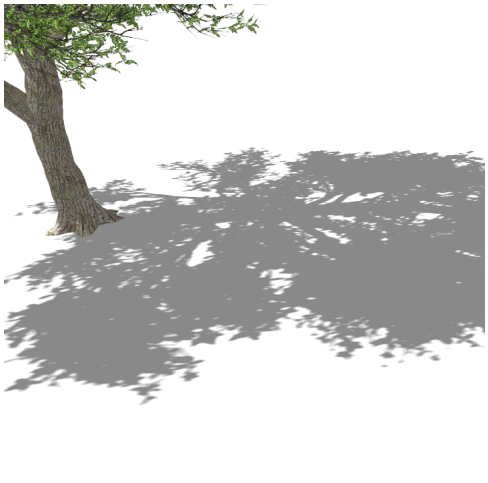
(c) Blue spruce trunk with bump mapping

(d) Blue spruce truck without bump mapping

Figure 4.3 Bump mapping effect comparison.

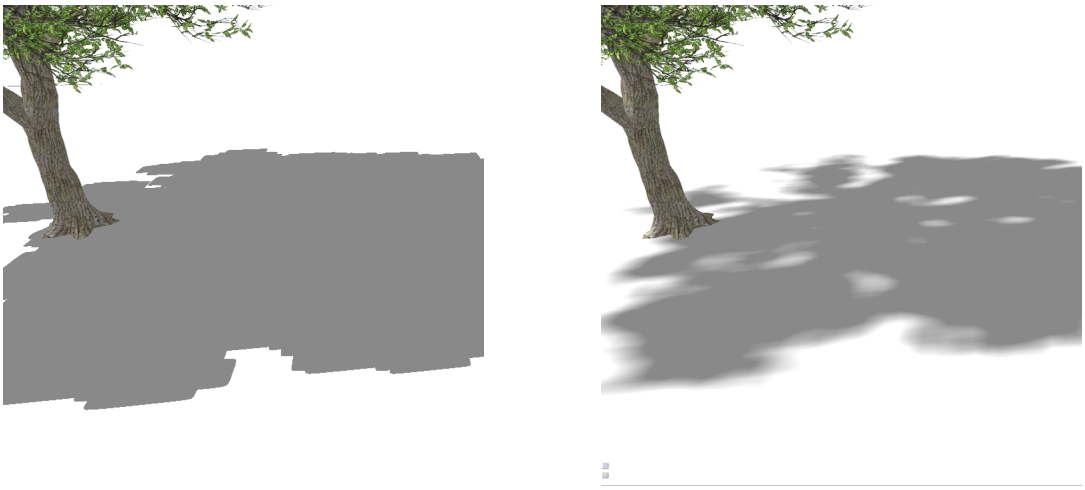


(a) Standard shadow maps.



(b) Variance shadow map.

Figure 4.4 Shadow maps comparison using a depth map with a resolution of $4096 \times 4096$.



(a) Standard shadow maps. (b) Variance shadow map.

Figure 4.5 Shadow maps comparison using a depth map with a resolution of $512 \times 512$.

(a) Render with transparency. (b) Render without transparency.

Figure 4.6 Transparency rendering.

transparency. The boundaries of leaves are mainly transparent which eliminates boundary aliasing.

# Chapter 5

# Conclusion

The aim of this thesis – to improve performance of model reduction by using GPU – has been achieved through the completion of the objectives in three key areas, namely:

(1) Development of a novel CUDA algorithm for $Uq$ computation and vertex updating. The algorithms for single domain $Uq$ computation, multi-domain $Uq$ computation and vertex updating have been described. Experimental results show better performance compared to pre-existing systems.

(2) Design of the new system architecture for CPU-GPU model reduction simulation. The architecture utilizes GPUs and CPUs in parallel. The architecture accelerates performance and improves GPU hardware utilization.

(3) Design of a new rendering system. The rendering system is developed by using the core OpenGL profile. By using the combination of the existing rendering algorithms, the rendering system improves the final effects as well as the performance.

This thesis has been successful in accelerating the system performance and improving the rendering effects and has achieved good results on the available data sets. However, additional work may be devoted to further improving the performance and rendering results. Therefore, the main recommendations for further work include:

(1) Improvement of the performance of thread scheduling and synchronization. As shown in the experimental results, the CUDA algorithms are fast while the FPS is still low. The problem could be alleviated by analyzing the critical resources and the parallel structure of threads in our system.

(2) Improvement of the rendering effects. The rendering system still has room for improvement. As shown in the experimental results, rendering occupies a relatively small number of GPU resources. Therefore, more rendering techniques could be added to our system. Since

deferred rendering pipeline is already implemented in our system, the post-processing effects such as fast approximate anti-aliasing and blooming could be added into the system in order to enhance the rendering effects.

# References


[1] Barbič, J. and Zhao, Y. (2011). Real-time large-deformation substructuring. In *ACM SIGGRAPH 2011 Papers*, SIGGRAPH '11, pages 91:1–91:8, New York, NY, USA. ACM.

[2] Bell, N. and Garland, M. (2008). Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation.

[3] Blinn, J. F. (1978). Simulation of wrinkled surfaces. *ACM SIGGRAPH Computer Graphics*, 12(3):286–292.

[4] Cozzi, P. and Riccio, C. (2012). *OpenGL Insights*. CRC Press. http://www.openglinsights.com/.

[5] Crow, F. C. (1977). Shadow algorithms for computer graphics. In *Acm siggraph computer graphics*, volume 11, pages 242–248. ACM.

[6] Donnelly, W. and Lauritzen, A. (2006). Variance shadow maps. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 161–165. ACM.

[7] ETH (2011). CUDA memory architecture. http://www.cvg.ethz.ch/teaching/2011spring/gpgpu/cuda_memory.pdf.

[8] INTEL (1998). Write combining memory implementation guidelines. http://download.intel.com/design/PentiumII/applnots/24442201.pdf.

[9] Kubisch, C. (2014). Order independent transparency in opengl 4.x. http://on-demand.gputechconf.com/gtc/2014/presentations/S4385-order-independent-transparency-opengl.pdf.

[10] Lengyel, E. (2012). *Mathematics for 3D game programming and computer graphics*. Cengage Learning.

[11] McDonald, J. (2013). Alpha blending: To pre or not to pre. https://developer.nvidia.com/content/alpha-blending-pre-or-not-pre.

[12] Nguyen, H. (2007). *GPU gems 3*. Addison-Wesley Professional.

[13] NVIDIA (2012). GTX 680 Kepler whitepaper. http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf.

[14] NVIDIA (2014a). CUDA C programming guide. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[15] NVIDIA (2014b). CUDA home page. http://www.nvidia.com/object/cuda_home_new.html.

[16] OpenGL (2015). Core and compatibility in contexts. https://www.opengl.org/wiki/Core_And_Compatibility_in_Contexts.

[17] Pharr, M. and Fernando, R. (2005). *GPU gems 2: programming techniques for high-performance graphics and general-purpose computation*. Addison-Wesley Professional.

[18] Phong, B. T. (1975). Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317.

[19] Porter, T. and Duff, T. (1984). Compositing digital images. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, pages 253–259, New York, NY, USA. ACM.

[20] Reeves, W. T., Salesin, D. H., and Cook, R. L. (1987). Rendering antialiased shadows with depth maps. In *ACM SIGGRAPH Computer Graphics*, volume 21, pages 283–291. ACM.

[21] Segal, M. and Akeley, K. (2011). The OpenGL graphics system: A specification. https://www.opengl.org/registry/doc/glspec42.core.20110822.withchanges.pdf.

[22] Stamminger, M. and Drettakis, G. (2002). Perspective shadow maps. In *ACM Transactions on Graphics (TOG)*, volume 21, pages 557–562. ACM.

[23] Wexler, D., Gritz, L., Enderton, E., and Rice, J. (2005). GPU-accelerated high-quality hidden surface removal. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '05, pages 7–14, New York, NY, USA. ACM.

[24] Williams, L. (1978). Casting curved shadows on curved surfaces. In *ACM Siggraph Computer Graphics*, volume 12, pages 270–274. ACM.

[25] Woolley, C. (2013). GPU optimization fundamentals. https://www.olcf.ornl.gov/wp-content/uploads/2013/02/GPU_Opt_Fund-CW1.pdf.

[26] Zhang, F., Sun, H., Xu, L., and Lun, L. K. (2006). Parallel-split shadow maps for large-scale virtual environments. In *Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications*, pages 311–318. ACM.

[27] Zhao, Y. (2014). *Plant Substructuring and Real-time Simulation Using Model Reduction*. PhD thesis, University of Southern California.