Exploiting the Graphics Hardware to solve two compute intensive problems: Singular Value Decomposition and Ray Tracing Parametric Patches

Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science (by Research)
in
Computer Science

by

Sheetal Lahabar
200402024
sheetal@research.iiit.ac.in
lahabarsheetal@gmail.com

Center for Visual Information Technology
International Institute of Information Technology
Hyderabad - 500032, INDIA
August 2010

International Institute of Information Technology
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled "Exploiting the Graphics Hardware to solve two compute intensive problems: Singular Value Decomposition and Ray Tracing Parametric Patches " by Ms. Sheetal Lahabar (200402024), has been carried out under my supervision and is not submitted elsewhere for a degree.

Date	Adviser: Prof. P. J. Narayanan

To Lata, My Mother

Abstract

The rapid increase in the performance of graphics hardware have made the GPU a strong candidate for performing many compute intensive tasks, especially many data-parallel tasks. Off-the-shelf GPUs are rated at upwards of 2 TFLOPs of peak power today. The high compute power of the modern Graphics Processing Units (GPUs) has been exploited in a wide variety of domain other than graphics. They have proven to be powerful computing work horses in various fields. We leverage the processing power of GPUs to solve two highly compute intensive problems: Singular Value Decomposition and Ray Tracing Parametric Patches. Singular Value Decomposition is an important matrix factorization technique used to decompose a matrix into several component matrices. It is used in a wide variety of application related to signal processing, scientific computing, etc. However, little work has been done to implement Singular Value Decomposition on parallel architectures. Direct ray tracing of parametric patches provides high geometric precision resulting in less artifacts and results in better rendering using true normals for lighting at every pixel. Researchers have developed algorithms to ray tracing parametric surfaces directly, but these algorithms are either computationally expensive or have other disadvantages. Both these problems are both computationally demanding and amenable to massively parallel implementations. Singular Value Decomposition computations requires factorizing a dense matrix into component matrices. Finding the intersection of a ray with a Bézier patch requires finding the roots of a 18 degree polynomial. They require enormous number of floating point operations. The advent of GPUs with support for higher precision and their ever-growing amount of parallel horsepower makes them a tempting resource for such numerical computations. We propose parallel implementations for these problems on a GPU. They outperform the best CPU implementations available significantly.

We implement the Singular Value Decomposition of a dense matrix on GPUs using the two step Golub Reinsch algorithm. In the first step, bidiagonalization decomposes a given dense matrix into bidiagonal matrix and component orthogonal matrix using a series of Householder transformations. In the next step, diagonalization is performed by applying the implicitly shifted QR algorithm which takes $O(n)$ iterations. It decomposes the bidiagonal matrix into a diagonal matrix and orthogonal matrices. Each householder transformation reduces a row-column pair. We divide the number of householder transformations required into n/L blocks. L householder transformations are applied in parallel by mapping them to CUBLAS operations to derive maximum performance. We thus, exploit the underlying hardware to the maximum. We use a hybrid implementation for the diagonalization step that splits the computations between the CPU and the GPU. Every iteration of the QR algorithm applies Givens rotations on the elements of the bidiagonal matrix and the corresponding inverse rotations are applied on the rows of the orthogonal matrices which are initially identity. Transformation applied on each row depends on the previous row and transformation applied on it, thus making computations on every row independent. Rotations on the elements of the bidiagonal matrix are applied on the CPU. The inverse transformations are performed on every row in parallel on the GPU. We combine the power and functionality of using CUDA and the high performance software libraries available with it to exploit the GPU parallelism and achieve high computing performance. This approach can be used for solving other graphics and non-graphics tasks. Our complete Singular Value Decomposition implementation outperforms the MATLAB and Intel ®Math Kernel Library (MKL) LAPACK implementation significantly on the CPU. We show a speedup of upto 60 over the MATLAB implementation and upto 8 over the Intel MKL implementation on a Intel Dual Core 2.66GHz PC with NVIDIA GTX 280. We are able to compute the Singular Value Decomposition of very large matrices, upto 14K which is otherwise impossible on the CPU due to memory limitations. The GPUs are limited to single precision numbers, though that is changing with the newer generations. The error due to lower precision was less than 0.001%.

We present correct ray tracing of parametric bicubic surfaces on the GPU using Kajiya's approach without subdividing to the GPU. We use a BVH representation of the patches to remove non-intersecting rays. The process starts with a list of rays to be traced in each frame. The BVH is traversed for each ray to enumerate the potential ray-patch intersections. This method forms a univariate 18-degree polynomial for each intersection and finds its roots using the Laguerre's method. The real roots are the parameter values at the intersection points. Polynomial formation and root finding for each ray-patch intersection can be performed independently, making it ideal for processing on the GPU. Our algorithm treats all bounces in a modular fashion. Rays for a subsequent bounce are generated at the end of each bounce and process can be repeated. It finds the exact points of intersection and it is able to provide exact lighting using per pixel normal. We perform the BVH traversal, finding the v-root using Laguerre's method, finding the u-root using a GCD computation, generating rays for subsequent bounces, and shading on the GPU in parallel. The bulk of the time (as high as 82%) is spent on polynomial root finding. Slow double precision computations need to be used for it. Direct ray tracing facilitates shading at each pixel using the true normals. The ray tracing time depends on the number of ray-patch intersections evaluated. Each intersection takes around 3.7 microseconds on GTX 280. This makes the method faster on future hardware and highly amenable to multi-GPU processing. The time taken is about 466 milliseconds for primary rays on the Killeroo model, with similar times for subsequent bounces, on a GTX 280. We achieve a speed up of 340x on NVIDIA GTX 280 and 990x on NVIDIA GTX 480 over the MATLAB implementation of the Kajiya's algorithm on a AMD Dual Core Processor PC. The limiting factors of performance are slow double precision arithmetic and the low amount of shared memory on today's GPUs. The next generation of GPUs have significantly improved both these aspects and we expect to see interactive rates on them. Our algorithm exploits parallelism by dividing the computations into independent tasks.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graphics Processing Unit (GPU) is a processor inside a graphics card commonly used for graphics rendering. It offloads 3D or 2D graphics rendering from the processor. It is mainly used for playing 3D games of high-end 3D rendering and in embedded systems, mobile phones, personal computers, work stations, and game consoles. It implements a number of graphics primitive operations in a way that makes running them much faster. Modern day GPUs are efficient at manipulating computer graphics, and their highly parallel structure makes them more effective than general-purpose CPUs for a range of complex algorithms.

Recent graphics architectures provide high memory bandwidth and computational horsepower. For example, the NVIDIA GeForce GTX 280 can achieve a sustained 141 GB/sec of memory bandwidth, and 1 TFLOPs of theoretical peak performance. Graphics hardware performance doubled every six months in recent past. The rate of growth outpaces the oft-quoted Moores Law as applied to traditional microprocessors; compare to a yearly rate of roughly 1.4x for CPU performance as shown in Figure 1.1. The disparity in performance can be attributed to fundamental architectural differences: CPUs are optimized for high performance on sequential code, many transistors are dedicated to supporting non-computational tasks like branch prediction and caching. Highly parallel nature of graphics computations enables GPUs to use additional transistors for computation, achieving high arithmetic intensity with the same transistor count. The raw speed, increased precision, and rapidly expanding programmability of the hardware make it an attractive platform for general-purpose computation.

The rapid increase in the performance of graphics hardware, coupled with recent improvements in its programmability, have made graphics hardware a compelling platform for computationally demanding tasks in a wide variety of application domain other than graphics. We leverage the processing power of GPUs to solve two highly compute and data intensive problems: Singular Value Decomposition and Ray Tracing Parametric Patches.

Computing the Singular Value Decomposition of a dense matrix and ray tracing parametric patches requires enormous number of floating point operations. Although, the problems are not related, they are both computationally demanding and amenable to massively parallel implementations. Singular Value Decomposition computation requires factorizing a dense matrix into component matrices whereas

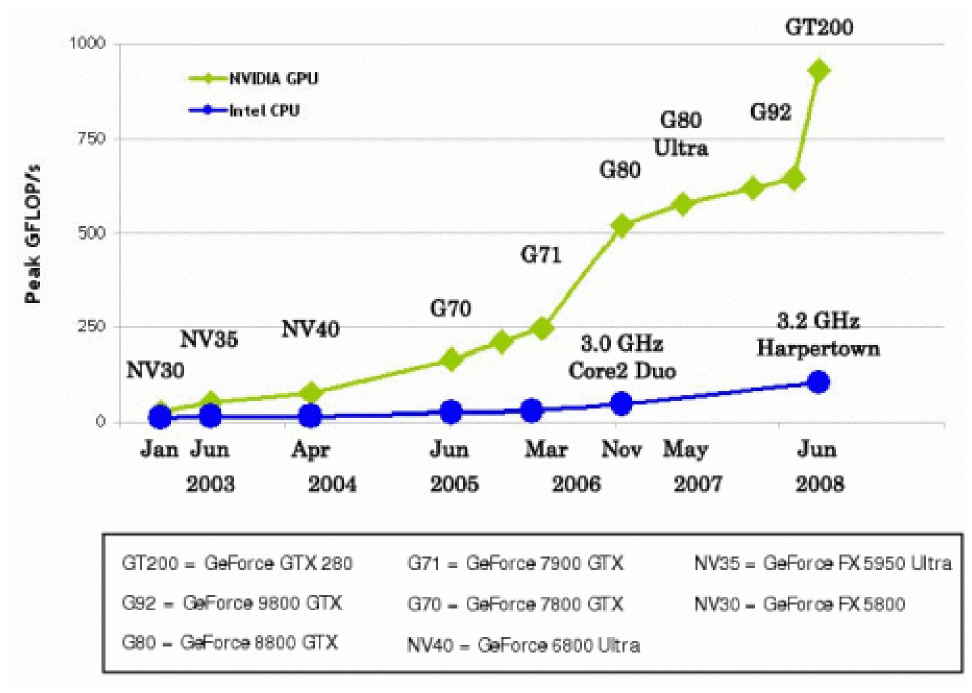


Figure 1.1 CPU/GPU performance growth plot. Theoretical performance of graphics processors is much higher than CPU performance. Image obtained from the documents provided by NVIDIA.

finding the intersection of a ray with a Bézier patch requires finding the roots of a very high degree polynomial. Both these operations require tremendous amounts of floating point calculations. The advent of GPUs with support for higher precision and their ever-growing amount of parallel horsepower makes them a tempting resource for such numerical computations. Our GPU implementations exploits the architecture well and outperforms the best CPU implementations available significantly.

Singular Value Decomposition is a way of factoring matrices into a series of linear approximations that expose the underlying structure of the matrix and has many applications. However, not much work was done for implementing it on the GPU. Ours is, the recent implementation of the Singular Value Decomposition algorithm on the GPU. Computing the Singular Value Decomposition requires factorizing the given dense matrix into a bidiagonal one by applying a series of Householder transformations. The bidiagonal matrix is then reduced to a diagonal matrix by applying the implicit shifted QR algorithm. The Singular Value Decomposition algorithm has a high computational complexity and requires enormous data transfer from the device. The computations can be mapped well on the GPU. The householder transformations required for bidiagonalizing a matrix are mapped to GPU BLAS operations. We propose a hybrid algorithm for diagonalizing a bidiagonal matrix. It divides the computations between CPU and GPU to achieve maximum performance. Our implementation outperforms the popular and fast CPU implementations significantly.

Ray tracing is one of the major methodologies in realistic image synthesis. The heart of ray tracer lies in the ray-environment intersection routine. Our work deals with designing a ray tracer for direct ray tracing of parametric surfaces, especially Bézier surfaces. However, one problem with Bézier patches is that calculating their intersections with lines is difficult making them awkward for pure ray tracing. Several researchers have developed algorithm for intersecting rays with parametric surfaces. Unfortunately, most of these algorithms are either very expensive or have other disadvantage. We address the above issue by adapting Kajiya's ray patch intersection algorithm to the GPU. It is robust and finds the exact point of intersection of a ray with a patch. It requires finding the roots of an 18 degree polynomial per ray patch intersection. It requires enormous number of double precision operations. Our algorithm renders highly accurate scenes in interactive time with shadows and reflections.

1.1 GPU architecture

Commodity computer graphics chips are probably today's most powerful computational hardware for unit cost. GPUs have gone from afterthought peripherals to modern, powerful, and programmable processors in their own right. Many researchers and developers have become interested in harnessing the power of commodity graphics hardware for general-purpose computing.

In 3D computer graphics, a rendering pipeline is meant to accept certain representations of a 3D scene as input and produce a fast 2D raster image as output. The various pipeline stages are implemented as fixed functions on hardware for graphics acceleration. The vertex and fragment stages were made programmable in 2005. They were programmed using shaders. The shader model 4.0 involves significant changes to the rendering pipeline. It keeps all the shaders at the same level calling it a unified architecture, because of which all shaders have fast access to the textures. It introduced a new programmable stage called the geometry shader which gives the capability of generating more vertices and primitives and saves the CPU to GPU bandwidth. The new rendering pipeline in Figure 1.2 also features streaming data out in the middle of the pipeline to the memory. This gives the facility of not going through the whole pipeline to process stream data which is not to be rendered.

The Shader Model 4.0 GPUs use the same processor core to implement vertex, geometry and fragment processing. Separate processors with different capabilities were used for different stages of the earlier GPUs. It dynamically allocate the available processing resources to vertex, geometry and pixel units are demanded by the load increasing resource utilization. Unified architecture provides the advantage to access the texture memory for all shaders. It allows to store geometry in form of textures and efficiently access the same from vertex shader.

NVIDIA kicked off the GPU computing revolution with the introduction of CUDA, its massively parallel computing ISA and hardware architecture. CUDA has unleashed unprecedented performance boosts across a wide range of applications. First introduced with the NVIDIA GeForce 8800 GPU and now standard across all NVIDIA's modern GPUs, CUDA is the foundation of NVIDIA's parallel computing strategy. The addition of OpenCL is another major milestone in the GPU revolution that gives developers another powerful programming option. OpenCL (Open Computing Language) is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs

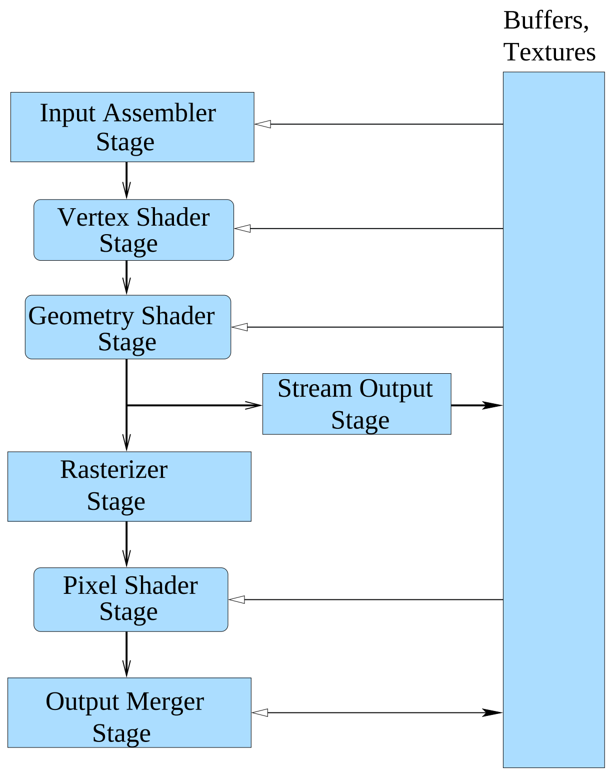


Figure 1.2 Shader Model 4

and other processors. It allows developers to harness the massive parallel computing power of the GPU. Its architecture shares a range of computational interfaces with two competitors, NVIDIA's Compute Unified Device Architecture and Microsoft's Direct Compute.

GPUs are high performance, many-core processors capable of very high computation and data throughput. Once specially designed for computer graphics and difficult to program, todays GPUs are general-purpose parallel processors with support for accessible programming interfaces and industry standard languages such as C. Developers who port their applications to GPUs often achieve speedups of orders of magnitude vs. optimized CPU implementations.

Figure 1.3 shows the design of a GPU. The key to performance is the multitude of thread processors. In the figure, there are eight thread processors and in each sixteen multiprocessors, for 128 thread processors total. NVIDIA delivers GPUs with up to 30 multiprocessors and 240 thread processors. In each clock, each TP can produce a result, giving this design a very high peak performance rating.

Each multiprocessor executes in SIMD mode, meaning each thread processor in that multiprocessor executes the same instruction simultaneously. If one thread processor is executing a floating point add, they all are; if one is doing a memory read, they all are. Moreover, each instruction is quad-clocked. Meaning the SIMD width is 32, even though there are only eight thread processors. Many scalar threads happen to be executing in SIMD mode, called SIMT execution. Careful orchestration of the 32 threads

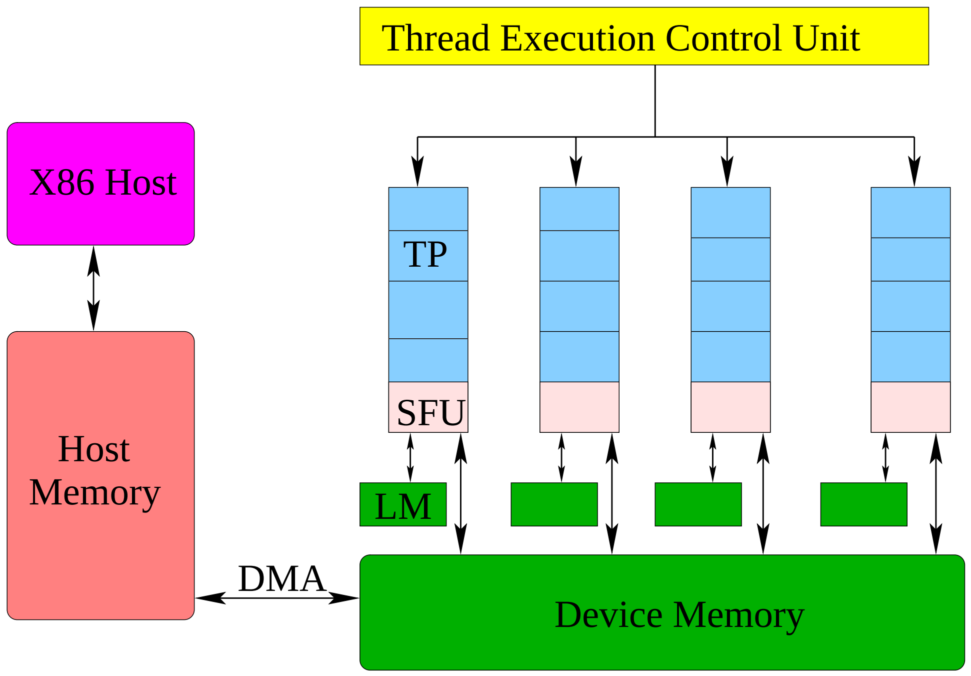


Figure 1.3 GPU architecture.

that execute in SIMD mode is necessary for best performance. The multiprocessors themselves execute asynchronously, and without communication.

Each multiprocessor has a special function unit, which handles infrequent, expensive operations, like divide, square root and so on; it operates more slowly than other operations, but since it is infrequently used, it doesnt affect performance. There is a high bandwidth, low latency local memory attached to each multiprocessor. The threads executing on that multiprocessor can communicate among themselves using this local memory. In the current NVIDIA generation, the local memory is quite small (16KB).

There is also a large global memory (over 1GB). This is physical, not virtual and paging is not supported, so all the data has to fit in memory. The device memory has very high bandwidth, but high latency to the multiprocessors. The device memory is not directly accessible from the host, nor is the host memory directly visible to the GPU. Data from the host that needs to be processed by the GPU must be moved via DMA (across the IO bus) from the host to the device memory and the results are moved back.

There is a hierarchy of parallelism; threads executing within a multiprocessor can share and communicate using the local memory, while threads executing on different multiprocessors cannot communicate or synchronize. This hierarchy is explicit in the programming model as well. Parallelism comes in two flavors; outer, asynchronous parallelism between thread groups or thread blocks, and inner, synchronous parallelism with a thread block. All the threads of a thread block will always be assigned as a group to a single multiprocessor, while different thread blocks can be assigned to different multiprocessors.

Because of the high latency to the device memory, the multiprocessors are highly multithreaded as well. When one set of SIMD threads execute a memory operation, the multiprocessor will switch to execute another set of SIMD threads. The other SIMD threads may be part of the same thread block, or may come from a different block assigned to the same multiprocessor.

The GPU is designed as a throughput engine; its designed to get a lot of work done on a lot of data, all very quickly, all in parallel. To get high performance, the program needs enough parallelism to keep the thread processors busy. Each thread block needs enough parallelism to fill all the thread processors in a multiprocessor, and at least as many thread blocks as you have multiprocessors. More parallelism must be maintained to keep the multiprocessors busy when they need to thread switch past a long latency memory operation.

1.2 Singular Value Decomposition

The Singular Value Decomposition is an important technique used to decompose a rectangular real or complex matrix into several component matrices, exposing many of the useful and interesting properties of the original matrix. The matrix is decomposed into a set of factors (often orthogonal and independent) that are optimal based on some criterion. It represents an expansion of the original data in a coordinate system where the covariance matrix is diagonal. Matrix computations using the Singular Value Decomposition are more robust to numerical errors.

Using the Singular Value Decomposition, one can determine the dimension of the matrix range called the rank. The Singular Value Decomposition can also quantify the sensitivity of a linear system to numerical error or obtain a matrix inverse. It also provides solutions to least-squares problems and handles situations when matrices are either singular or numerically very close to singular.

A Singular Value Decomposition of an $m \times n$ matrix A is any factorization of the form

$$A = U \Sigma V^T \tag{1.1}$$

where U is an $m \times m$ orthogonal matrix, V is an $n \times n$ orthogonal matrix, and Σ is an $m \times n$ diagonal matrix with elements $s_{ij} = 0$ if $i \neq j$ and $s_{ii} \geq 0$ in descending order along the diagonal.

The Singular Value Decomposition of a matrix A can be computed using the Golub-Reinsch (Bidiagonalization and Diagonalization) algorithm or the Hestenes method. We use the Golub-Reinsch method as it is simple and compact, and maps well to the SIMD GPU architecture. It is also popular in many numerical libraries. Hestenes algorithm is a Jacobi type approach that gives low performance and hence not popular. Golub-Reinsch algorithm is a two step algorithm [51]. LAPACK Singular Value Decomposition routine implements this method. In the first step, the matrix is reduced to a bidiagonal matrix using Householder reflections. This takes $O(mn^2)$ floating point operations, assuming that $m \geq n$. The second step is to compute the Singular Value Decomposition of the bidiagonal matrix using implicitly shifted QR iterations. In practice, it suffices to compute the Singular Value Decomposition up to a certain precision taking $O(n)$ iterations, each costing $O(n)$ flops. Thus, the first step is more expensive, and the overall cost is $O(mn^2)$ flops.

Algorithm 1 outlines the Singular Value Decomposition algorithm for a input matrix A.

```
1: $B \leftarrow Q^T AP$          {Bidiagonalization of $A$ to $B$}
2: $\Sigma \leftarrow X^T BY$          {Diagonalization of $B$ to $\Sigma$}
3: $U \leftarrow QX$
4: $V^T \leftarrow (PY)^T$          {Compute orthogonal matrices $U$ and $V^T$ and Singular Value
   Decomposition of $A = U\Sigma V^T$}
```

Algorithm 1: Singular Value Decomposition

Singular Value Decomposition is widely used in applications related to principal component analysis, pattern recognition and image processing for singular value spectral analysis. The Singular Value Decomposition also has a variety of applications in signal processing, scientific computing, automatic control, and many other areas. The Singular Value Decomposition is also applied extensively to the study of linear inverse problems, and is useful in the analysis of regularization methods and plays a crucial role in the field of Quantum information and numerical weather prediction.

1.3 Ray Tracing Parametric Patches

Ray tracing is one the most important technique for generating realistic images. It produces very high degree of photo realism. It is capable of simulating a wide variety of effects, such as reflection, refraction and shadows. The serious disadvantage of ray tracing is the performance since it deals with thousands of ray-object intersections for every image. Surface representations such as splines, NURBS or subdivision surfaces provides a simple and powerful method of describing three-dimensional geometrical objects for use in computer graphics applications. They form the foundation of most CAD systems used in the industry today. Due to the increasing popularity it becomes more important to ray trace these shapes fast and in high quality. Generally, in a modern ray tracing system, triangle meshes are commonly used for defining 3D shapes or free-form surfaces are tessellated into triangles which may cause visual artifacts to appear. Moreover, triangulating the surfaces has high preprocessing costs and require large amount of memory for storing the tessellated triangles. These problems can be avoided by directly ray tracing free form surfaces. It reduces the preprocessing costs and requires less memory due to smaller number of primitives. Rendering free-form surfaces directly provides higher geometric precision resulting in less artifacts. Better rendering is achieved by using true normals for lighting at every pixel.

Bézier patches are visually intuitive, and for many applications, mathematically convenient. They can be of any degree, but bicubic Bézier surfaces generally provide enough degrees of freedom for most applications. They are more compact, easier to manipulate, and have much better continuity properties. In addition, other common parametric surfaces such as spheres and cylinders can be well approximated by relatively small numbers of cubic Bézier patches.

Several algorithms were proposed by researches for direct ray tracing of parametric patches. They can be classified into two categories, Subdivision methods and Numerical methods. Recursive subdivision methods were the first to be used for ray tracing parametric surface patches. If the bounding volume of a patch is pierced by the ray, then the patch is subdivided and bounding volumes are produced for each subpatch. The subdivision process is repeated until either no bounding volumes are intersected or the intersected bounding volume is smaller than a predetermined minimum. Numerical methods finds the exact point of intersection of a ray with a patch without subdivision. It transforms finding the intersection to solving a numerical equation. They are generally robust and do not require preliminary subdivision to satisfy some a priori approximation. However, solving the numerical equation has high computational complexity that prelude their use in ”real-life” rendering systems.

Kajiya [25] uses ideas from algebraic geometry to obtain a numerical procedure for intersecting a ray with a bicubic surface patch without subdivisions. A ray represented by two planes is intersected with the patch to form two bicubic curves. Intersection of the bicubic curves gives the point at which the ray pierces the patch. It requires finding the roots of a univariate 18 degree polynomial and computing gcd of bicubic polynomials for every ray patch intersection. This method has several advantages: It is robust and guarantees to find the correct point of intersection. For patches of lower degree, it proceeds more quickly. The algorithm is simply structured, with few indeterminate loops, and thus can be implemented in hardware. No memory overhead is required. However, it needs enormous amount of floating point operations for finding all the roots of a univariate 18 degree polynomial, making it unsuitable for interactive implementation on a CPU.

1.4 Contributions of the thesis

In this thesis, we present efficient implementations of two highly compute intensive problems on the GPU using the CUDA programming framework. We implement the Singular Value Decomposition algorithm on the GPU using the twin step Golub Reinsch algorithm (Bidiagonalization followed by Diagonalization). Computing the Singular Value Decomposition requires applying transformations to a given dense matrix. The transformations map well to the GPU architecture.

We also give an algorithm for direct ray tracing of parametric surfaces on the GPU using numerical method to find the intersection of a ray with the parametric patch without subdividing. In requires finding the roots of the univariate 18 degree polynomial for each ray patch intersection. The polynomial formation and finding its roots for a ray patch intersection pair is independent of every other intersection making it an ideal candidate for parallel processing.

Although, the two problems are not related, they exhibit similar properties. We try to leverage the performance of GPUs for finding the Singular Value Decomposition and ray tracing of parametric patches, being both computationally demanding and massively parallel. We apply similar optimizations techniques while designing the parallel solutions. Both problems have high memory requirements. Our algorithms exploits the parallelism in the GPU architecture and achieves high computing performance on them. The main contributions of this thesis are the following:

- We present an efficient implementation of Singular Value Decomposition of a dense matrix on GPU using the twin step of bidiagonalization followed by diagonalization, for the first time on the GPU.
  - Our Singular Value Decomposition implementation outperforms the well known MATLAB and Intel Math Kernel Library LAPACK implementation significantly on the CPU.
  - We map the series of Householder transformations required for bidiagonalizing a matrix to the BLAS operations and implement it using the optimized CUBLAS library.
  - We use a hybrid implementation for the diagonalization step that splits the computations between the CPU and the GPU to derive maximum performance.
  - We are able to compute the Singular Value Decomposition of very large matrices, upto $14\text{K}\times14\text{K}$ which is otherwise impossible on the CPU due to memory limitations. The error on the GPU using single precision is less than 0.0001% than by using double precision.
- We adapt the ray-patch intersection approach proposed by Kajiya which computes exact hit points of a ray with a Bézier patch without subdividing it to the GPU. It facilitates ray tracing of dynamic bicubic patches by dividing the computations into independent tasks. Ours is, a recent effort to bring ray tracing of bicubic surfaces to near interactive rates with interactivity a strong possibility in the immediate future.
  - We show exact lighting, shadowing and reflections. Our approach gives highly accurate rendering in near interactive rates.
  - We demonstrate smooth silhouettes, shadows and reflections on parametric models of different complexity in interactive time.
  - The time taken to ray trace a scene is linear in the number of ray patch intersections evaluated and is independent of the number of primitives. Each intersection takes around 3.7 microseconds on GTX 280, irrespective of the number of primitives, whether primary or secondary ray tracing, etc.
  - Our ray tracing system is highly amenable to multi-GPU processing and is expected to be faster on future hardware.

Chapter 2

Background and Related Work

GPUs were originally designed to perform the highly parallel computations required for graphics rendering. But over the last couple of years, they have proven to be powerful computing workhorses across more than just graphics applications. Designed with more resources devoted to data processing rather than flow control and data caching, GPUs can be leveraged to significantly accelerate portions of codes traditionally run on CPUs, ranging from science and engineering applications to financial modeling.

2.1 GPGPU

GPGPU stands for General-Purpose computation on GPUs, also known as GPU Computing. GPUs are capable of very high computation and data throughput. Once specially designed for computer graphics and difficult to program, todays GPUs are general-purpose parallel processors with support for accessible programming interfaces and industry-standard languages such as C. Developers who port their applications to GPUs often achieve speedups of orders of magnitude vs. optimized CPU implementations.

Modern graphics architectures have become flexible as well as powerful. Once fixed function pipelines capable of outputting only 8-bit per channel color values, modern GPUs include fully programmable processing units that support vectorized floating point operations at full IEEE single precision and double precision. The current generation of GPUs include vertex texture access, full branching support in the vertex pipeline, and limited branching capability in the fragment pipeline and geometry shaders (programmable primitive assembly) bringing flexibility to an entire new stage.

As GPUs evolved, major graphics software libraries such as OpenGL and Direct3D began to exhibit enhanced ability to program these new GPUs by defining special shading functions in their API. Shaders are used to program the GPU programmable rendering pipeline, which has mostly superseded the fixed-function pipeline that allowed only common geometry transformation and pixel shading functions; with shaders, customized effects can be used.

A new concept is to use a general purpose graphics processing unit as a modified form of stream processor. This concept turns the massive floating-point computational power of a modern graphics accelerators shader pipeline into general-purpose computing power, as opposed to being hard wired solely to do graphical operations. High level languages have emerged to support the new programmability of the vertex and pixel pipelines. In certain applications requiring massive vector operations, this can yield several orders of magnitude high performance than a conventional CPU. The two largest discrete GPU designers, ATI and NVIDIA, are beginning to pursue this new approach with an array of applications. Both NVIDIA and ATI have teamed with Stanford University to create a GPU-based client for the Folding@Home distributed computing project, for protein folding calculations. In certain circumstances the GPU calculates forty times faster than the conventional CPUs traditionally used by such applications.

2.2 CUDA architecture

Recently NVIDIA began releasing cards supporting an API extension to the C programming language CUDA (Compute Unified Device Architecture), which allows specified functions from a normal C program to run on the GPU's stream processors. This makes C programs capable of taking advantage of a GPUs ability to operate on large matrices in parallel, while still making use of the CPU when appropriate. One takes a traditional C code that runs on the host (CPU) and offloads data parallel sections of the code, or kernels, to the device (GPU). CUDA is also the first API to allow CPU-based applications to access directly the resources of a GPU for more general purpose computing without the limitations of using a graphics API. It provides an access to the tremendous processing power of NVIDIA GPUs through the revolutionary new programming interface. The CUDA architecture can be programmed in industry-standard C with a few extensions. It supports a device-level application programming interface (API) and supports the upcoming OpenCL and DirectX 11 Compute standards. GPUs now act as co-processor to the CPU, off loading major part of processing.

The CUDA software stack is composed of several layers: a hardware driver, an application programming interface (API) and its runtime, and two higher-level mathematical libraries of common usage, CUFFT and CUBLAS that are both described in separate documents. The hardware has been designed to support lightweight driver and runtime layers, resulting in high performance. Figure 2.1 and 2.2 shows the CUDA hardware and software model respectively.

CUDA Hardware Model: At the hardware level the GTX 280 processor is a collection of 30 multiprocessors, with 8 processors each. Each multiprocessor has its own shared memory which is common to all the 8 processors inside it. It also has a set of 32-bit registers, texture, and constant memory caches. In any cycle, each processor of the multiprocessor executes the same instruction on different data. Communication between multiprocessors is through the device memory, which is available to all the processors of the multiprocessors.

CUDA Programming Model: For the programmer, the CUDA model is a collection of threads running in parallel. A warp is a collection of threads that can run simultaneously on a multiprocessor. The

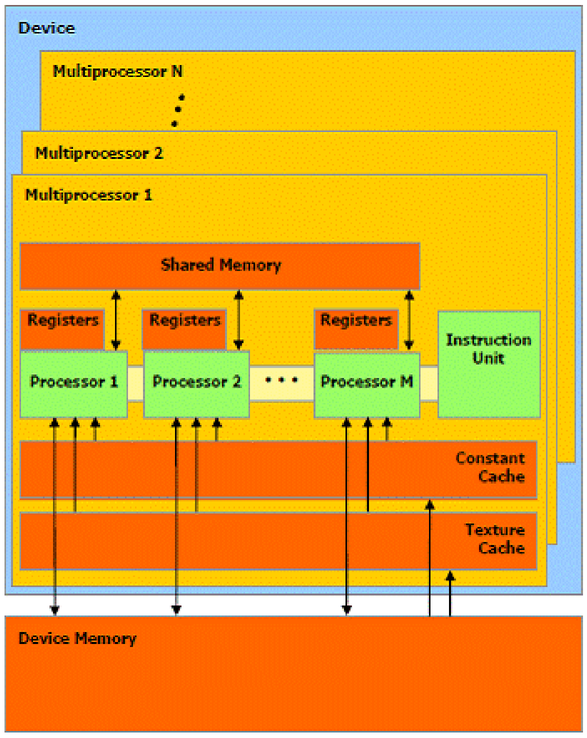


Figure 2.1 CUDA hardware model. Image obtained from the documents provided by NVIDIA.

warp size is fixed for a specific GPU, 32 on present GPUs. The programmer decides the number of threads to be executed. If the number of threads is more than the warp size, they are time-shared internally on the multiprocessor. A collection of threads (called a block) is mapped to a multiprocessor at a given time. Multiple blocks can be assigned to a multiprocessor and their execution is time-shared. A single computation on a device generates a number of blocks. A collection of all the blocks in a single computation is called a grid. All threads of the blocks mapped to a multiprocessor divide its resources equally amongst themselves. Each thread and block is given a unique ID that can be accessed within the thread during its execution. Each thread executes a single instruction set called the kernel. GPU is a co-processor to the CPU and needs to be initiated by the CPU.

CUDA has several advantages over traditional general purpose computation on GPUs (GPGPU) using graphics APIs.

- Scattered reads: code can read from arbitrary addresses in memory.
- Shared memory: CUDA exposes a fast shared memory region (16KB in size) that can be shared amongst threads. This can be used as a user-managed cache, enabling higher bandwidth than is possible using texture look ups.
- Faster downloads and read backs to and from the GPU
- Full support for integer and bit wise operations, including integer texture look ups.

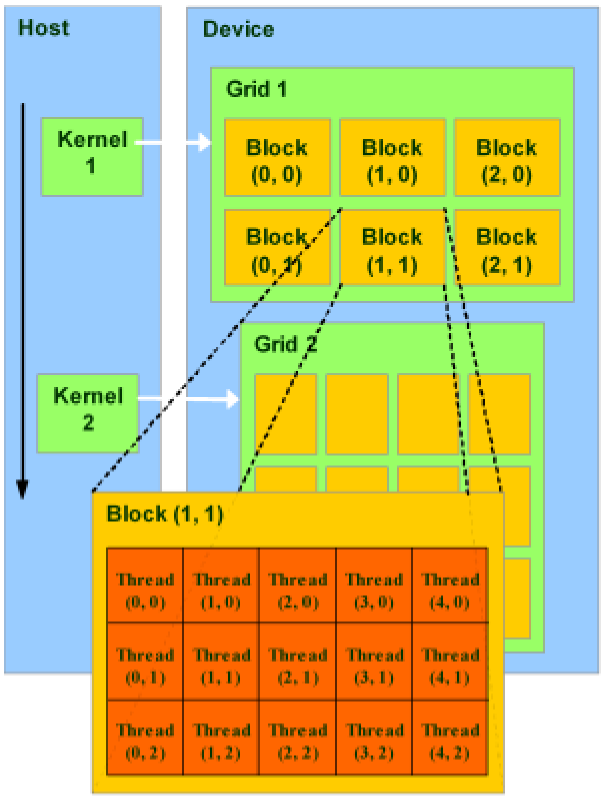


Figure 2.2 CUDA programming model. Image obtained from the documents provided by NVIDIA.

The CUDA programming model has the following limitations:

- CUDA uses a recursion-free, function-pointer-free subset of the C language, plus some simple extensions. However, a single process must run spread across multiple disjoint memory spaces, unlike other C language runtime environments.
- Texture rendering is not supported.
- For double precision (only supported in newer GPUs like GTX 260) there are some deviations from the IEEE 754 standard: round-to-nearest-even is the only supported rounding mode for reciprocal, division, and square root. In single precision, denormals and signalling NaNs are not supported; only two IEEE rounding modes are supported (chop and round-to-nearest even), and those are specified on a per-instruction basis rather than in a control word; and the precision of division/square root is slightly lower than single precision.
- The bus bandwidth and latency between the CPU and the GPU may be a bottleneck.
- Threads should be running in groups of at least 32 for best performance, with total number of threads numbering in the thousands. Branches in the program code do not impact performance significantly, provided that each of 32 threads takes the same execution path; the SIMD execution

model becomes a significant limitation for any inherently divergent task (e.g. traversing a space partitioning data structure during raytracing).

- Unlike OpenCL, CUDA-enabled GPUs are only available from NVIDIA (GeForce 8 series and above, Quadro and Tesla).

2.3 Singular Value Decomposition

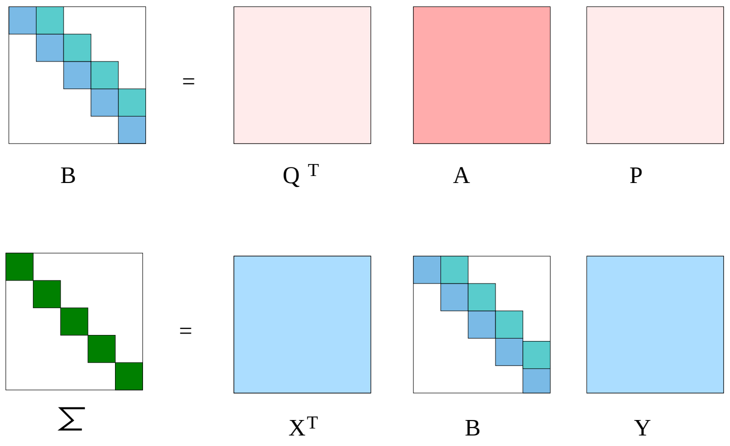


Figure 2.3 Subdivision of a matrix into blocks of size L

Singular Value Decomposition of a dense matrix is computed using the two step Golub Reinsch algorithm as shown in Figure 2.3. It decomposes the matrix A into component orthogonal matrices U, V and a diagonal matrix Σ. In the first step, dense matrix A of size $m \times n$ is reduced to a bidiagonal matrix B by applying n Householder transformations. Each transformation reduces a row-column pair. The asymptotic time of which is order of 3. Inverse transformations are applied on the orthogonal matrices Q and P which are initially identity.

The matrix B is then reduced to a diagonal matrix Σ by applying the iterative implicitly shifted QR algorithm. Every iteration of the QR algorithm applies Givens rotations on the elements of B and the corresponding inverse rotations are applied on the rows of the orthogonal matrices X^T and Y^T which are initially identity. Each rotation reduces the superdiagonal elements of B. The iterations converge when all the superdiagonal elements of B converge to zero. The final orthogonal matrices of the Singular Value Decomposition are computed as $U = QX$ and $V^T = (PY)^T$.

2.3.1 Related Work

Several algorithms have been developed on the GPUs for mathematical computations like sorting [20], geometric computations, matrix multiplications, FFT [34] and graph algorithms [23, 48]. Kruger and Westermann [27] introduced a framework for the implementation of linear algebra operators on vectors and matrices that exploits the parallelism on GPUs. Galoppo *et al.* [17] reduced the matrix decomposition and row operations to a series of rasterization problems on the GPU. Christen *et al.* [10] proposed algorithms to accelerate sparse direct factorization and non-linear interior point optimization on the GPU using CUDA. Barrachina *et al.* [3] proposed two high-level application programming interfaces that use the GPU as a co-processor for dense linear algebra operations. There have been many efforts towards optimizing and tuning of the Level 3 CUBLAS Graphics Processors. Barrachina *et al.* [2] proposed several alternative implementations that are competitive with those in CUBLAS. Fujimoto [15] proposed a new algorithm for matrix-vector multiplication on NVIDIA CUDA architecture. Barrachina *et al.* [4] presented several algorithms to compute the solution of a linear system of equations. Fatica and Jeong [14] proposed how MATLAB could be extended to take advantage of the computational power offered by the latest GPUs. NVIDIA's CUDA library [36] comes with an implementation of simple Basic Linear Algebra Subprograms (BLAS) on GPU, called the CUBLAS.

There have been many efforts towards parallelizing the Singular Value Decomposition algorithm on architectures like the FPGA, Cell Processors, GPU, etc., which have scalable parallel architecture. Ma *et al.* [32] proposed the implementation of two-sided rotation Jacobi Singular Value Decomposition algorithm on a two million gate FPGA. They proposed a mesh connected array structure of simple 2×2 processors to compute Singular Value Decomposition of large matrix. Bobda and Steenbock [6] proposed an efficient implementation of the Singular Value Decomposition for large matrices and the possibility of integrating FPGA's as a part of a Distributed Reconfigurable System (DRS). Baker [1] described a parallel algorithm to compute the Singular Value Decomposition of block circulant matrices on the Cray-2. Dickson *et al.* [12] designed a programmable processor for computing the Givens rotation using approximate rotation method. The processor can also be programmed for Singular Value Decomposition computation. Yamamoto *et al.* [53] proposed a method to speed up the Singular Value Decomposition of very large rectangular matrices using the CSX600 floating point co-processor. They achieve up to 3.5 times speedup over the Intel MKL on 3.2GHz Xeon processor for a 100000×4000 matrix but was not efficient on smaller matrices. Zhang Shu [42] presented the implementation of One Sided Jacobi method for Singular Value Decomposition on GPU using CUDA. The performance of their algorithm is limited by the availability of shared memory and works well only for small size matrices. Bondhugula *et al.* [7] proposed a hybrid GPU based implementation of Singular Value Decomposition using fragment shaders and frame buffer objects in which the diagonalization would be performed on the CPU.

There are several numerical libraries such as ATLAS and the Intel Math Kernel Library which are widely used for different applications on the CPU. They are designed to achieve high accuracy as well as high memory bandwidth and computational throughput on the CPUs, e.g. Intel MKL is optimized for Intel processors. Intel MKL LAPACK gives better performance than standard LAPACK on Intel processors.

2.4 Ray Tracing Parametric Patches

Kajiya's ray patch intersection algorithm is briefly described in this section. It uses algebraic geometry approach to find a ray patch intersection point without subdivision. The procedure guarantees to find the exact point of intersection. A ray is represented as an intersection of two planes (l^0, l^1) as shown in Figure 2.4. These planes intersect the bicubic patch in two bicubic curves a and b. Intersection of these bicubic curves gives the point at which the ray pierces the patch. Finding the point of intersection of two bicubic curves requires solving a univariate 18 degree polynomial. The real roots of the polynomial give the points of intersection of the bicubic curves, hence the point of intersection of the ray. Polynomial roots gives the v parameter of the point of intersection. The u parameter is found by finding the GCD of a and b on substituting v parameter value. A ray can intersect a patch in 18 points (both real and complex). Per pixel normals can be found by evaluating the partial derivatives of the patch equation.

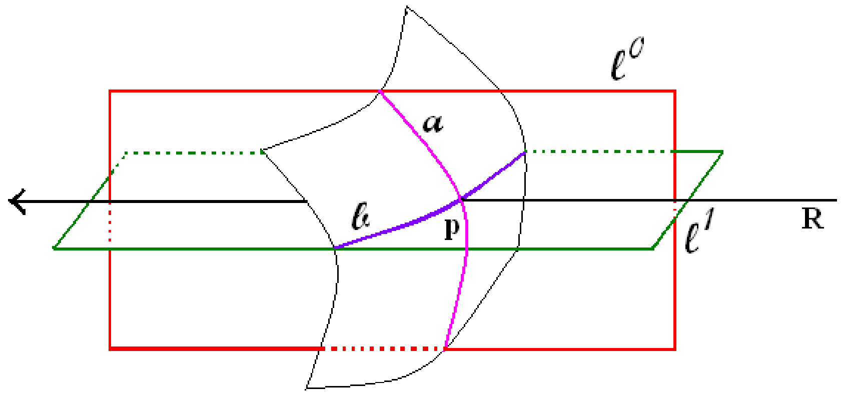


Figure 2.4 Shows the intersection of a ray with a Bézier patch. A ray is represented as an intersection of two planes (l^0, l^1). Planes l^0 and l^1 intersect a Bézier patch in two bicubic curves a and b respectively. The intersection of the bicubic curves gives the point of intersection of the ray with the patch.

The roots of the polynomial are found using the numerically stable Laguerre's method. It is an efficient algorithm for numerically computing the roots of a polynomial and guaranteed to converge to a root irrespective of the initial guess. It is cubically convergent tripling the number of digits of accuracy at every step. Finding all the roots of the polynomial guarantees finding the correct point of intersection of ray with patch giving better rendering and shading for the parametric models.

2.4.1 Related Work

Two broad approaches have been used to render bicubic patches in high quality. The first approach computes the exact hit point numerically. The second refines the surface using subdivision usually till an error criterion is met, followed by rendering using polygons.

Exact ray tracing of bicubic surfaces was proposed by Kajiya [25]. His method represents a ray using two planes, which are intersected with the patches. The resulting pair of bicubic curve equations result in a univariate 18-degree polynomial equation, whose solution yields the v parameter of the intersection, with the u parameter obtained using a GCD computation. Kajiya proposed Laguerre's method for polynomial roots. Toth solved the ray surface intersection directly using multivariate Newton iteration [47]. Utilizing results from interval analysis, Toth proposes a method for identifying regions of parameter space in which the Newton iteration is guaranteed to converge to a unique solution. Toth's method is robust, dealing correctly with all possible cases. His method still consumes considerable amounts of computing time. Joy and Bhetanabhotla [24] advocate the use of quasi-Newton methods for finding local minima of a function representing the squared distance of a ray from points on a parametric surface. However, naive utilization of ray coherence may cause convergence to incorrect solutions. Manocha and Krishnan found the polynomial roots using eigenvalues of its characteristic matrix [33]. Iteratively solving the ray-patch intersection in the ray space or parameter space has also been attempted using the Newton's method [18] or after restricting the intersections using Bézier Clipping [35, 49]. Lewis *et al.* proposed a hardware based design of a pipelined architecture for ray Bézier patch intersection [29]. These methods are computationally very expensive for interactive rendering.

In recursive subdivision, the patch is subdivided if the bounding volume of a patch is pierced by the ray. The procedure is then repeated until either no bounding volumes are intersected or the ray intersects the bounded subpatch. The method was first described by Whitted [50], who uses spheres as bounding volumes. Rubin and Whitted [40] use basically the same method with bounding boxes instead of spheres. An approach based on recursive subdivision was presented by Woodward [52]. The subdivision process is carried out in the orthogonal viewing coordinate system of the ray. Benthin *et al.* [5] subdivides the patch in the parameter space followed by triangulation for fast rendering of bicubic patches. They achieved near interactive frame rates on a PC.

Real-time rendering using view-dependent, GPU-assisted tessellation was achieved recently. Patney and Owens use a REYES-style adaptive surface subdivision exploiting the data-parallelism of the GPU [37]. Eisenacher *et al.* use a view dependent adaptive subdivision that bounds the screen space projection error [13]. Their method assigns more quads to curved areas and fewer to flatter parts and achieve over 70 fps on models like the Killeroo on a top-line GPU. These methods are the fastest available today, but still approximate the surface using triangles, making exact lighting and reflections not so easy to produce.

Exact hit points can be computed by adapting Kajiya's approach to the GPU in interactive time. It will provide exact lighting, shadowing, and reflections. Ray tracing of algebraic [31], implicit [43], and triangulated [39, 26, 28] surfaces on the GPU has been reported. The GPU specific optimizations they used are relevant to our work also.

2.5 GPU Applications

In the late 1990's computer scientists and researchers began to use GPUs for running general purpose computational tasks. It let to a huge performance boost for a wide range of scientific applications. Architectural advances and new software tools have made GPU hardware more amenable to general purpose programming. As a result, GPU's are now used to accelerate computationally intensive applications. The GPUs ability to operate on large data sets in parallel makes it more effective than a general-purpose CPU for a range of complex image processing, computational finance, and other applications. With the introduction of CUDA architecture, GPGPU problems are now addressed with a much simpler API for GPU.

In recent years, there has been a renewed interest in real-time ray tracing for interactive applications. Ray Tracing algorithms can be highly parallelized on shared memory and distributed memory systems. Lauterbach *et al.* [28] presented two novel parallel algorithms for rapidly constructing bounding volume hierarchies on many-core GPUs for fast ray tracing. Loop and Blinn [31] proposes a parallel algorithm on GPU for real-time rendering of algebraic surfaces defined by Bézier tetrahedra. Singh and Narayanan [43] presented a ray-tracing procedure to render general implicit surfaces efficiently on the GPU. Guntury and Narayanan [22] present a novel approach to ray tracing by decomposing the algorithm into several data-parallel stages that are mapped efficiently to GPU. In addition to graphics rendering, GPUs are used in game physics calculations; examples include Physx [38]. CUDA has also been used to accelerate non-graphical applications in computational biology, cryptography and other fields by an order of magnitude or more. Several algorithms have been developed for sorting, geometric computations, matrix multiplication, Fast Fourier Transform and graph algorithms.

Medical imaging is one of the earliest applications to take advantage of GPU computing to get acceleration. Liang [30] proposed a new hardware-based accelerated algorithm for volume rendering of CT scans on GPU for rapid visualization. Stone *et al.* [44] proposed GPU accelerated advanced magnetic resonance imaging reconstruction algorithm, thereby improving the quality of MR images across a broad spectrum of applications. Several researches have developed GPU implementations of many computer vision algorithms. OpenVIDIA: Parallel GPU Computer Vision introduced by Fung and Mann [16] utilizes the computational power of the GPU to provide real-time computer vision and imaging (Canny edge detection, filtering, matching features, image registration). Yang and Pollefeys [54] used GPUs for real-time stereo depth extraction from multiple images. Fast gain adaptive KLT tracking on the GPU was proposed by Zach *et al.* [55].

Recently, GPUs have been used to accelerate a variety of numerical algorithms. Tomov and Dongarra [46] presented a Hessenberg reduction algorithm for hybrid multicore and GPU systems that gets more than 16 performance improvement over the current LAPACK algorithm. Cevahir *et al.* [8] propose a scalable implementation of a Conjugate Gradient solver for unstructured matrices on a GPU-extended cluster. A well known data parallel primitives library CUDPP was developed by Sengupta *et al.* [41]. It implements data-parallel algorithm primitives such as parallel prefix sum, parallel sort and parallel reduction.

Chapter 3

Singular Value Decomposition

Singular Value Decomposition of a dense matrix factorizes it into component matrices. Section 2.3 gives the overview of the two Golub Reinsch Singular Value Decomposition algorithm. The algorithm and its implementation on the GPU is described in detail in the following sections.

3.1 Singular Value Decomposition Algorithm

In the first step, the matrix is reduced to a bidiagonal matrix. The operation is performed by applying a series of Householder transformations on the original matrix. These transformations can be mapped well to the BLAS operations on the GPU. In the second step, the bidiagonal matrix is reduced to a diagonal matrix by applying implicitly shifted QR iterations. Each iteration reduces the elements of the bidiagonal matrix close to a diagonal matrix. After sufficient iterations are reached, the bidiagonal matrix is reduced to diagonal matrix. We describe the bidiagonalization and diagonalization procedure in detail in the following sections.

3.1.1 Bidiagonalization

In this step, the given matrix A is decomposed as

$$A = QBP^T \tag{3.1}$$

by applying a series of householder transformations where B is a bidiagonal matrix and Q and P are unitary householder matrices. For a matrix of size $m \times n$ with $m \geq n$, we first select a householder vector $\mathbf{u}^{(1)}$ of length m for vector $A(1:m,1)$ and $\mathbf{v}^{(1)}$ of length n for $A(1,2:n)$ such that

$$\begin{aligned} \hat{A}_1 &= (I - \sigma_{1,1}\mathbf{u}^{(1)}\mathbf{u}^{(1)^T})A(I - \sigma_{2,1}\mathbf{v}^{(1)}\mathbf{v}^{(1)^T}) \qquad (3.2) \\ &= H_1AG_1 = \begin{bmatrix} \alpha_1 & \beta_1 & 0 & \dots & 0 \\ 0 & x & x & \dots & x \\ \vdots & \vdots & & & \vdots \\ 0 & x & \dots & & x \end{bmatrix}. \end{aligned}$$

$\hat{A}_1$ has zeros below the diagonal and to the right of the superdiagonal of the first row and $A(1,1)$ is updated to α_1 and $A(1,2)$ is updated to β_1. This is the first column-row elimination.

We denote the left householder $m \times m$ matrices as H_i's and right householder $n \times n$ matrices as G_i's and the corresponding σ's as $\sigma_{1,i}$'s and $\sigma_{2,i}$'s respectively. The elimination procedure is then repeated for second column $A(2:m,2)$ and row $A(2,3:n)$ and so on. If $m > n$, n columns and $n-2$ rows must be eliminated. After all the columns and rows are eliminated we obtain a final bidiagonal matrix B such that

$$B = Q^T A P, \tag{3.3}$$

where

$$Q^T = \prod_{i=1}^{n} H_i, P = \prod_{i=1}^{n-2} G_i. \tag{3.4}$$

Here, $H_i = I - \sigma_{1,i}\mathbf{u}^{(i)}\mathbf{u}^{(i)^T}$ and $G_i = I - \sigma_{2,i}\mathbf{v}^{(i)}\mathbf{v}^{(i)^T}$. The $\mathbf{u}^{(i)}$'s are vectors of length m with $i-1$ leading zeros and $\mathbf{v}^{(i)}$'s are vectors of length n with i leading zeros. These are formed as $\mathbf{u}^{(i)} = [0 \ldots 0, \hat{\mathbf{u}}^{(i)}]^T$ and $\mathbf{v}^{(i)} = [0 \ldots 0, \hat{\mathbf{v}}^{(i)}]^T$ where $\hat{\mathbf{u}}^{(i)}$ is a vector of $m-i+1$ trailing components of $\mathbf{u}^{(i)}$ and $\hat{\mathbf{v}}^{(i)}$ is a vector of $n-i$ trailing components of $\mathbf{v}^{(i)}$. In general, for a vector $\mathbf{y} = [y_1, \ldots, y_l]$ of length l the selection of householder vector $\mathbf{r}$ and scalars σ and α as given below

$$\begin{aligned} \sigma &= (y_1 + a)/a && (3.5) \\ \text{and} \quad \mathbf{r} &= \frac{\mathbf{y} + [a, 0, \ldots, 0]^T}{y_1 + a} && (3.6) \end{aligned}$$

satisfies $(I - \sigma\mathbf{r}\mathbf{r}^T)\mathbf{y} = [\alpha, 0, \ldots, 0]^T$. $\hat{\mathbf{u}}^{(i)}$ of length $m-(i-1)$ and α_i for $A(i:m,i)$ and $\hat{\mathbf{v}}^{(i)}$ of length $n-i$ and β_i for $A(i,i+1:n)$ are computed similar to $\mathbf{r}$ and α in Equation 3.5 to 3.6.

The householder bidiagonalization can be achieved by alternating matrix vector multiplies with rank-one updates introduced by Golub and Kahan [19]. The multiplication of A matrix by H_i updates $A(i:m,i+1:n)$ and $A(i,i)$ and multiplication by G_i updates $A(i+1:m,i+1:n)$ and $A(i,i+1)$. We can summarize the update of the trailing matrix A after i^{th} column-row elimination as two rank updates as

$$\begin{aligned} A(i+1:m,i+1:n) = A(i+1:m,i+1:n) \\ -\hat{\mathbf{u}}^{(i)}\hat{\mathbf{z}}^{(i)^T} - \hat{\mathbf{w}}^{(i)}\hat{\mathbf{v}}^{(i)^T} \end{aligned}$$

where

$$\begin{aligned} \hat{\mathbf{z}}^{(i)^T} &= \mathbf{x}^T - \sigma_{2,i}(\mathbf{x}^T\hat{\mathbf{v}}^{(i)})\hat{\mathbf{v}}^{(i)^T}, \\ \hat{\mathbf{w}}^{(i)} &= \sigma_{2,i}A(i:m,i+1:n)\hat{\mathbf{v}}^{(i)} \\ \text{and} \quad \mathbf{x} &= \sigma_{1,i}A^T(i:m,i+1:n)\hat{\mathbf{u}}^{(i)}. \end{aligned}$$

The householder matrices Q and P given in Equation 3.4 are computed similarly as it also involves multiplication by H_i's and G_i's respectively, but in reverse order. The update rule after i^{th} column-row elimination is

$$\begin{aligned} Q(1:m,i:m) &= Q(1:m,i:m) - \hat{\mathbf{k}}^{(i)}\hat{\mathbf{u}}^{(i)T} \quad \text{and} \\ P^T(i:n,1:n) &= P^T(i:n,1:n) - \hat{\mathbf{v}}^{(i)}\hat{\mathbf{l}}^{(i)^T}, \quad \text{where} \\ \hat{\mathbf{k}}^{(i)} &= \sigma_{1,i}Q(1:m,i:m)\hat{\mathbf{u}}^{(i)} \quad \text{and} \\ \hat{\mathbf{l}}^{(i)} &= \sigma_{2,i}P^T(i:n,1:n)^T\hat{\mathbf{v}}^{(i)}. \end{aligned}$$

The updates can be expressed using BLAS level 2 operations. After every column-row elimination, the trailing matrix is updated. This method is computationally expensive and involves many reads and writes to the memory after each elimination. We can increase the computation to read ratio by deferring the update of the trailing matrix, by bidiagonalizing a block of columns and rows together and updating the trailing matrix as proposed in [9]. The LAPACK implementation also uses the blocking approach. This requires the computation of new rows and columns belonging to the block just before elimination due to the previous eliminations in the block.

The matrix A is divided into blocks of size L as shown in Figure 3.1 and the update occurs only after L columns and rows are bidiagonalized. Extra computations are performed for the updated columns and rows of the same block, which require $\hat{\mathbf{u}}^{(i)}$'s, $\hat{\mathbf{v}}^{(i)}$'s, $\hat{\mathbf{w}}^{(i)}$'s and $\hat{\mathbf{z}}^{(i)}$'s due to previous eliminations in the block. These vectors are also needed for updating the trailing matrix once L columns and rows are eliminated. As the set of update vectors is incremented with every elimination, more computations are required to update the columns and the rows before elimination. The householder matrices Q and P^T are also block updated for which $\hat{\mathbf{k}}^{(i)}$'s and $\hat{\mathbf{l}}^{(i)}$'s are stored. The value of L is chosen depending on the performance of the BLAS routines. The algorithm has a total floating point operation count of $O(mn^2)$ for $m \geq n$.

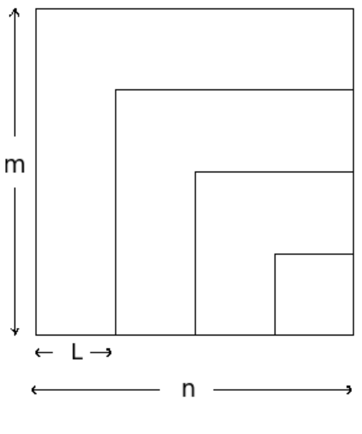


Figure 3.1 Subdivision of a matrix into blocks of size L

This method requires an additional storage of a $m \times L$ array U_{mat} to store $\hat{\mathbf{u}}^{(i)}$'s, a $L \times n$ array V_{mat} to store $\hat{\mathbf{v}}^{(i)}$'s, a $m \times L$ array W_{mat} to store $\hat{\mathbf{w}}^{(i)}$'s, a $L \times n$ array Z_{mat} to store $\hat{\mathbf{z}}^{(i)}$'s, a $m \times L$ array Q_{mat} to store $\hat{\mathbf{k}}^{(i)}$'s and a $L \times n$ array P_{mat} to store $\hat{\mathbf{l}}^{(i)}$'s. Since these are required only for updating the matrix, they can be reused after every block update. The trailing matrix is accessed only during the matrix update.

```
Require: $m \geq n$
1: $kMax \leftarrow \frac{n}{L}$ {$L$ is the block size}
2: for $i = 1\ to\ kMax$ do
3:     $t \leftarrow L(i-1)+1$
4:     Compute $\hat{\mathbf{u}}^{(t)}$, $\alpha_{1,t}, \sigma_{1,t}, \hat{\mathbf{k}}^{(t)}$                                CUBLAS
5:     Eliminate $A(t:m,t)$ and update $Q(1:m,t)$                          CUBLAS
6:     Compute new $A(t,t+1:n)$                                          CUBLAS
7:     Compute $\hat{\mathbf{v}}^{(t)}$, $\alpha_{2,t}, \sigma_{2,t}, \hat{\mathbf{l}}^{(t)}$                                CUBLAS
8:     Eliminate $A(t,t+1:n)$ and update $P^T(t,1:n)$                      CUBLAS
9:     Compute $\hat{\mathbf{w}}^{(t)}$, $\hat{\mathbf{z}}^{(t)}$ and store the vectors                       CUBLAS
10:    for $k = 2\ to\ L$ do
11:        $t \leftarrow L(i-1)+k$
12:        Compute new $A(t:m,t)$ using $k-1$ update vectors              CUBLAS
13:        Compute $\hat{\mathbf{u}}^{(t)}$, $\alpha_{1,t}, \sigma_{1,t}, \hat{\mathbf{k}}^{(t)}$                            CUBLAS
14:        Eliminate $A(t:m,t)$ and update $Q(1:m,t)$                      CUBLAS
15:        Compute new $A(t,t+1:n)$                                      CUBLAS
16:        Compute $\hat{\mathbf{v}}^{(t)}$, $\alpha_{2,t}, \sigma_{2,t}, \hat{\mathbf{l}}^{(t)}$                            CUBLAS
17:        Eliminate $A(t,t+1:n)$ and update $P^T(t,1:n)$                  CUBLAS
18:        Compute $\hat{\mathbf{w}}^{(t)}$, $\hat{\mathbf{z}}^{(t)}$ and store the vectors                   CUBLAS
19:    end for
20:    Update $A(iL+1:m, iL+1:n)$, $Q(1:m, iL+1:m)$ and $P^T(iL+1:n, 1:n)$    CUBLAS
21: end for
```

Algorithm 2: Bidiagonalization algorithm. All the steps are executed using CUBLAS Matrix-Vector and Matrix-Matrix multiplication routines.

3.1.2 Diagonalization

The bidiagonal matrix can be reduced to a diagonal matrix by iteratively applying the implicitly shifted QR algorithm [11]. The matrix B obtained in the first step is decomposed as

$$\Sigma = X^T B Y \tag{3.7}$$

where Σ is a diagonal matrix, X and Y are orthogonal unitary matrices.

Algorithm 3 describes the diagonalization procedure. The $d(i)'s$ are the diagonal elements and $e(i)'s$ are the superdiagonal elements of the matrix B. Every iteration updates the diagonal and the superdiagonal elements such that the value of the superdiagonal elements becomes less than their previous value. On convergence of the algorithm, $d(i)'s$ contains the singular values and X and Y^T contains the singular vectors of B.

```
1: iter ← 0, maxitr ← $12 * N * N$ {N is the number of main diagonal elements}, $k_2 \leftarrow N$
2: for $i = 1$ to maxitr do
3:     if $k_2 <= 1$ then
4:         break the loop
5:     end if
6:     if iter > maxitr then
7:         return false
8:     end if
9:     matrixsplitflag ← false
10:    for $l = 1$ to $k_2 - 1$ do
11:        $k_1 \leftarrow k_2 - l$ {Find diagonal block matrix to work on}
12:        if abs($e(k_1)$) <= thres then
13:            matrixsplitflag ← true, break the loop
14:        end if
15:    end for
16:    if !matrixsplitflag then
17:        $k_1 \leftarrow 1$
18:    else
19:        $e(k_1) \leftarrow 0$
20:        if $k_1 == k_2 - 1$ then
21:            $k_2 \leftarrow k_2 - 1$, continue with next iteration
22:        end if
23:    end if
24:    $k_1 = k_1 + 1$
25:    if $k_1 == k_2 - 1$ then
26:        Compute SVD of $2 \times 2$ block and coefficient vectors $\mathbf{C}_1$, $\mathbf{S}_1$ and $\mathbf{C}_2$, $\mathbf{S}_2$ of length 1
27:        Apply forward row transformation on the rows $k_2 - 1$ and $k_2$ of $Y^T$ using $\mathbf{C}_1$, $\mathbf{S}_1$          GPU kernel
28:        Apply forward column transformation on the columns $k_2 - 1$ and $k_2$ of $X$ using $\mathbf{C}_2$, $\mathbf{S}_2$   GPU kernel
29:        $k_2 \leftarrow k_2 - 2$, continue with next iteration
30:    end if
31:    Select shift direction: forward if $d(k_1) < d(k_2)$, else backward, Apply convergence test on the sub block
32:    Compute the $shift$ from 2-by-2 block at the end of the sub matrix, iter ← iter+$k_2 - k_1$
33:    Apply simplified/shifted forward/backward Givens rotation on the rows $k_1$ to $k_2$ of $B$ and compute $\mathbf{C}_1$,
       $\mathbf{S}_1$ and $\mathbf{C}_2$, $\mathbf{S}_2$ of length $k_2 - k_1$
34:    Apply forward/backward transformation on the rows $k_1$ to $k_2$ of $Y^T$ using $\mathbf{C}_1$, $\mathbf{S}_1$          GPU kernel
35:    Apply forward/backward transformation on the columns $k_1$ to $k_2$ of $X$ using $\mathbf{C}_2$, $\mathbf{S}_2$         GPU kernel
36: end for
37: Sort the singular values and corresponding singular vectors in decreasing order.                 GPU kernel
```

Algorithm 3: Diagonalization algorithm. Steps 27-28, 34-35 and 37 are executed on the GPU.

The algorithm finds indexes k_1 and k_2 with $k_1 < k_2$ in each iteration such that $e(k_1)$ is below a threshold which depends on the machine precision. If k_1 and k_2 differ by 1 or 2, one or two singular values can be extracted directly and k_2 moves up. Otherwise, a series of Givens rotations modify $d(i)$ and $e(i)$ in the range k_1 to k_2 such that $e(i)$'s become smaller than before. Each rotation is captured in the coefficient vectors ($\mathbf{C}_1$,$\mathbf{S}_1$) and ($\mathbf{C}_2$,$\mathbf{S}_2$). Corresponding inverse rotations are applied on X and Y^T matrix using the coefficient vectors. Algorithm 4 and 5 describes the rotations applied on the rows of Y^T in the forward and backward direction respectively. Similar rotations are applied on the columns of

X using $\mathbf{C}_2$ and $\mathbf{S}_2$. See [11] for more details on the steps. The computation converges when all the singular values are found.

```
Require: $k_1 < k_2$
1: for $j$=$k_1$ to $k_2 - 1$ do
2:     $\mathbf{t} \leftarrow Y^T(j+1, 1:n)\mathbf{C}_1(j - k_1 + 1)$
3:     $\mathbf{t} \leftarrow \mathbf{t} - Y^T(j, 1:n)\mathbf{S}_1(j - k_1 + 1)$
4:     $Y^T(j, 1:n) \leftarrow Y^T(j, 1:n)\mathbf{C}_1(j - k_1 + 1) + Y^T(j+1, 1:n)\mathbf{S}_1(j - k_1 + 1)$
5:     $Y^T(j+1, 1:n) \leftarrow \mathbf{t}$
6: end for
```

Algorithm 4: Forward transformation on the rows of Y^T

```
Require: $k_1 < k_2$
1: for $j$=$k_2 - 1$ to $k_1$ do
2:     $\mathbf{t} \leftarrow Y^T(j+1, 1:n)\mathbf{C}_1(j - k_1 + 1)$
3:     $\mathbf{t} \leftarrow \mathbf{t} - Y^T(j, 1:n)\mathbf{S}_1(j - k_1 + 1)$
4:     $Y^T(j, 1:n) \leftarrow Y^T(j, 1:n)\mathbf{C}_1(j - k_1 + 1) + Y^T(j+1, 1:n)\mathbf{S}_1(j - k_1 + 1)$
5:     $Y^T(j+1, 1:n) \leftarrow \mathbf{t}$
6: end for
```

Algorithm 5: Backward transformation on the rows of Y^T

3.2 Singular Value Decomposition on GPU

3.2.1 Bidiagonalization on GPU

Algorithm 2 describes the bidiagonalization procedure. Each step can be performed using CUDA BLAS functions. CUBLAS [36] provides high performance matrix-vector, matrix-matrix multiplications and norm computation function. The blocking approach for bidiagonalization can be performed efficiently since CUBLAS gives high performance for matrix-vector, matrix-matrix multiplications even if one of the dimensions is small. Experiments prove that CUBLAS deliver much higher performance when operating on matrices with dimensions that are a multiple of 32 due to memory alignment issues [2]. Hence, we pad the vectors and matrices with zeros, transforming their dimensions to the next multiple of 32.

The performance of the GPU libraries depend on data placement and how the library is used. The movement of data is of consideration when using BLAS in general. We assume that initially the input matrix A is on the CPU and is transferred to the GPU. NVIDIA 8800 GTX has a sustained internal memory bandwidth of 86.4 GB/s but the bandwidth between the CPU and the GPU is an order of magnitude lower. Hence CPU to GPU transfers should be minimized. The matrices Q, P^T, U_{mat}, V_{mat}, W_{mat} and Z_{mat} are initialized on the device. All the operations required for bidiagonalization are done on the data local to the GPU using CUBLAS library routines. Since the thread-processors of the GPU operate on GPU data, there is no expensive data transfer between the GPU and the CPU. The bidiagonalization is performed inplace, i.e., A becomes the bidiagonal matrix. After the bidiagonalization of matrix A on the GPU, the diagonal and superdiagonal elements of the bidiagonal matrix are copied to the CPU to proceed with the diagonalization as described in the next section while matrices Q and P^T reside on the device memory. The sequential bidiagonalization algorithm has a complexity of $O(mn^2)$ for $m \geq n$. Our use of the latest CUBLAS 2.0 library keeps the GPU implementation very efficient on the given hardware. The total storage requirement for the algorithm is $(3(mL + Ln) + m^2 + n^2 + mn + 2\max(m, n)) \times 4$ bytes on the GPU.

3.2.2 Diagonalization on GPU

In this section, we present the parallel version of the diagonalization algorithm and its implementation on the GPU. The diagonal and superdiagonal elements of B are copied to the CPU. Applying Givens rotations on B and computing the coefficient vectors can be done sequentially on the CPU as it only requires access to the diagonal and superdiagonal elements. In Algorithm 4, the computations for every row of Y^T depends only on the next row, i.e., every element of a row depends only on the element below it and its corresponding coefficient vector element. In Algorithm 5, the computations depends only on the row above it. Similarly for the columns of X. The computations for each row depends on the results from the previous row, making it difficult to parallelize across rows. However, the results for all the elements of the row can be computed in parallel. We use the thread processors of the GPU to process elements of each row in parallel. This gives high performance on large matrices but also works well for medium sized matrices. The transformation of the matrices Y^T and X would be done in parallel on the GPU. In Algorithm 3, steps 27-28, 34-35 and 37 are executed on the GPU. A simple swap kernel is called for sorting the vectors. The matrices Y^T and X reside on the device memory and are initialized to identity.

Our algorithm divides a row of the matrix into blocks as shown in Figure 3.2. Each thread operates on one element of the row. Since the transformations are applied on $k_2 - k_1 + 1$ rows, the kernel runs a loop of $k_2 - k_1$ similar to Algorithm 4 with each modifying two rows. This division of the row into blocks and looping can be done efficiently on CUDA architecture, since each thread performs independent computations. The data required for the computations in the block is stored in shared memory and the computations are performed efficiently on a multiprocessor.

The coefficient vectors $\mathbf{C}_1$ and $\mathbf{S}_1$ are copied from the CPU to the device memory. At any instant during the kernel execution, an element of the coefficient vector is required by all elements of two rows. Hence, we use the shared memory to store the vectors.

We allocate 64 to 256 threads for a block depending on the size of the matrix. This ensures that there are enough blocks and all the multiprocessors are alloted atleast 2 blocks. When the thread block contains T threads, it must use the shared memory of at most 2KB to keep 8 blocks active on a multiprocessor which will give optimal performance. At any instant, a block will require $2 \times (T \times 4)$ bytes of

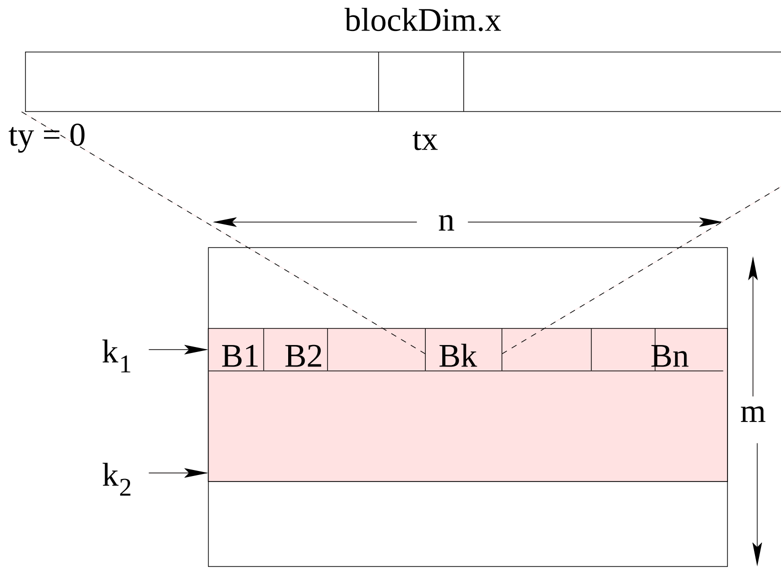


Figure 3.2 Division of a matrix row into CUDA thread blocks

shared memory for the T elements of the two rows of the matrix it is working on as we use floating point arithmetic. Every iteration of the loop in the forward kernel modifies two rows of the matrix. Since the second row is again modified in the next iteration only the first updated row is copied back to the device. The second updated row remains in the shared memory for the next iteration. The shared memory is reused for copying the third row for the next iteration and the iteration proceeds.

Hence, the amount of shared memory that could be used to store coefficient vectors is $2K-(2\times T\times 4)$ bytes. However, the memory required for the coefficient vectors is $2 \times (k_2 - k_1) \times 4$ bytes which can exceed $2K - (2 \times T \times 4)$ bytes for large matrices. In order to only use the available shared memory for the coefficient vectors we copy a fixed number of coefficient vector elements of $\mathbf{C}_1$ and $\mathbf{S}_1$ into $2K - (2 \times T \times 4)$ bytes, process the same number of rows and then reuse the shared memory for copying the next set of vector elements. The backward row transformation kernel is similar to the forward row transformation kernel.

Since the column transformations are similar to row transformations, we use the row transformation kernel on the rows of X^T instead of the columns of X. This requires copy of $\mathbf{C}_2$ and $\mathbf{S}_2$ to the GPU. As the elements are accessed sequentially there are no noncoalesced memory accesses. The access to the shared memory has no bank conflicts. All the threads in a block are used for copying the vectors from the global memory to the shared memory which requires that the vectors are padded to the nearest multiple of block size.

On convergence, $d(i)$'s contain the singular values. Y^T and X^T reside in the device memory which will be further used for computing the orthogonal matrices U and V. Our algorithm is efficient as it performs exactly the same number of operations on the GPU as the corresponding sequential algorithm. The total storage requirement for the algorithm is $(6\min(m,n))\times 4$ bytes on the CPU and $(m^2+n^2)\times 4$ bytes on the GPU.

3.2.3 Complete Singular Value Decomposition

We perform two matrix-matrix multiplications at the end to compute orthogonal matrices $U = QX$ and $V^T = (PY)^T$ as given in Equation 1.1. We use CUBLAS matrix multiplication routines. The matrices Q, P^T, X^T, Y^T, U and V^T are on the device. The orthogonal matrices U and V^T can then be copied to the CPU. $d(i)$'s contains the singular values, i.e., diagonal elements of Σ and is on the CPU.

3.3 Performance of Singular Value Decomposition

In this section, we analyze the performance of our algorithm with the optimized CPU implementation of Singular Value Decomposition on MATLAB and Intel MKL 10.0.4 LAPACK. We enable dynamic threading in Intel MKL for good performance. We tested our algorithm on an Intel Dual Core 2.66GHz PC and a NVIDIA GeForce 8800 GTX graphics processor with CUDA 1.1 and a NVIDIA GTX 280 processor with CUDA 2.0. The 8800 GTX has 128 stream processors divided into 16 multiprocessors with 8 texture access units and a total of 768 MB of memory. The GTX 280 has 240 stream processors divided into 30 multiprocessors with 10 texture access units and a total of 1 GB of memory. According to NVIDIA, GTX 280 can achieve a peak performance of 622 GFLOPS and 8800 GTX of 345.6 GFLOPS. However, 8800 GTX gives 120 GFLOPS performance for single precision matrix multiply and 375 GFLOPS on GTX 280. We used Intel Core 2 Duo CPU E6750 @ 2.66Ghz processor for our experiments which is said to be rated 22.4 GFLOPS.

We generated 10 random dense matrices of single precision numbers for each size. The Singular Value Decomposition algorithm was executed for each matrix 10 times. To avoid a particularly good or bad sample, we averaged over the random matrices for each size. The average did not vary much if 10 or more matrices were used. Table 3.1 gives the overall average times in seconds using CUDA, MATLAB and Intel MKL. We achieve a speedup of 3.04-8.2 over the Intel MKL implementation and 3.32-59.3 over the MATLAB implementation for the square and non-square matrices on NVIDIA GTX 280. The CPU still out-performs the GPU implementation for small matrices. For large square matrices, the speedup increases with the size of the matrix. Figure 3.3 shows the time required for computing the Singular Value Decomposition of square matrices and Figure 3.4 shows the time required for computing the Singular Value Decomposition of rectangular matrices with leading dimension M=8K.

Bobda and Steenbock in [6] report only the timing for Singular Value Decomposition computation of $10^6 \times 10^6$ matrix which takes about 17 Hrs. We compute Singular Value Decomposition for much smaller matrices. Bondhugula *et al.* [7] only report the time for the bidiagonalization of the matrix. Dickson *et al.* [12] presented a programmable processor design suitable for Singular Value Decomposition, but do not give any Singular Value Decomposition results. Yamamoto *et al.* [53] give the optimized algorithm only for large rectangular matrices. The maximum speedup they achieve is 4 with CSX600 board over the CPU implementation for a large rectangular matrix, but get little speedup on smaller matrices.

SIZE	SVD MATLAB	SVD MKL	SVD GTX 280	SVD 8800	Speedup MKL/280
64×64	0.01	0.003	0.054	0.048	0.05
128×128	0.03	0.014	0.077	0.116	0.18
256×256	0.210	0.082	0.265	0.319	0.31
512×512	3.19	0.584	0.958	1.129	0.61
1K×1K	72	11.255	3.725	4.28	3.02
2K×2K	758.6	114.625	19.6	21.656	5.84
3K×3K	2940	402.7	52.8	61.31	7.62
4K×4K	6780	898.23	114.32	133.68	7.85
1K×512	5.070	2.27	1.523	3.749	1.48
2K×512	10.74	12.8	3.118	4.072	4.11
4K×512	34.33	54.7	8.311	12.418	6.58
8K×32	24.310	17.112	3.506	-	4.88
8K×64	47.87	33.7	5.016	-	6.72
8K×256	107.57	103.8	13.96	-	7.4
8K×512	137.98	215	26.33	-	8.16
8K×1K	254.26	417	50.364	-	8.2
8K×2K	1371.9	808	111.3	-	7.25

Table 3.1 Total computation time for Singular Value Decomposition (SVD) (in seconds) for different matrices

Table 3.2 gives the timings for bidiagonalization on the GPU and Intel MKL. Since Intel MKL routine performs the partial bidiagonalization, i.e., it bidiagonalizes the given matrix and returns the householder vectors instead of householder matrix, we compare it with the time required for partial bidiagonalization on the GPU. We achieve a speedup of 1.58-16.5 over Intel MKL on bidiagonalization. We experimented with different values of block size. We used the block size of 1 when n is small and 16 for large n. The performance for the square matrices increases with the size of the matrix. For rectangular matrices, the performance increases with the increase in n since blocking can be performed efficiently. The bidiagonalization by Bondhugula *et al.* [7] performs only partial bidiagonalization. Their timings given on *http://www.cs.unc.edu/ geom/Numeric/svd/* are best compared with partial bidiagonalization timings given in Table 3.2, Column 5. Our timing is comparable (11 seconds on GTX 280, 14 seconds on 8800 GTX, compared to 19 seconds on 7900). Raw rating of the GPU speed doesn't guarantee proportionate performance on this operation as can be seen from the minor speedup on GTX 280 over 8800 GTX. We also compare our work efficient diagonalization algorithm with the Intel MKL's diagonalization algorithm. Table 3.3 gives the timings for diagonalization on the GPU and Intel MKL. We achieve a speedup of 1.41-17.72 on the diagonalization step over Intel MKL implementation. The performance of our kernel is limited by the availability of registers per thread. We used 64 threads in a block for small matrices and 128 for larger matrices. It is done to keep all the multiprocessors active. We could achieve 67%-83% occupancy on diagonalization since only 8 blocks could be active at a time. The performance increases with the increase in the size of the matrix.

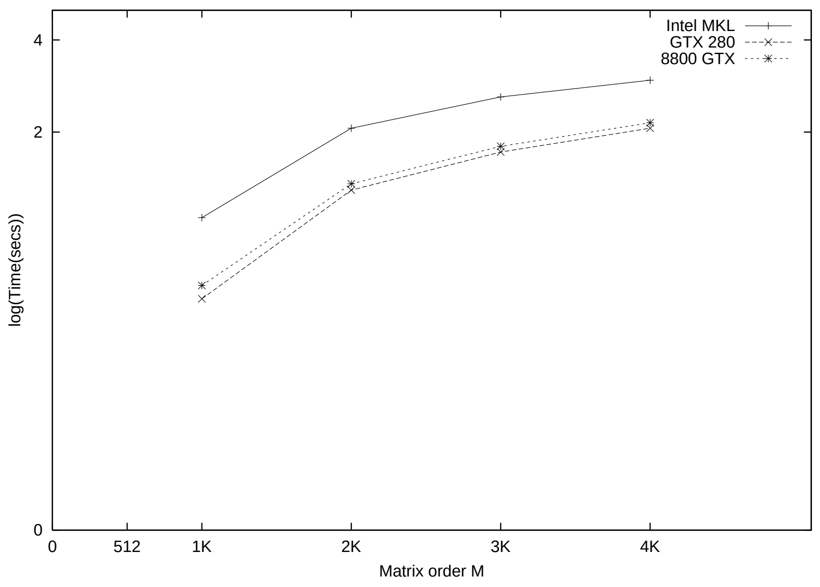


Figure 3.3 Singular Value Decomposition computation time for square matrices on Intel MKL and GPU(GTX 280 and 8800 GTX)

Figure 3.5 shows the speed up achieved on GTX 280 over Intel MKL for Singular Value Decomposition, partial bidiagonalization and diagonalization. A sustained bandwidth of 2 GB/s can be easily obtained from the CPU to the GPU. This will translate to $2ms$ of transfer time for $1K \times 1K$ matrix and $32ms$ for $4K \times 4K$ matrix. The Singular Value Decomposition computation time makes the data transfer time irrelevant. The timings in Table 3.1, 3.2 and 3.3 exclude the cost of transferring the matrix from the CPU to the GPU since the overhead of transfer of data from the CPU to the GPU and back is only to the order of tens of milliseconds.

GPUs are today limited to single precision arithmetic mostly. The GTX 280 has very limited double precision support, but at a very heavy performance penalty. We explored the discrepancy or error due to the reduced precision by comparing the results of the GPU version with the CPU version term by term. The maximum difference in the singular values was 0.013% but the average was less than 0.00005%. Similarly, the maximum error of any entry in the U and V matrices was 0.01% with an average of 0.001%. Figure 3.6 shows the plot of the error distribution in the singular values for a 3K$\times$3K matrix.

Tables 3.1, 3.2 and 3.3 bring out the following points. The optimized Intel MKL does a very good job on smaller matrices, but the performance of the GPU improves with the size of the matrix. The diagonalization contributes a major share to the performance improvement on the GPU especially on larger matrices. The bidiagonalization step takes more time on the CPU than the diagonalization step. On the GPU, however, diagonalization is much faster. The GTX 280, surprisingly, improves the performance only by 10-15$\%$ over the 8800 GTX but is able to handle larger matrices due to the larger internal memory.

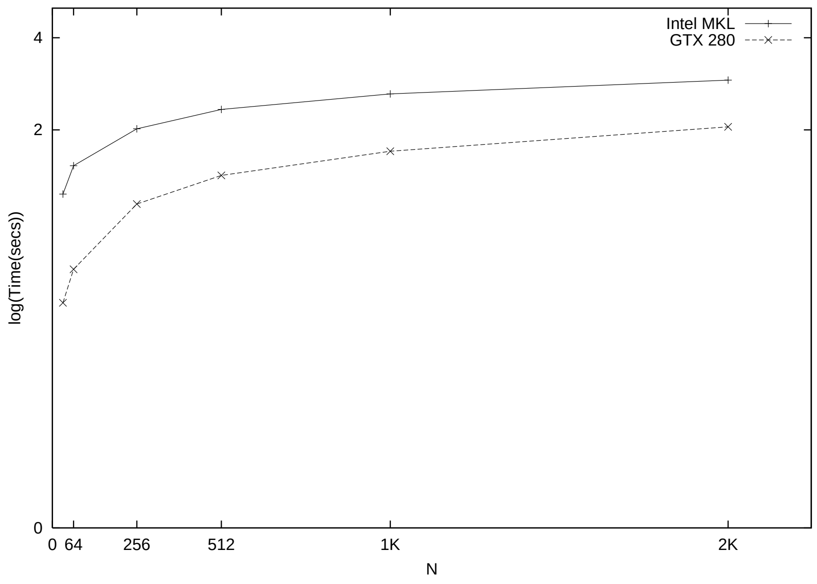


Figure 3.4 Singular Value Decomposition computation time for rectangular matrices($M \times N$) with leading dimension 8K and varying N on Intel MKL and GPU(GTX 280)

SIZE	Bidiag. GTX 280	Bidiag. 8800	Partial Bidiag. MKL	Partial Bidiag. GTX 280	Partial Bidiag. 8800
128×128	0.060	0.075	0.003	0.050	0.063
512×512	0.570	0.637	0.1478	0.373	0.430
1K×1K	2.40	2.588	3.8122	1.068	1.304
3K×3K	41	51.80	184	11.114	14.088
4K×4K	92.7	105.071	361.8	21.8	27.576
8K×32	1.499	–	0.020	0.143	0.066
8K×256	11.8	–	2.721	1.245	1.276
8K×512	23.8	–	13.8	2.650	2.8
8K×2K	101	–	220.500	14.3	20.281

Table 3.2 Bidiagonalization time (in seconds) for different matrices

SIZE	Diagonalization Intel MKL	Diagonalization GTX 280	Diagonalization 8800
128×128	0.010	0.017	0.041
512×512	0.5439	0.385	0.381
1K×1K	6.417	1.3	1.347
3K×3K	159.413	11.6	11.821
4K×4K	354.3	20	21.7
8K×32	0.022	0.007	–
8K×256	0.564	0.159	–
8K×512	2.239	0.530	–
8K×2K	100.000	8.2	–

Table 3.3 Diagonalization time (in seconds) for different matrices

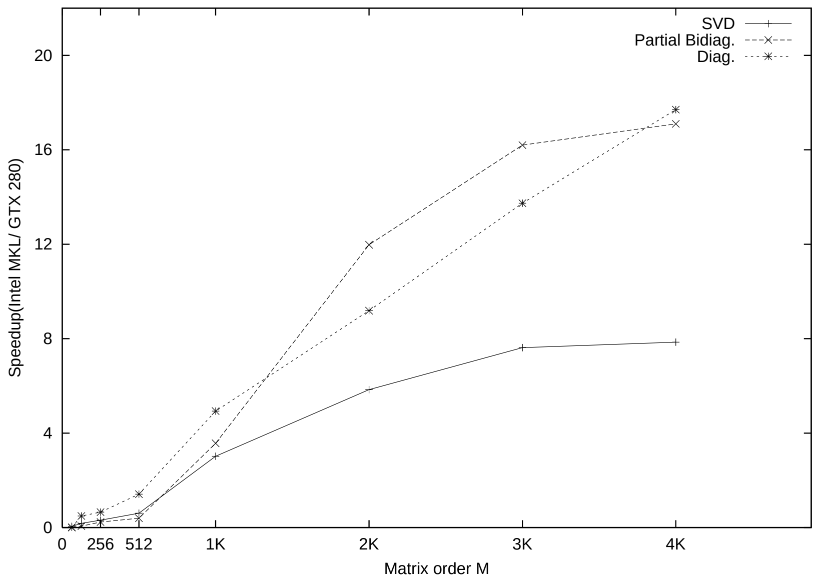


Figure 3.5 Speedup for Singular Value Decomposition, Partial Bidiagonalization and Diagonalization on GTX 280 over Intel MKL

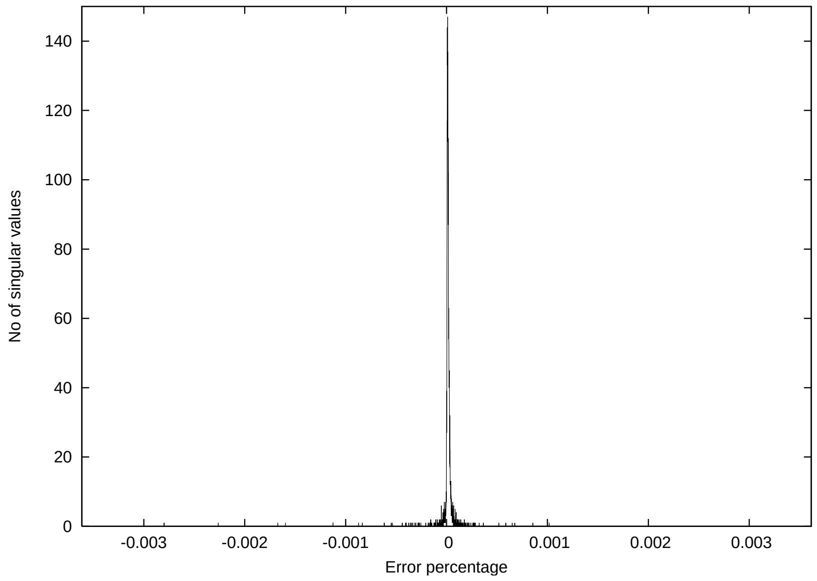


Figure 3.6 Discrepancy plot for singular values of a 3K$\times$3K matrix

Chapter 4

Ray Tracing Parametric Patches

Kajiya's ray patch intersection algorithm is described briefly in Section 2.4. We describe the Kajiya's ray patch intersection algorithm and its implementation on the GPU in detail in the following sections.

4.1 The Ray-Bézier Patch Intersection Algorithm

We adapt the ray-patch intersection approach proposed by Kajiya [25]. It is both robust and suitable for implementation on a pipelined architecture. It needs a large number of floating point operations. There were attempts to closely approximate Kajiya's algorithm by finding the closest point of intersection using Newton's iteration on the CPU [18]. However, no implementation exists to find the exact points of intersection in interactive time. The advent of GPUs with support for higher precision and their ever-growing amount of parallel horsepower makes them a tempting resource for such numerical computations. We try to leverage the performance of GPUs for ray tracing parametric surfaces, being both computationally demanding and massively parallel.

A Bézier bi-cubic patch of degree 3 is defined by

$$\mathbf{Q}(u,v) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \mathbf{MPM}^t \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix}, \tag{4.1}$$

where $0 \le u, v \le 1$ are the patch parameters. **P** is the control matrix and **M** is the Bézier basis matrix.

Parametric derivative with respect to u is given by

$$Q_u(u,v) = \frac{\partial Q}{\partial u} = \begin{bmatrix} 3u^2 & 2u & 1 & 0 \end{bmatrix} \mathbf{MPM}^t \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix}.$$

$Q_v(u,v)$ for v can be given similarly.

A ray is represented as intersection of two planes (l^0, l^1). Intersecting a ray with a patch gives two algebraic equations:

$$\begin{aligned} a_{ij}u^{3-i}v^{3-j} &= 0 \qquad (4.2)\\ b_{ij}u^{3-i}v^{3-j} &= 0, \qquad (4.3) \end{aligned}$$

where $i, j \in \{0, 1, 2, 3\}$. $a_{ij} = l_k^0 \mathbf{P}_{ij}^k$ and $b_{ij} = l_k^1 \mathbf{P}_{ij}^k$, where $k \in \{0, 1, 2, 3\}$ and l_k^0 and l_k^1 are the horizontal and vertical planes representing the ray.

The intersection points of Equations 4.2 and 4.3 give the (u, v) parameter values at which the ray intersects the patch. There are a maximum of 18 intersection points when a ray pierces a patch. Equations 4.2 and 4.3 can be written as polynomials in u:

$$\begin{aligned} a(u, v) &= A_0u^3 + A_1u^2 + A_2u + A_3 = 0\\ b(u, v) &= B_0u^3 + B_1u^2 + B_2u + B_3 = 0, \end{aligned}$$

where A_i and B_i are cubic polynomials in v. To calculate the intersection of $a(u, v)$ and $b(u, v)$ curves we find the common roots of the cubic polynomials in u. The vanishing of the Bezout determinant R given as

$$R = \begin{vmatrix} B_0A_1 - B_1A_0 & B_0A_2 - B_2A_0 & B_0A_3 - B_3A_0 \\ B_0A_2 - B_2A_0 & \begin{matrix} B_0A_3 - B_3A_0 \\ +B_1A_2 - B_2A_1 \end{matrix} & B_1A_3 - B_3A_1 \\ B_0A_3 - B_3A_0 & B_1A_3 - B_3A_1 & B_2A_3 - B_3A_2 \end{vmatrix} \qquad (4.4)$$

is an 18 degree polynomial in v.

For every real root v_i such that $0 \leq v_i \leq 1$ of R, u_i can be obtained by computing the GCD of $a(u, v_i)$ and $b(u, v_i)$ using the standard Euclidean algorithm for polynomials. Real (u_i, v_i) pair represents a real point of intersection. A ray can intersect a patch in 18 points (real or complex). The (x, y, z) coordinates of the point in 3D space are computed using Equation 4.1 and the corresponding normal **n** is computed as the cross product of the parametric tangents:

$$\mathbf{n} = \frac{\partial Q}{\partial u} \times \frac{\partial Q}{\partial v}. \qquad (4.5)$$

4.2 Ray Tracing Algorithm on GPU

Algorithm 6 shows the high level view of our ray tracing system. We create a BVH using the surface area heuristics (SAH) of parametric patches. This reduces the number of ray patch intersections significantly. The patch id of every potential intersection is recorded against the ray by traversing the BVH.

We form the 18 degree polynomial for all ray patch intersections in parallel on the GPU. The 18 roots of the polynomials, i.e., v parameters are found one by one using the Laguerre's root finding procedure on GPU for all intersections. Finally, the u parameter is obtained by finding the GCD of the corresponding cubic polynomials. The 3D point and the corresponding normal is calculated subsequently. We recursively spawn secondary rays at each intersection point and intersect with the BVH. Similar to primary rays, values for parameter v and u are found at each level to find a real intersection point. For a real point of intersection the shading information is accumulated. All the calculations are performed in the world space. The polynomial evaluation and root finding are done using complex arithmetic. All numbers are represented using double precision as single precision turned out to be inadequate.

The overall process is controlled and coordinated by the CPU, with separate kernels for steps 2-8. We describe the individual components in detail in the following subsections.

```
Preprocessing
1: Create the BVH of the model using SAH on the CPU. Store in depth first order with skip pointers.
   Repeat every frame
2: Compute $M$ horizontal and $N$ vertical planes on the GPU in the world space using frustum corner
   information.
3: Traverse BVH for every ray in parallel on the GPU. Record the (rayid, patchid) for each patch the
   ray may intersect. Let Num_Intersect be the total number of such potential intersections.
   For all potential intersections in parallel on the GPU.
4: Compute the $18$ degree polynomial by evaluating $R$.
5: Solve the polynomial, giving $18$ complex roots for $v$.
6: For each real $v$, find the GCD of bicubic polynomials, giving the $u$ parameter.
7: Find $(x, y, z)$ and normal $\mathbf{n}$ for real $(u, v)$ pair.
8: Spawn secondary (shadow or reflection) rays. Compute the plane equations in the direction of the
   secondary ray. Repeat steps 3-7.
9: Compute the shading information by combining colors from all levels and record the color.
```

Algorithm 6: Overview of our ray tracing system on GPU

4.2.1 Preprocessing

We create a BVH of axis-aligned bounding box of the patches in the object space on the CPU once. The algorithm approximates the SAH using streamed binning of centroids [21]. BVH is better suited for dynamic scenes. It examines the potential partitions at every level and uses a split plane and dimension which minimizes the SAH cost function. Depth first layout with skip offsets [45] is used to store the BVH on GPU. We use the texture memory to store the patch and BVH data to exploit it caching properties.

A BVH node is represented using 56 bytes: six double values for the axis aligned bounding box, one 32-bit integer value for the patch id (-1 for non-leaf nodes) and one 32-bit integer value to store the skip pointer. Each value is stored in a different texture. A patch is represented using 48 doubles on GPU. In the texture memory, the first control point of all the P patches are stored linearly followed by the second and so on as shown in Figure 4.1. $(X(i,j), Y(i,j), Z(i,j))$ represents the j^{th} control point of i^{th} patch. Creating the BVH of Killeroo model with 11532 patches takes 340ms on the CPU. The BVH can be recomputed on the CPU in parallel with the intersections on the GPU, if the patches are dynamic.

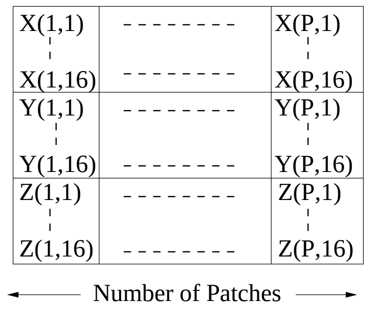


Figure 4.1 Patch data layout in the texture memory of the GPU. The first control point of all the P patches are stored linearly followed by the second and so on. The layout increases the spatial locality of memory accesses.

To find the ray patch intersection, we represent a ray as the intersection of two planes. One plane remains constant as we move along a row on the screen, similarly along a column. For a screen resolution of $M \times N$ we compute M horizontal and N vertical planes. $M \times N$ rays can be represented using M + N plane equations which are computed once on the GPU. A ray passing through the pixel (i, j) uses i^{th} horizontal and j^{th} vertical planes. This is implemented on the GPU. We use kernels *planesX* and *planesY* to compute horizontal and vertical planes respectively, in which every thread computes a plane equation. It takes less than 1 milliseconds to estimate the plane equations on the GPU. Plane equations are computed every frame in the world space using frustum corner information.

4.2.2 BVH Traversal on the GPU

Each ray traverses the BVH of patches in depth first order on the GPU. The ray id (x, y) and patch id tuple is stored for every potential intersection after the traversal on the global memory. We then rearrange the ray-patch intersection data for further processing.

We write separate kernels to map the BVH traversal to efficient GPU execution. We assign a thread for each ray in screen space and traverse the BVH independent of other rays. Kernels (*traverse*, *scan*, *rearrange*) are executed in that order. We use an execution configuration of $(M \times N)/256$ blocks, each with 256 threads for the kernels. We allocate a memory block of size $M \times N \times K$ for the working queue *Mem_Queue* used in the traversal procedure. The value of K depends on the maximum number of ray patch intersections. We use a value of 6 in the current version. We initialize the integer array *Sum* with zeros.

In the *traverse* kernel, the ray id is computed by each thread from block and thread information. Starting node data of the BVH is read from position 0 in the texture. At each node we perform intersection tests with AABB of the node using ray-AABB test. Whenever a ray intersects the BVH leaf node, we store the patch id in *Mem_Queue*. The node position is incremented when a ray intersects the node, otherwise the node position is updated to value stored in the skip pointer. The number of intersected children is saved in *Sum*. Traversing the BVH in depth first order avoids the use of stack. An efficient parallel packet based traversal algorithm using a shared stack is proposed by [21]. However, we restrict ourselves to simple depth first traversal.

Since the intersected patch id information is irregular in *Mem_Queue*, we use a scan to rearrange the data [41]. We perform a parallel scan operation using *scan* kernel on *Sum* array and store the results in *Prefix_Sum*.

We use the *rearrange* kernel to write pixel positions and rearrange the patch id's in linear arrays using the *Prefix_Sum* values. We propagate the value of patch id from *Mem_Queue* to the output array *patch_ID* specified by offset equal to Sum_i and length equal to $Prefix\_Sum_i$. Ray id (x, y) is written to *pixel_x* and *pixel_y*. The rearranged regular data is used independently for further computations on the GPU. It gives the number of ray patch intersections, say *Num_Intersect*. Thus, we store *Num_Intersect* patch id's in *patch_ID* and ray id's in *pixel_x* and *pixel_y*. Figure 4.2 shows an example of BVH traversal and data rearrangement.

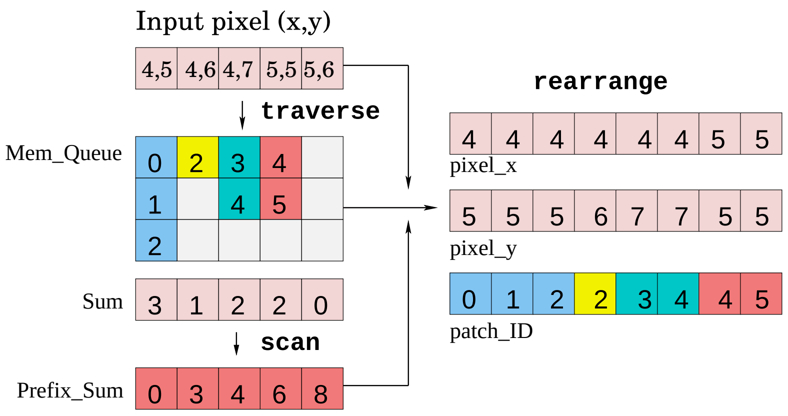


Figure 4.2 Depth first ray traversal example. Each thread in the *traverse* kernel traverses the BVH for the ray assigned to it. The patch id of the potential intersections and the total number of intersections are recorded in *Mem_Queue* and *Sum* respectively. The intersected patch id and pixel information is then rearranged using *rearrange* kernel after performing scan on *Sum* and stored in *patch_ID*, *pixel_x* and *pixel_y*.

In our algorithm, each ray makes independent traversal decisions resulting in multibranching within a CUDA warp of threads. We minimize divergence by assigning adjacent pixel locations to the nearby threads as we expect adjacent rays to traverse the tree nodes in the same order. Reading the node data from textures incurs little read overhead. Our kernels use only a few live registers and limited shared memory, giving 100% device utilization.

4.2.3 Computing the 18 degree polynomial

We compute the 18 degree polynomial in parallel for *Num_Intersect* intersections. Equation 4.4 can be simplified as

$$R(a,b) = \begin{vmatrix} a & b & c \\ b & c+d & e \\ c & e & f \end{vmatrix}, \tag{4.6}$$

where every term is a 6 degree polynomial in v. Expanding the determinant R requires the computation of six 6 degree polynomials, six 12 degree polynomials, and three 18 degree polynomials.

We use a kernel *bezout* for forming the 18 degree polynomial in v parameter for all the ray patch intersections in parallel. A CUDA block processes 16 ray patch intersections, each using 21 threads. Number of blocks used is the ceiling of *Num_Intersect*/16, each with 21 $\times$ 16 threads. The use of 336 threads per block may seem inefficient on today's GPU but gives the best performance among the alternatives. The number of ray patch intersections can be greater than the number of visible pixels. Arrays *patch_ID*, *pixel_x* and *pixel_y* are passed to the kernel and the patch control points for every pixel are read from the texture memory. Pseudocode for *bezout* kernel which computes 16 intersections in i^{th} block is given in Algorithm 7.

```
1:  Allocate Shared memory (SM), register $q$ per thread
2:  $i$ = Block Id, $j = 0$
3:  **while** $j < 16$ **do**
4:      Read *ind* = *patch_ID*[$i \times 16 + j$], *pixx* = *pixel_x*[$i \times 16 + j$], *pixy* = *pixel_y*[$i \times 16 + j$]
5:      Read *planeX* and *planeY* for *pixx* and *pixy* and copy patch data to shared memory
6:      Compute 32 coefficients $A = (A_0, A_1, A_2, A_3)$ and $B = (B_0, B_1, B_2, B_3)$ using 32 threads
7:      Write the coefficients to *d_A* and *d_B* in the global memory
8:      Increment $j$
9:  **end while**
    For all the sixteen intersections assigned to a block
10: Read $A$ and $B$ coefficients in shared memory
    Compute:
11: ($a$,$b$,$c$) using 21 threads and store in SM
12: ($d$,$e$,$f$) using 21 threads and store in SM
13: $T_1 = (c+d)f - e^2$ using 13 threads and store in SM
14: $S_1 = a((c+d)f - e^2)$ using 19 threads and store in $q$
15: $T_2 = bf - ce$ using 13 threads and store in SM
16: $S_2 = b(bf - ce)$ using 19 threads and update $q$
17: $T_3 = be - c(c+d)$ using 13 threads and store SM
18: $S_3 = c(be - c(c+d))$ using 19 threads and update $q$
19: 19 register values of $q$ corresponding to coefficients of a polynomial is written to *d_pr* in the global
    memory
```

Algorithm 7: *bezout* kernel evaluates 16 polynomials assigned to i^{th} intersection block

Kernel computes the (A_0, A_1, A_2, A_3) and (B_0, B_1, B_2, B_3) coefficients sequentially for the sixteen intersections in the block. 32 threads compute the coefficients in parallel. The coefficients are written to *d_A* and *d_B* in the global memory as shown in Figure 4.4(a). $A(i, j)$ denote the j^{th} coefficient of A for i^{th} ray-patch intersection.

The computations in a block requires a maximum of 21 $\times$ 16 threads. We formulate the parallel computations for sixteen intersections in the block as matrix operations. Computing a, b and c (degree 6) polynomials together for an intersection requires calculating 21 coefficients in parallel. Each half warp of 16 threads computes a coefficient of the 21 coefficients for the sixteen intersections. We therefore require, 21×16 threads in a block, of which first 7×16 compute coefficients of a, next 7×16 compute coefficients of b and next 7×16 compute coefficients of c. Similarly for d, e and f. Computation of T_i and S_i where $i \,\epsilon\, \{1, 2, 3\}$ as computed in Algorithm 7 requires 13×16 and 19×16 threads respectively. Steps 10-19 are executed sequentially in the kernel. Mapping between threads and the coefficients to be computed of polynomial T_i is shown in Figure 4.3. Similar division is used for computing (a,b,c), (d,e,f) and S_i.

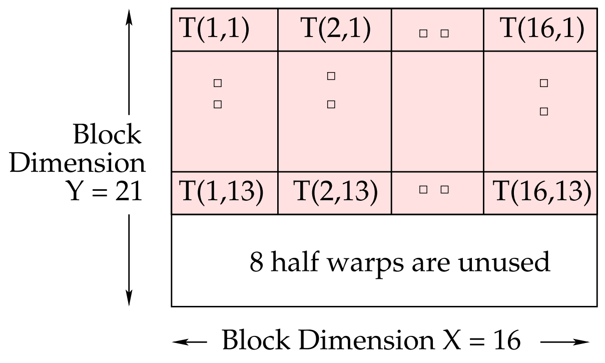
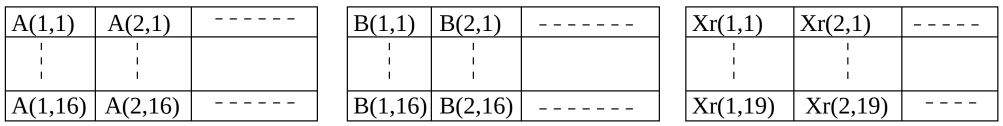


Figure 4.3 Each thread of the block computes a coefficient for an intersection. $T(k, j)$ denotes the j^{th} coefficient of T_i polynomial for the k^{th} intersection in the block. A thread with thread id (k, j) computes $T(k, j)$. Every half warp computes the same coefficient but for different intersection. In this case, 4 warps do not perform any computations. 2 warps do not perform any computations while computing S_i. No threads are wasted while computing (a,b,c) and (d,e,f)

Every half warp computes the same coefficients but for different intersections, it runs for same length. For example, first coefficient of a requires only a multiplication operation, but second coefficient requires 2 multiplications and an addition operation. Texture memory is used to pass the lengths of the calculation for a coefficient. Coefficients for 12 and 18 degree polynomials are computed similarly.

We observe thread divergence between half warps within a warp during coefficient computation. It cannot be avoided due to different lengths of computation required for different coefficients. Other mapping configurations give worse results. Although, few warps are wasted at different steps in the kernel, this configuration utilizes the resources best.

Sixteen 18-degree polynomials are formed on each CUDA block, and are written to the global memory as shown in Figure 4.4(b). The real coefficients are stored in *d_pr* in the global memory. While



(a) Data layout for *d_A* and *d_B* in the global memory (b) Data layout for *d_pr* in the global memory

Figure 4.4 The first coefficient of A, B for the *Num_Intersect* intersections is stored continuously in the linear memory. Similar layout is used to store the real coefficients of the polynomial. The layout increases coherent access to the coefficients and avoids any uncoalesced data read and write since a thread is assigned for every intersection. The memory block has a width of *Num_Intersect*.

writing the polynomial coefficients, we store the first coefficient of all the polynomials continuously followed by the second, etc. $Xr(i, j)$ denotes the j^{th} coefficient of the i^{th} polynomial respectively. This data layout avoids all uncoalesced reads and writes.

The performance of the *bezout* kernel is limited by the use of registers. We are able to obtain a occupancy of 69% using this configuration.

4.2.4 Finding the polynomial roots

The polynomial obtained in the above section is solved using the Laguerre's root finding method. We use this method as it guarantees the convergence to some root of the polynomial irrespective of the initial guess and has good global convergence properties. Because of the cubic convergence, 5 to 7 iterations are sufficient in practice. Algorithm 8 shows the procedure for finding a root x of polynomial $p(x)$ using Laguerre's method. The constants *maxIter* and *threshold* control the accuracy.

```
1:  Set initial guess $x_0$
2:  for $k = 0$ to maxIter do
3:      $G = p'(x_k)/p(x_k)$
4:      $H = G^2 - p''(x_k)/p(x_k)$
5:      $\alpha = n/(G \pm \sqrt{(n-1)(nH - G^2)})$
6:      $x_{k+1} = x_k - \alpha$
7:      if $\alpha <$ threshold then
8:          $break$
9:      end if
10: end for
```

Algorithm 8: Laguerre's root finding procedure

The v parameter corresponding to 18 complex roots of the polynomial are calculated first. We find all the roots using the Laguerre's method, initializing them to $1/\sqrt{2} + 1/\sqrt{2}j$. Once a real root is found, the polynomial is deflated by synthetically dividing by the monomial corresponding to the root. Since complex roots always appear in pairs, the polynomial is deflated by the quadratic polynomial obtained by multiplying the complex conjugates for complex root. The coefficients of the deflated polynomial are always real.

We use separate kernels to find the root and to deflate the polynomial. We use an execution configuration of *Num_Intersect*/64 blocks, each with 64 threads for the kernels. The order in which the kernels are invoked from the CPU is shown below:

```
1: $i = 0$, Set initial guess for all roots
2: while $i < 18$ do
3:     Invoke laguerre kernel for finding the $i^{th}$ root $x_i$
4:     Invoke deflate kernel, deflates polynomial by $x_i$
5:     Increment $i$
6: end while
```

Each CUDA block processes 64 intersections in parallel. Every thread handles one polynomial. A root is found after every invocation of *laguerre* kernel. We maintain a shared variable *count* in a block. It is initialized to 0 for every iteration in the kernel. If the root is not found for a polynomial in an iteration in the block, the corresponding thread updates the *count* value to 1. The kernel loops as long as a polynomial exists for which a root has not been found or if the maximum number of iterations is not reached. Once a root is found for all the polynomials in the block, only the real root is written to *d_realv* in the global memory below the previously found real root. Since, roots are required for deflating the polynomial, we also store the complex roots in a temporary array which can be reused after deflating the polynomial.

Finding a root of a polynomial requires polynomial coefficients which are represented using $19 \times 8 = 152$ bytes. Each block would require $152 \times 64 = 9728$ bytes. Polynomial coefficients are transferred repeatedly from the global memory to the kernel. The *laguerre* kernel executes all steps listed in Algorithm 8. The polynomial coefficients are read once in the *deflate* kernel and the updated coefficients are written to *d_pr* in the global memory.

The effective number of registers available is half when using double precision arithmetic. The performance of *laguerre* kernel is limited by the number of registers used per block. We use 64 threads per block. Splitting the root finding method into *laguerre* and *deflate* kernels reduces the number of registers used per kernel. We are able to keep the occupancy of the device above 25% for the *laguerre* kernel and 50% for *deflate* kernel. The first real root of all the polynomials are stored continuously in the linear memory, followed by the real second root, etc., as shown in Figure 4.5(a). $Vr(i, j)$ denotes the j^{th} real root of i^{th} polynomial. Pseudocode for *laguerre* and *deflate* kernels which finds i^{th} root of $p(x)$ is given in Algorithm 9.

4.2.5 Computing the GCD of bi-cubic polynomials

To find the u parameter at intersections, we successively substitute the v_i obtained earlier in Equations 4.2 and 4.3. The GCD of the polynomials gives the u parameter, $u_i = GCD(a(u, v_i), b(u, v_i))$.

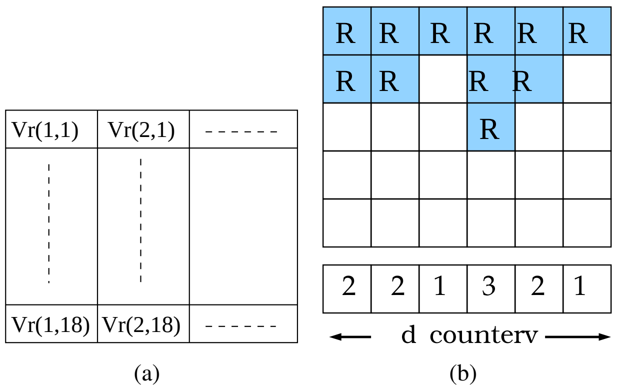


Figure 4.5 4.5(a) shows the data layout for *d_realv* in the global memory. The memory block has a width of *Num_Intersect*. 4.5(b) shows how the real roots of 5 degree polynomials are arranged to lie continuously in a column. The count of the number of real roots for every polynomial is stored in *d_countv*

The GCD is computed using the standard Euclidean algorithm for polynomials. The 3D point (x, y, z) and the corresponding normal **n** for a real (u_i, v_i) is calculated as given in Equations 4.1 and 4.5. We compute the GCD only for real v_i and further reduce the computations by processing only real v_i and u_i pairs. We invoke (*gcd*, *3Dpoints*) in that order using *Num_Intersect*/256 blocks, each with 256 threads.

We use a *gcd* kernel which finds u_i only for real v_i by computing the GCD of cubic polynomials for the intersections. The number of real v for every polynomial can be read from *d_countv* array. Every CUDA block finds the u parameter for 256 polynomials. Each thread handles one polynomial and loops for *count* (read from *d_countv*) times and finds the GCD using the Euclidean algorithm.

Whenever a real (u_i, v_i) pair is found, flag value is set and the real pair counter is updated. The u_i and v_i are then written below the previous real root in the column in *d_realu* and *d_realv* respectively. The point is rendered only if the flag indicates a real point of intersection. We keep the count of the number of real (u, v) pair for every polynomial in an integer array *d_countu*. The (A_0, A_1, A_2, A_3), (B_0, B_1, B_2, B_3) coefficient values are loaded from the global memory. Limited shared memory makes it unsuitable to store the coefficients.

We use kernel *3Dpoints* which finds the points (x, y, z) and **n** only for real (u, v). Every CUDA block finds the points for 256 intersections. Each thread handles one intersection and loops for *count* (read from *d_countu*) times and finds the (x, y, z) points. The patch control points could not be stored in the shared memory, and hence are read every time from the texture.

The performance of the kernels is limited by the availability of registers. Shared memory is used to store the register values. We achieve occupancy of 25% for *gcd* and *3Dpoints* kernel. Pseudocode for *gcd* and *3Dpoints* which processes the j^{th} polynomial is given in Algorithm 10.

```
    *laguerre* kernel to find $i^{th}$ root of $p(x)$
1:  Initialize shared variable *count* to 1
2:  **while** *count* $== 1$ **do**
3:      *count* $= 0$
4:      Read $p(x)$ from *d_pr* and compute $\alpha$ as given in Algorithm 8
5:      **if** $\alpha >$ *threshold* **then**
6:          Update $x_i$ to $x_i - \alpha$, *count* $= 1$
7:      **else**
8:          *break*
9:      **end if**
10: **end while**
11: Write real $x_i.r$ to *d_realv*. Store $x_i$ in temporary array.
12: For real $x_i$, increment value in *d_countv*.
    *deflate* kernel to divide $p(x)$ by $x_i$
13: Read $p(x)$. Read $x_i$ from temporary array.
14: If $x_i$ is real, divide $p(x)$ by $x - x_i.r$
15: For complex $x_i$, divide $p(x)$ by $x^2 - 2x_i.rx + (x_i.r)^2 + (x_i.i)^2$
16: Write new coefficients to *d_pr*
```

Algorithm 9: Find i^{th} root of the polynomial $p(x)$

4.2.6 Secondary Rays

We spawn secondary (shadow or reflected) rays at the nearest real point of intersection of the primary ray with a patch. Similar to primary ray traversal, the secondary ray traverses the BVH and stores (primary ray id, patch id) as in Step 3 of Algorithm 6. This information is used for computing the color at a pixel at which the primary rays intersects the patch.

Two orthogonal planes for secondary rays are chosen using their direction. The steps 3-8 in Algorithm 6 are invoked separately for shadow and reflected ray. The real points of intersection of the secondary ray with the patch is found using the Laguerre's method and Euclidean algorithm on the GPU.

If a real intersection is found for a shadow ray, the point from which it is spawned is shadowed. Similarly, for the reflected ray, the nearest real intersection is used to compute the final color at the point from which it is spawned.

4.2.7 Rendering

We use the standard illumination equation for computing the color value at a pixel. Points with real (u, v) value found by intersecting primary rays with the patch and are not shadowed are used to compute the color at a pixel (x, y), i.e., primary ray id. For reflected ray, full illumination at the intersection point is computed. Contributions of the secondary rays are combined in the reverse order as in standard ray

```
    *gcd* kernel computes $u$ parameter
1:  $count_1$ = *d_countv*[j]
2:  Compute $j$ value from block and thread index
3:  Initialize $count_2$ to zero
4:  **while** $i < count_1$ **do**
5:      Read $v.r$ from *d_realv*
6:      Compute bicubic $a(u, v.r)$ and $b(u, v.r)$
7:      $x$ = GCD$(a, b)$
8:      **if** $x.i == 0$ and $0 \leq x.r \leq 1$ **then**
9:          $index = j + count_2 \star Num\_Intersect$
10:         *d_realv*[*index*] = $v.r$
11:         *d_realu*[*index*] = $x.r$
12:         Increment $count_2$
13:     **end if**
14:     Increment $i$
15: **end while**
16: *d_countu*[$j$] = $count_2$

    *3Dpoints* kernel computes $(x, y, z)$ and **n**
17: *count* = *d_countu*[$j$]
18: Compute $j$ value from block and thread index
19: Initialize *d_flag* to zero
20: **while** $i < count$ **do**
21:     Read $v$ from *d_realv* and $u$ from *d_realu*
22:     Compute $x$, $y$, $z$ and **n** as given in Equations 4.1 and 4.5
23:     *d_flag*[$j + i \star Num\_Intersect$] = 1
24:     Increment $i$
25: **end while**
```

Algorithm 10: Finds the GCD and 3D points for j^{th} polynomial

tracing. The color and depth information of the point is stored in the color and depth buffer respectively. Per pixel normals are used to compute the light equation.

4.3 Results and Analysis

We test our ray tracing system for primary and secondary rays (at 512×512 resolution). We use the Teapot (32 patches), Bigguy (3570 patches) and Killeroo (11532 patches) Bézier models. We build the BVH on the CPU for these models using streamed binning algorithm. The leaf of the BVH contains exactly one patch. All measurements were done using an Intel 2.4GHz Core 2 CPU and NVIDIA GeForce 280 GTX graphics card and GTX 480 (Fermi) graphics processor. The kernels were compiled using CUDA 2.2. We used a threshold of 10^{-20} to 10^{-15} for convergence of root finding.

4.3.1 Ray Tracing Performance

Figures 4.10(a)-4.10(e) shows the results of our ray tracing system on different models. Constructing BVH using the SAH gives high quality trees and achieves fast construction time. BVH traversal on GPU is easier to implement and uses less live registers. Although, rays make independent traversal decision and traverse the tree in depth first order the time taken is very less in comparison to time taken to ray trace the scene. The number of patches intersected per ray depends on the depth complexity of the model and the heuristics used while constructing the BVH. In our case, number of patches intersected per ray are in the range 2-4.

Model	Type of Ray	Number of intersections	Patch per ray	BVH Traversal Time(secs)	Polynomial Formation Time(secs)	Solve poly-nomial Time(secs)	GCD, (x,y,z) and **n** Time(secs)	Time per frame (secs)
Teapot	P	54389	2.01	0.004	0.019	0.175	0.013	0.211
	S	29626	2.32	0.003	0.012	0.111	0.010	0.136
	R	41096	3.21	0.004	0.031	0.143	0.011	0.189
Bigguy	P	114048	3.23	0.007	0.043	0.352	0.015	0.417
	S	114112	3.47	0.007	0.048	0.350	0.015	0.420
	R	143040	4.34	0.008	0.104	0.480	0.022	0.614
Killeroo	P	127040	1.43	0.010	0.050	0.390	0.016	0.466
	S	138240	1.72	0.011	0.061	0.420	0.016	0.508
	R	146432	1.82	0.013	0.105	0.446	0.022	0.586

Table 4.1 Time for executing different kernels of the ray tracing algorithm on GPU for primary (P), shadow (S) and reflection rays (R) for different models on GTX 280.

Model	Type of Ray	Number of intersections	Patch per ray	BVH Traversal Time(secs)	Polynomial Formation Time(secs)	Solve poly-nomial Time(secs)	GCD, (x,y,z) and **n** Time(secs)	Time per frame (secs)
Teapot	P	54389	2.01	0.001	0.0064	0.0596	0.004	0.071
	S	29626	2.32	0.001	0.0037	0.0333	0.003	0.041
	R	41096	3.21	0.0012	0.0093	0.0432	0.0033	0.057
Bigguy	P	114048	3.23	0.00245	0.01505	0.1242	0.00525	0.147
	S	114112	3.47	0.00245	0.0168	0.1235	0.00525	0.148
	R	143040	4.34	0.00248	0.03224	0.1488	0.00682	0.190
Killeroo	P	127040	1.43	0.0035	0.0175	0.1375	0.0056	0.164
	S	138240	1.72	0.00385	0.02135	0.148	0.0056	0.179
	R	146432	1.82	0.00429	0.03465	0.149	0.00726	0.195

Table 4.2 Time for executing different kernels of the ray tracing algorithm on GPU for primary (P), shadow (S) and reflection rays (R) for different models on GTX 480.

Model	Type of Ray	Time (secs) GTX 480	Time (secs) GTX 280	Speedup
Teapot	P	0.071	0.211	2.94
	S	0.041	0.136	3.28
	R	0.057	0.189	3.3
Bigguy	P	0.147	0.420	2.83
	S	0.148	0.420	2.83
	R	0.190	0.614	3.23
Killeroo	P	0.164	0.466	2.83
	S	0.179	0.508	2.83
	R	0.195	0.586	2.99

Table 4.3 Shows the time taken in seconds on GTX 280 and GTX 480 to render a frame for different models and type of rays.

Model	Time (secs) GTX 480	Number of Batches
Killeroo	0.3612	1
	0.182	2
	0.090	4

Table 4.4 Shows the time taken in seconds on GTX 480 to render a frame for Killeroo model with primary rays and 280000 intersections. The intersections are processed in batches using limited device memory.

The time taken for each step of ray tracing system for different rays and models on GTX 280 is given in Table 4.1. The measurements are averaged over 10-15 frames. The total time includes the time taken for computing the plane equations, about 1-2 ms per frame. Figure 4.6 presents the time spent in different kernels for primary, shadow and reflection rays. Finding the roots of the higher degree polynomials is the most time critical step of our algorithm and takes about 82% of the total time on average. Table 4.1 shows that the time to process a ray-patch intersection is about 3.7 microseconds. It is nearly constant for primary and secondary rays. This gives us a way to predict performance on other situations, as the rendering time scales linearly with the number of intersections (i.e., total *Num_Intersect*). Table 4.2 gives the time taken for ray tracing models on GTX 480. The time taken to process a ray-patch intersection is about 1.28 microcseconds on GTX 480.

Table 4.4 shows the time taken to process the intersections for Killeroo model in batches. The intersections are processed using the limited device memory. Experiments show that the time taken to process a intersection remains almost constant if enough intersections are batched to keep the device busy. Table 4.5 shows the timings of multi GPU implementation on Tesla S1070. Per intersection processing time remains almost constant on multi GPU. The overall rendering time decreases by a factor of the number of GPUs used.

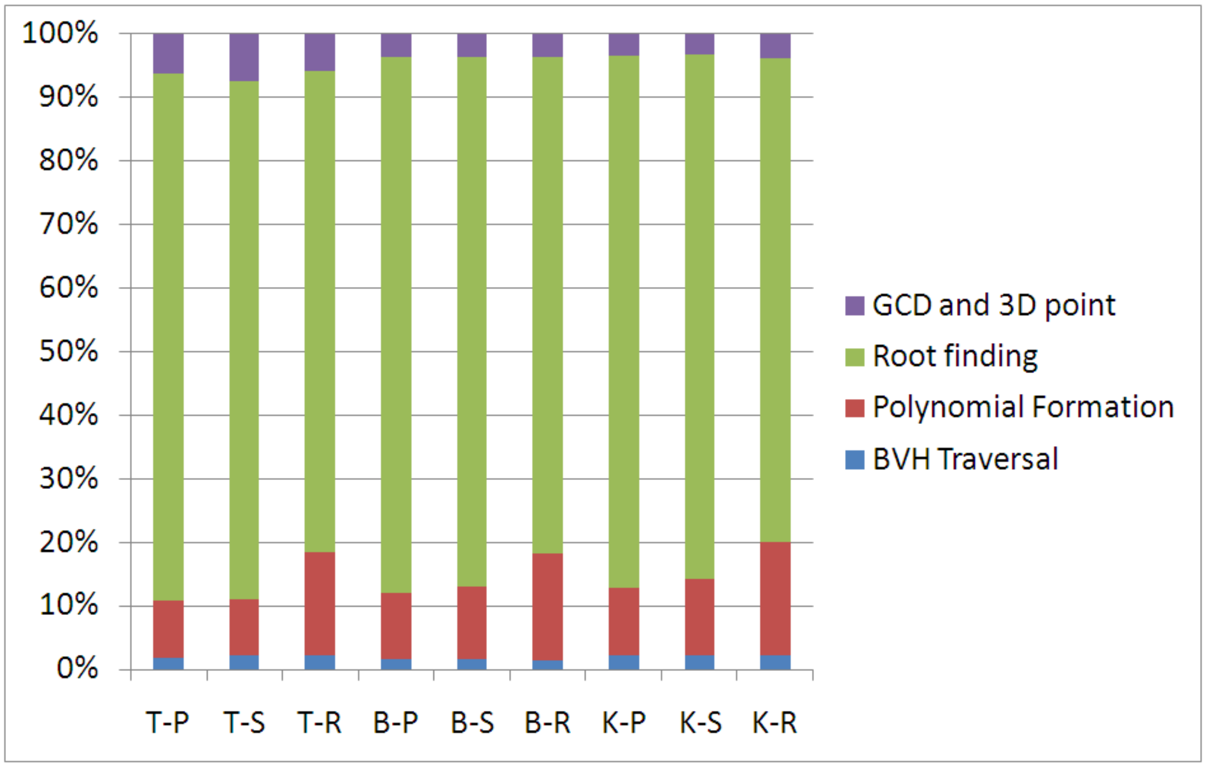


Figure 4.6 Time spent in different kernels of our ray tracing system for Teapot (T), Bigguy (B) and Killeroo (K) models for primary (P), shadow (S) and reflected (R) rays. Y axis shows (Model, Type of ray) tuple.

Table 4.3 shows the time taken in seconds on GTX 280 and GTX 480 for rendering a frame for different models and type of rays. The improvement in the double precision performance on GTX 480 is 4x over GTX 280. Speedup of 3 is achieved on GTX 480 over GTX 280. Figure 4.8 shows the average time taken in microseconds on GTX 280 and GTX 480 to compute a ray patch intersection. It takes on average 3.7 microseconds on GTX 280 and 1.3 microseconds on GTX 480 to find a ray patch intersection. 48KB of memory is used as L1 cache on GTX 480 and 16KB shared memory available is used to store the intermediate values in the kernels. This configuration guarantees maximum coherence between the coefficients in the cache.

Figure 4.7 shows the speedup achieved on GTX 280 and GTX 480 over the equivalent CPU MATLAB implementation on a Intel 2.4GHz dual core processor. GTX 480 gives a speedup of 1000x and GTX 280 gives a speedup of 350x.

Each thread in *laguerre* kernel finds a real or complex root. 20-25% real roots are found in every iteration. Figure 4.9 shows the percentage of threads that finds real root in different warps for finding the first root. Real root deflates the polynomial by one degree. When a complex root is found the polynomial is deflated by a quadratic polynomial obtained by multiplying the complex root and its conjugate thus deflating the polynomial by two degree. The polynomial $p(x_i)$, first derivative $p'(x_i)$

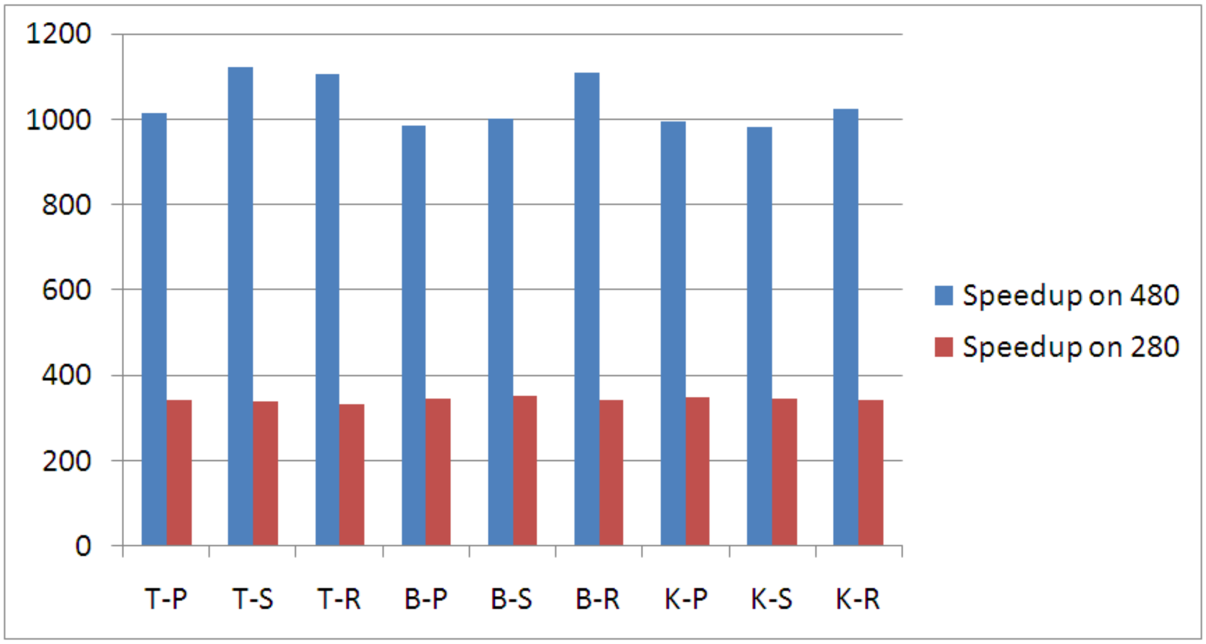


Figure 4.7 Shows the speedup on GTX 280 and GTX 480 over the CPU MATLAB implementation for Teapot (T), Bigguy (B) and Killeroo (K) models for primary (P), shadow (S) and reflected rays (R).

and second derivative $p''(x_i)$ are computed using Werner's method for polynomial evaluation. The highest degree coefficient of all the polynomials is stored continuously in a row. This guarantees less divergence while evaluating the polynomials in further iterations. Divergence occurs only at the end while evaluating the polynomials.

Enough active threads are assigned to keep the GPU busy. The occupancy ranges from 25% to 100%. The performance of the kernels is mostly limited by the number of registers used per thread. We split the kernels wherever possible to reduce the number of registers used. Uncoalesced data reads and writes are avoided by using appropriate and compact data structure.

4.3.2 Memory Requirements and Bandwidth

Memory required to store the BVH in depth first order equals the total number of nodes formed. Each node is represented using 8 values stored in different textures and requires a total of 56 bytes (Section 4.2.1). A working queue is maintained for traversing the BVH. Ray patch intersections to be evaluated are kept in integer arrays. Only the patch control points (16 per patch) are stored in the texture memory. For a ray-patch intersection, 32 A and B coefficients and 19 polynomial coefficients are stored in the global memory. The memory requirements are maximum for the first root and come down as the polynomial is deflated. The total memory requirements are $32 + 19 = 51$ doubles or 408 bytes per ray-patch intersection to be evaluated. For a million pixel display, with 50% empty pixels and an average

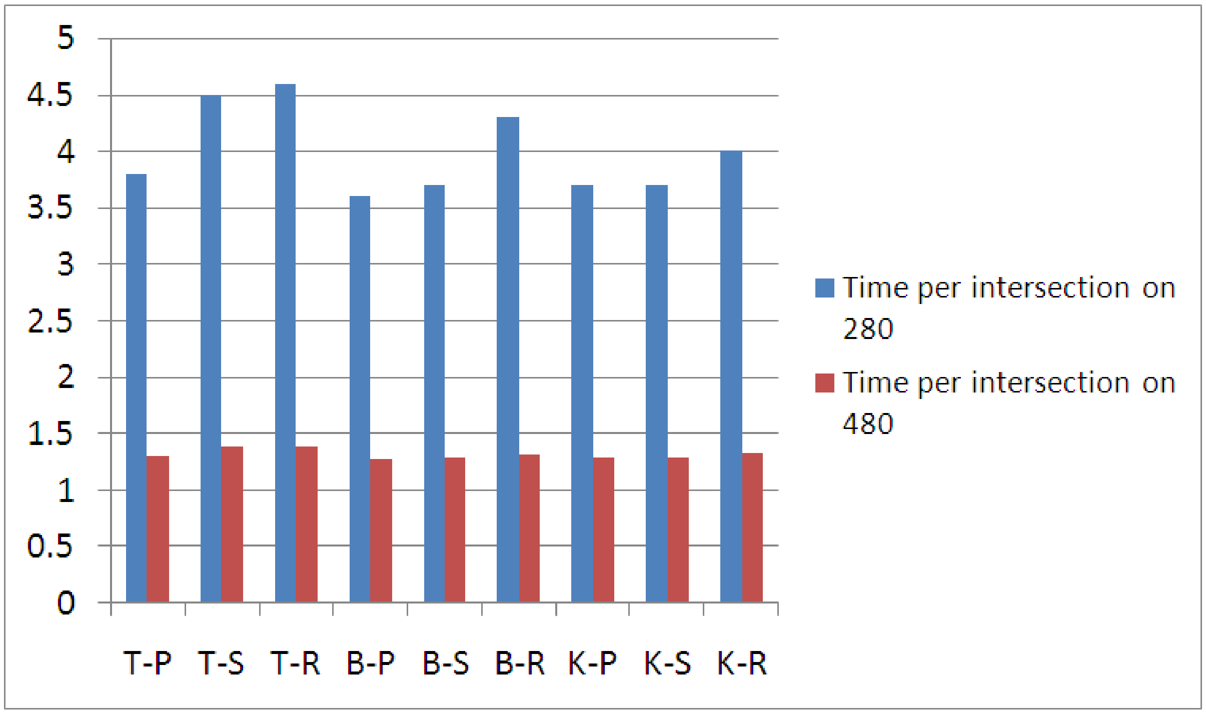


Figure 4.8 Shows the time taken in microseconds on GTX 280 and GTX 480 to compute a ray patch intersection.

ray-patch intersection of 2, the total memory required is 408 MB. However, the ray-patch intersections can be processed in batches, based on available memory, as they are totally independent of one another.

Another related issue is the memory bandwidth used while executing the *laguerre* kernel. The patch coefficients are repeatedly read from the global memory. Multiple reads from the global memory incurs a performance penalty while solving the polynomials making it the most time critical step of our algorithm. The coefficients cannot be stored in shared memory on GTX 280. GTX 480 has a maximum of 48KB shared memory which can be used to store the coefficients. However, storing the coefficients in shared memory than directly reading from global memory incurs a load overhead thereby decreasing the performance.

4.3.3 Strengths and Limitations

Our ray tracing method facilitates ray tracing of dynamic bicubic patches. All the operations except BVH construction (constructed only once on CPU) are performed on GPU. Our algorithm exploits parallelism by dividing the computations into independent tasks. Kernels have low branch divergence and high memory access coherence. The time taken is linear in the number of ray-patch intersections evaluated. Shadow or further bounces do not add to any additional overheads. This gives a way to predict performance based on the rendering and scene complexity. A data structure that can reduce the average number of ray-patch intersections evaluated can boost the performance directly.

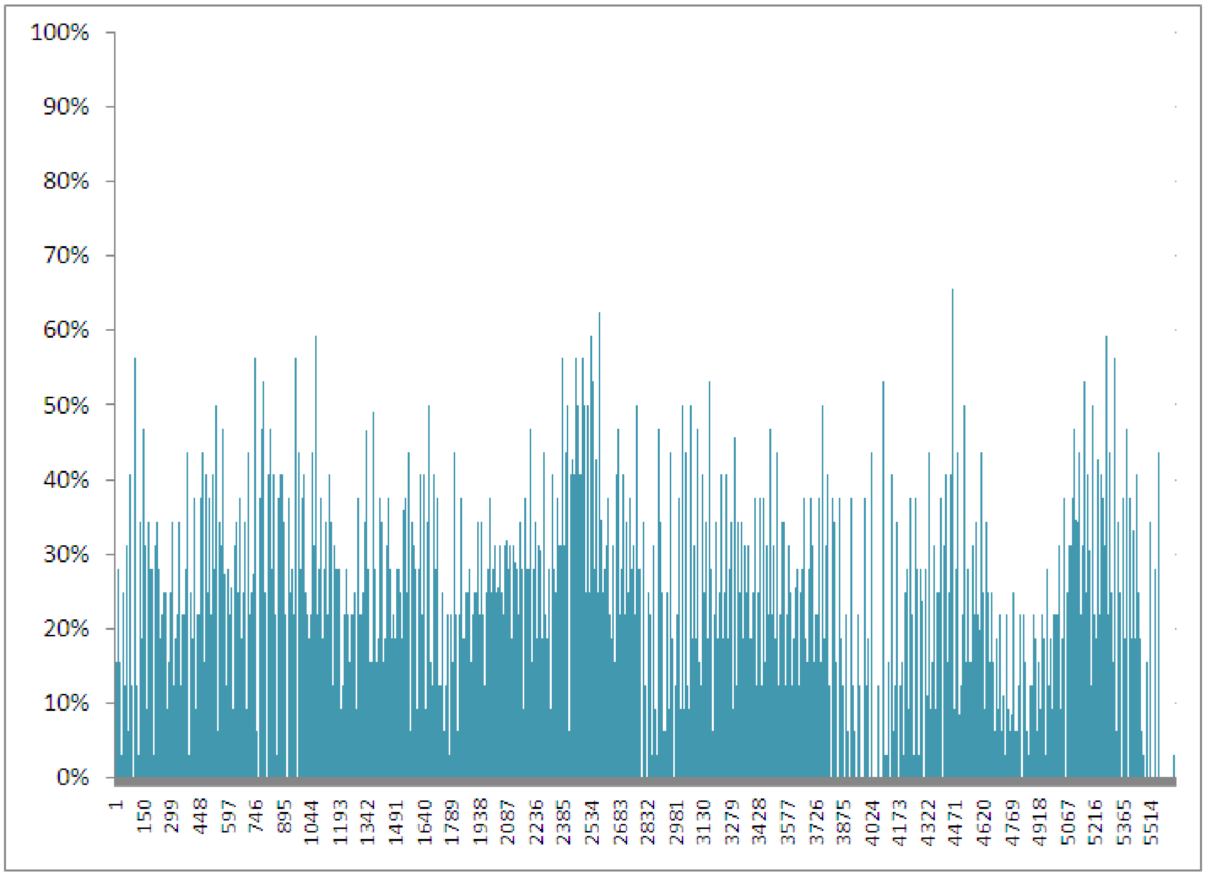


Figure 4.9 Shows the percentage of threads that find real root while finding the first root in different warps.

The primary limitation of our method is its memory usage. However, this can be solved by computing ray-patch intersections in batches. Batch processing has no appreciable impact on the performance if sufficient number of threads run in each to keep the GPU busy, based on our experiments. Two factors that limit the speed of our approach on GTX 280 are double precision computation and registers. Double precision is 4 times faster on GTX 480. Shared memory is used to store the intermediate register values.

Our method can be speeded up further by the use of multiple GPUs, as the ray-patch evaluations are strictly independent. The root finding step will achieve linear speedup. The data transferred across the GPUs can be limited to the relatively light *(rayID, patchID)* pairs.

4.3.4 Comparison with other ray tracing systems

Comparison with other ray-tracing efforts is not meaningful as most are from a different era. Ours is, to the best of our knowledge, the first effort to bring ray tracing of bicubic surfaces to near interactive rates with interactivity a strong possibility in the immediate future. Our approach will be advantageous

Model	Type of Ray	Time (secs) GTX 280	Time (secs) Tesla S1070
Teapot	P	0.211	0.06
	S	0.136	0.035
	R	0.189	0.05
Bigguy	P	0.417	0.109
	S	0.420	0.111
	R	0.614	0.168
Killeroo	P	0.466	0.117
	S	0.508	0.134
	R	0.586	0.152

Table 4.5 Shows the time taken in seconds on GTX 280 and Tesla S1070 GPU to render a frame for different models and type of rays.

in situations which require highly accurate rendering in near-interactive rates. Figures 4.10(a)-4.10(e) and the accompanying video show smooth silhouettes, shadows and reflections on parametric surfaces.

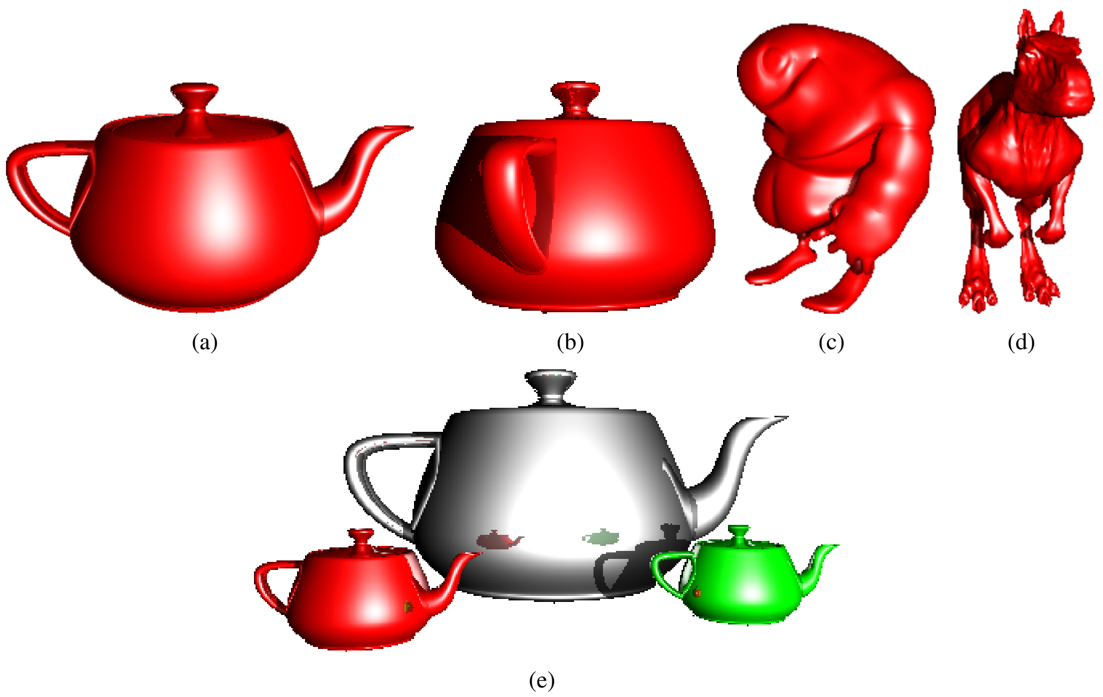


Figure 4.10 Screen shots for different models. (a) Teapot with one level of reflection. (b) Teapot with reflection and shadow. (c) Bigguy with shadow. (d) Killeroo with shadow. (e) Three teapots with shadows and one level of reflection. True normals used for shading guarantees accurate shading and gives high quality images.

Chapter 5

Conclusions

We presented the implementation of two compute intensive problems on the GPU using the CUDA programming language. Computing Singular Value Decomposition of a matrix and direct ray tracing of parametric patches both require enormous amount of floating point operations making them ideal candidates for processing on the GPU. Our GPU implementations exploits the parallelism in the GPU architecture well and achieves high computing performance on them. Ours is, to the best of our knowledge, the first implementation of these problems on the GPU.

Singular Value Decomposition is implemented on the GPU using the two step Golub Reinsch algorithm. The bidiagonalization of a matrix is performed entirely on the GPU using the optimized CUBLAS library to derive maximum performance. We used a hybrid implementation for the diagonalization of the matrix that splits the computations between the CPU and the GPU, giving good performance results. Our implementation using the CUBLAS and CUDA kernels outperforms the optimized CPU implementations available significantly. We show a speedup of upto 60 over the MATLAB implementation and upto 8 over the Intel MKL implementation on a Intel Dual Core 2.66GHz PC on NVIDIA GTX 280.

We are able to compute the Singular Value Decomposition of very large matrices of the order 14K which is impossible on the CPU due to memory limitations. The GPUs are limited to single precision numbers, though that is changing with the newer generations. The error due to the lower precision was less than 0.001% on the random matrices we experimented with. Our approach of using CUDA and the software libraries available with it can be used for solving many other graphics and non-graphics tasks.

We presented an implementation of direct ray tracing of bicubic parametric patches. Our method finds the ray-surface intersections on the GPU by adapting and optimizing Kajiya's method on it. The method guarantees to find the exact point of intersection by finding the roots of a 18 degree polynomial facilitating shading at each pixel using true normals. The time taken is directly proportional to the number of ray-patch intersections that are evaluated and is relatively independent of the number of patches. Thus, primary rays and subsequent bounces can all be handled equally well, providing high quality rendering. Each intersection takes around 3.7 microseconds on GTX 280, irrespective of the number of primitives. We are able demonstrate highly accurate rendering of complex models in near-interactive rates with smooth silhouettes, shadows and reflections on parametric surfaces. We achieve a speedup of 340x on GTX 280 and 990x on GTX 480 over the optimized MATLAB implementation on the AMD Dual CPU Processor of the same algorithm.

The linear dependence of our method on the ray-patch intersections provides a way towards improving the performance in the future. Firstly, faster double precision arithmetic and larger shared memory will improve the basic root finding step proportionately, bringing down the overall time. Both improvements are available in the next generation GPUs. The ray-patch intersections evaluated by traversing the BVH result in independent computations in our method. These may be performed in batches when memory or other resources are not available on a single GPU. They can also be performed in parallel when multiple GPUs are available. Direct ray tracing of Bézier surfaces at interactive rates will be a reality in the near future.

Related Publications

- Sheetal Lahabar and P. J. Narayanan. Singular value decomposition on GPU using CUDA. 23^{rd} IEEE International Parallel and Distributed Processing Symposium. 2009. pp $1-10$.

Bibliography

[1] J. R. Baker. Macrotasking the singular value decomposition of block circulant matrices on the cray-2. Technical report, LBL-27821, Lawrence Berkeley Nat. Lab., 1989.

[2] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí. Evaluation and tuning of the level 3 cublas for graphics processors. In *Proc. of* 22^{nd} *IEEE International Symposium on Parallel and Distributed Processing, Miami, Florida USA*, pages 1–8, 2008.

[3] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí. Glame@lab: An m-script api for linear algebra operations on graphics processors. In *Proc. of Para*, 2008.

[4] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí. Solving dense linear systems on graphics processors. In *Proc. of Euro-Par* 2008 - *Parallel Processing,* 14^{th} *International Euro-Par Conference, Las Palmas de Gran Canaria, Spain*, volume 5168, pages 739–748, 2008.

[5] C. Benthin, I. Wald, and P. Slusallek. Interactive ray tracing of free-form surfaces. In *Proc. of* 3^{rd} *International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction, South Africa*, pages 99–106, 2004.

[6] C. Bobda and N. Steenbock. Singular value decomposition on distributed reconfigurable systems. In *Proc. of* 12^{th} *IEEE International Workshop on Rapid System Prototyping Monterey, CA, USA*, pages 38–43, 2001.

[7] V. Bondhugula, N. Govindaraju, and D. Manocha. Fast svd on graphics processors. Technical report, University of North Carolina, 2006.

[8] A. Cevahir, A. Nukada, and S. Matsuoka. High performance conjugate gradient solver on multi-gpu clusters using hypergraph partitioning. In *Journal of Computer Science . Research and Development*, pages 83–91, 2010.

[9] J. Choi, J. J. Dongarra, and D. W. Walker. The design of a parallel dense linear algebra software library: Reduction to hessenberg, tridiagonal, and bidiagonal form. Technical report, UT-CS-95-275, University of Tennessee, 1995.

[10] M. Christen, O. Schenk, and H. Burkhart. General-purpose sparse matrix building blocks using the nvidia cuda technology platform. *CiteSeerX - Scientific Literature Digital Library and Search Engine, USA*, 2008.

[11] J. Demmel and W. M. Kahan. Computing small singular values of bidiagonal matrices with guaranteed high relative accuracy. *LAPACK Working Note* 3*m ANL-MCS-TM*-110*, Argonne National Laboratory*, 1988.

[12] K. Dickson, Z. Liu, and J. V. McCanny. Programmable processor design for givens rotations based applications. In *Proc. of* 4^{th} *IEEE Workshop on Sensor Array and MultiChannel Processing*, pages 84–87, 2006.

[13] C. Eisenacher, Q. Meyer, and C. T. Loop. Real-time view-dependent rendering of parametric surfaces. In *Proc. of Symposium on Interactive 3D Graphics, Boston, USA*, pages 137–143, 2009.

[14] M. Fatica and W. Jeong. Accelerating matlab with cuda. In *Proc. of High Performance Embedded Computing*, 2007.

[15] N. Fujimoto. Faster matrix-vector multiplication on geforce 8800 gtx. In *Proc. of* 22^{nd} *IEEE International Parallel and Distributed Processing Symposium, LSPP*-402, pages 1–8, 2008.

[16] J. Fung and S. Mann. Openvidia: parallel gpu computer vision. In *Proc. of the* 13^{th} *annual ACM international conference on Multimedia*, pages 849–852, 2005.

[17] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha. Lu-gpu: Efficient algorithms for solving dense linear systems on graphics hardware. In *Proc. of ACM/IEEE SC Conference on High Performance Networking and Computing, Seattle, USA*, page 3, 2005.

[18] M. Geimer and O. Abert. Interactive ray tracing of trimmed bicubic bézier surfaces without triangulation. In *Proc. of Winter School of Computer Graphics*, pages 71–78, 2005.

[19] G. H. Golub and W. M. Kahan. Calculating the singular values and pseudo-inverse of a matrix. In *Journal of the Society for Industrial and Applied Mathematics, Series B, Numerical Analysis* 2, pages 205–224, 1965.

[20] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *Proc. of ACM SIGMOD international conference on Management of data, Chicago, USA*, pages 325–336, 2006.

[21] J. Gunther, S. Popov, H. Seidel, and P. Slusallek. Realtime ray tracing on gpu with bvh-based packet traversal. In *Proc. of IEEE/Eurographics Symposium on Interactive Ray Tracing, Germany*, pages 113–118, 2007.

[22] S. Guntury and P. J. Narayanan. Ray tracing dynamic scenes with shadows on gpu. In *Proceedings of EGPGV*, pages 27–34, 2010.

[23] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *Proc. of High Performance Computing - HiPC* 2007*, 14th International Conference, Goa, India*, pages 197–208, 2007.

[24] K. I. Joy and M. N. Bhetanabhotla. Ray tracing parametric surface patches utilizing numerical techniques and ray coherence. In *Proc. of the* 13^{th} *annual conference on Computer graphics and interactive techniques*, pages 279–285, 1986.

[25] J. T. Kajiya. Ray tracing parametric patches. Technical report, SIGGRAPH, Proceedings of the 9th annual conference on Computer graphics and interactive techniques, 1982.

[26] J. Kalojanov and P. Slusallek. A parallel algorithm for construction of uniform grids. In *Proc. of Proceedings of the Conference on High Performance Graphics*, pages 23–28, 2009.

[27] J. Kruger and R. Westermann. Linear algebra operators for gpu implementation of numerical algorithms. In *Proc. of ACM Transactions on Graphics*, pages 22(3), 908–916, 2003.

[28] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast bvh construction on gpus. *Computer Graphics Forum*, 28(2):375–384, 2009.

[29] R. R. Lewis, R. Wang, and D. Hung. Design of a pipelined architecture for ray/bezier patch intersection computation. In *Canadian Journal of Electrical and Computer Engineering*, volume 28, 2002.

[30] R. Liang, Z. Pan, M. Krokos, M. Chen, J. Bao, and C. Li. Fast hardware-accelerated volume rendering of ct scans. In *Journal of Display Technology*, volume 4, pages 431–436, 2008.

[31] C. Loop and J. Blinn. Real-time gpu rendering of piecewise algebraic surfaces. In *Proc. of SIGGRAPH*, pages 664–670, 2006.

[32] W. Ma, M. E. Kaye, D. M. Luke, and R. Doraiswami. An fpga-based singular value decomposition processor. In *Proc. of Canadian Conference on Electrical and Computer Engineering, Ottawa, Canada*, pages 1047–1050, 2006.

[33] D. Manocha and S. Krishnan. Algebraic pruning: A fast technique for curve and surface intersection. Technical report, TR93-062, University of North Carolina at Chapel Hill, 1994.

[34] K. Moreland and E. Angel. The fft on a gpu. In *Proc. of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 112–119, 2003.

[35] T. Nishita, T. W. Sederberg, and M. Kakimoto. Ray tracing trimmed rational surface patches. In *Proc. of SIGGRAPH*, pages 337–345, 1990.

[36] NVIDIA. Nvidia corporation. *CUBLAS Library*, 2007.

[37] A. Patney and J. D. Owens. Real-time reyes-style adaptive surface subdivision. In *Proc. of ACM SIGGRAPH Asia, Singapore*, pages 1–8, 2008.

[38] Physx. Nvidia technologies. 2008.

[39] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *Journal of ACM Transactions on Graphics*, 21(3):703–712, July 2002.

[40] S. M. Rubin and T. Whitted. A 3-dimensional representation for fast rendering of complex scenes. In *Computer Graphics*, pages 110–116, 1980.

[41] S. Sengupta, M. Harris, Y. Zhang, and J. Owens. Scan primitives for gpu computing. In *Graphics Hardware*, 2007.

[42] Z. Shu. One sided jacobi method on cuda for svd. *Application Research of computers.*, 2007.

[43] J. M. Singh and P.J. Narayanan. Real-time ray tracing of implicit surfaces on the gpu. *IEEE Transactions on Visualization and Computer Graphics*, 99:261–272, 2009.

[44] S. S. Stone, J. P. Haldar, S. C. C. T., H. W. m. W., B. P. Sutton, and Z. P. Liang. Accelerating advanced mri reconstructions on gpus. *J. Parallel Distributed Computing*, 68(10):1307–1318, 2008.

[45] N. Thrane, L. O. Simonsen, and Peter. A comparison of acceleration structures for gpu assisted ray tracing. *Master's thesis, University of Aarhus*, 2005.

[46] S. Tomov and J. Dongarra. Accelerating the reduction to upper hessenberg form through hybrid gpu-based computing. In *Technical Report UT-CS*-09-642*, University of Tennessee*, 2009.

[47] D. L. Toth. On ray tracing parametric surfaces. In *Proc.of Computer Graphics*, volume 19, pages 171–179, 1985.

[48] V. Vineet and P. J. Narayanan. Cuda-cuts: Fast graph cuts on the gpu. In *Proc. of CVPR Workshop on Visual Computer Vision on GPUs*, pages 1–8, 2008.

[49] S. Wang, Z. Shih, and R. Chang. An improved rendering technique for ray tracing bézier and b-spline surfaces. In *Journal of Visualization and Computer Animation 11*, pages 209–219, 2000.

[50] T. Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, 1980.

[51] J. H. Wilkinson and C. Reinsch. Singular value decomposition. *Linear Algebra, vol* 2*. Handbook for automatic computation, Springer-Verlag, Berlin, Heidelberg and New York*, 1971.

[52] C. Woodward. Ray tracing parametric surfaces by subdivision in viewing plane. *Theory and practice of geometric modeling*, pages 273–287, 1989.

[53] Y. Yamamoto, T. Fukaya, T. Uneyama, M. Takata, K. Kimura, M. Iwasaki, and Y. Nakamura. Accelerating the singular value decomposition of rectangular matrices with the csx600 and the integrable svd. In *Proc. of Parallel Computing Technologies,* 9^{th} *International Conference, Russia*, pages 340–345, 2007.

[54] R. Yang and M. Pollefeys. Multi-resolution real-time stereo on commodity graphics hardware. In *Proc. of Computer Vision and Pattern Recognition Conference*, pages 211–217, 2003.

[55] C. Zach, D. Gallup, and J. M. Frahm. Fast gain-adaptive klt tracking on the gpu. In *Proc. of Visual Computer Vision on GPUs workshop*, pages 1–7, 2008.