

Fast, GPU-based Diffuse Global Illumination For Point Models

Fourth Progress Report

Submitted in partial fulfillment of the requirements
for the degree of

Ph.D.

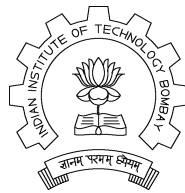
by

Rhushabh Goradia

Roll No: 04405002

under the guidance of

Prof. Sharat Chandran



Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai

August 22, 2008

Acknowledgments

I would like to thank Prof. Sharat Chandran for devoting his time and efforts to provide me with vital directions to investigate and study the problem.

I would also like to specially thank Prekshu Ajmera who supported me all through my work. I would also like to thank Prof. Srinivas Aluru, Iowa State University for his useful suggestions on the construction of octrees on the GPU.

This work was funded by an Infosys Ph.D. fellowship grant. I would also like to thank NVIDIA Pune for providing the graphics hardware and support whenever required. Also, I would like to thank the Stanford 3D Scanning Repository as well as Cyberware for freely providing geometric point models to the research community.

Last but not the least, I would like to thank all the friends and members of ViGiL for their valuable support during the work.

Rhushabh Goradia

Abstract

Advances in scanning technologies and rapidly growing complexity of geometric objects motivated the use of point-based geometry as an alternative surface representation, both for efficient rendering and for flexible geometry processing of highly complex 3D-models. Based on their fundamental simplicity, points have motivated a variety of research on topics such as shape modeling, object capturing, simplification, rendering and hybrid point-polygon methods.

*Global Illumination for point models is an upcoming and an interesting problem to solve. We use the Fast Multipole Method (FMM), a robust technique for the evaluation of the combined effect of pairwise interactions of n data sources, as the light transport kernel for inter-reflections, in point models, to compute a description – **illumination maps** – of the diffuse illumination. FMM, by itself, exhibits high amount of parallelism to be exploited for achieving multi-fold speed-ups.*

Graphics Processing Units (GPUs), traditionally designed for performing graphics specific computations, now have fostered considerable interest in doing computations that go beyond computer graphics; general purpose computation on GPUs, or “GPGPU”. GPUs may be viewed as data parallel compute co-processors that can provide significant improvements in computational performance especially for algorithms which exhibit sufficiently high amount of parallelism. One such algorithm is the Fast Multipole Method (FMM). This report describes in detail the strategies for parallelization of all phases of the FMM and discusses several techniques to optimize its computational performance on GPUs.

The heart of FMM lies in its clever use of its underlying data structure, the Octree. We present a novel algorithm for constructing octrees in parallel on GPUs which will eventually be combined with the GPU-based parallel FMM framework.

*Correct global illumination results for point models require knowledge of mutual point-pair visibility. **Visibility Maps (V-maps)** have been designed for the same. Parallel implementation of V-map on GPU offer considerable performance improvements and has been detailed in this report.*

A complete global illumination solution for point models should cover both diffuse and specular (reflections, refractions, and caustics) effects. Diffuse global illumination is handled by generating illumination maps. Achieving specular effects is a part of the work to be done in future.

Contents

1	Introduction	1
1.1	Point Based Modelling and Rendering	1
1.2	Global Illumination	2
1.2.1	Diffuse and Specular Inter-reflections	4
1.3	Fast computation with Fast Multipole Method	6
1.4	Parallel computations using the GPU	7
1.5	Octrees and FMM	9
1.5.1	Octrees	9
1.5.2	Visibility between Point Pairs	10
1.6	Problem Definition and Contributions	11
1.7	Overview of the Report	11
2	Parallel FMM on the GPU	12
2.1	Fast computation with Fast Multipole Method	12
2.2	Parallel FMM computations on GPU	13
2.3	Implementation Details	14
2.3.1	Upward Pass	15
2.3.2	Downward Pass	17
2.4	Results	20
2.4.1	Quality Comparisons	20
2.4.2	Timing Comparisons	21

3	Octrees	24
3.1	Octrees: Introduction	24
3.2	Parallel Memory Efficient Top-Down Adaptive Octree on the GPU	25
3.2.1	GPU Optimizations	29
3.3	Results	29
4	View Independent Visibility using V-map on GPU	31
4.1	GPU-based V-map Construction	31
4.2	The Visibility Map	32
4.3	V-map Computations on GPU	34
4.3.1	Multiple Threads Per Node Strategy	34
4.3.2	One Thread per Node Strategy	35
4.3.3	Multiple Threads per Node-Pair	35
4.4	Leaf-Pair Visibility	37
4.4.1	Prior Algorithm	37
4.4.2	Computing Potential Occluders	38
4.5	GPU Optimizations	39
4.6	Results	39
4.6.1	Visibility Validation	40
4.6.2	Quantitative Results	41
5	Conclusion and Future Work	43

Introduction

Photorealistic computer graphics attempts to match as closely as possible the rendering of a virtual scene with an actual photograph of the scene had it existed in the real world. Of the several techniques that are used to achieve this goal, *physically-based* approaches (i.e. those that attempt to simulate the actual physical process of illumination) provide the most striking results. The emphasis of this report is on a very specific form of the problem known as *global illumination* which happens to be a photorealistic, physically-based approach central to computer graphics. This report is about capturing interreflection effects in a scene when the input is available as point samples of hard to segment entities. Computing a mutual visibility solution for point pairs is one major and a necessary step for achieving good and correct global illumination effects. Graphics Processing Units (GPUs) have been used for increased speed-ups.

Before moving further, let us be familiar with the terms point models and global illumination.

1.1 Point Based Modelling and Rendering

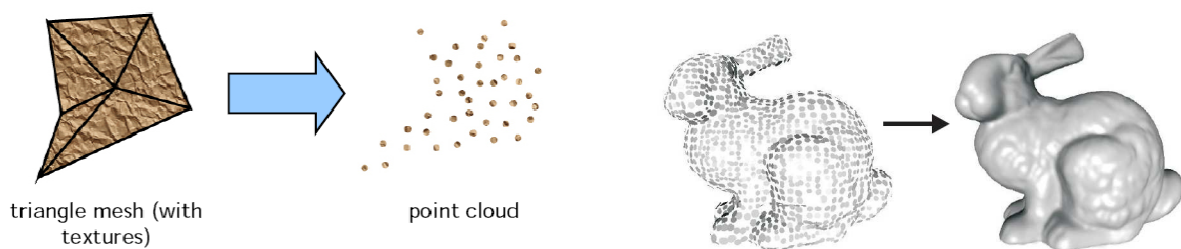


Figure 1.1: Point Model Representation. Explicit structure of points for bunny is visible. Figure on extreme right shows the same bunny with continuous surface constructed

Point models are nothing but a discrete representation of a continuous surface i.e. we model each point as a surface sample representation (Fig 1.1). There is no connectivity information between points. Each point has certain attributes, for example co-ordinates, normal, reflectance, emissivity values.



Figure 1.2: Example of Point Models

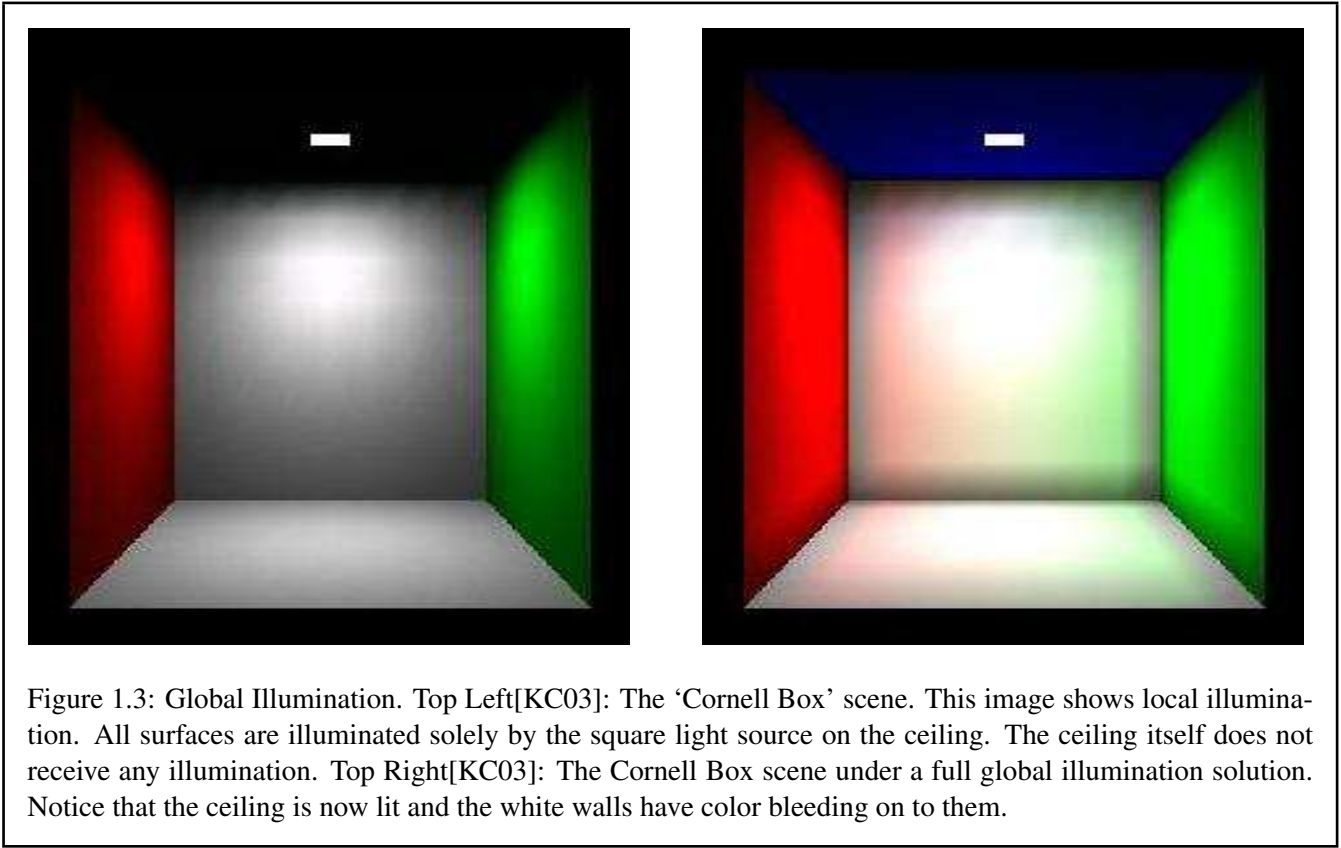
In recent years, point-based methods have gained significant interest. In particular their simplicity and total independence of topology and connectivity make them an immensely powerful and easy-to-use tool for both modelling and rendering. For example, points are a natural representation for most data acquired via measuring devices such as range scanners [LPC⁺00], and directly rendering them without the need for cleanup and tessellation makes for a huge advantage.

Second, the independence of connectivity and topology allow for applying all kinds of operations to the points without having to worry about preserving topology or connectivity [PKKG03, OBA⁺03, PZvBG00]. In particular, filtering operations are much simpler to apply to point sets than to triangular models. This allows for efficiently reducing aliasing through multi-resolution techniques [PZvBG00, RL00, WS03], which is particularly useful for the currently observable trend towards more and more complex models: As soon as triangles get smaller than individual pixels, the rationale behind using triangles vanishes, and points seem to be the more useful primitives. Figure 2.2 shows some example point based models.

1.2 Global Illumination

Local illumination refers to the process of a light source illuminating a surface through direct interaction. However, the illuminated surface now itself acts as a light source and propagates light to other surfaces in the environment. Multiple bounces of light originating from light sources and subsequently reflected throughout the scene lead to many visible effects such as soft shadows, glossy reflections, caustics and color bleeding. The whole process of light propagating in an environment is called Global Illumination and to simulate this process to create photorealistic images of virtual scenes has been one of the enduring goals of computer graphics. More formally,

Global illumination algorithms are those which, when determining the light falling on a surface, take into account not only the light which has taken a path directly from a light source (direct illumination), but also



light which has undergone reflection from other surfaces in the world (indirect illumination).

Figures 1.3 and 1.5 gives you some examples images showing the effects of *Global illumination*. It is a simulation of the physical process of light transport.



Figure 1.4: Grottoes, such as the ones from China and India form a treasure for mankind. If data from the ceiling and the statues are available as point samples, can we capture the interreflections?

Three-dimensional scanned *point models* of cultural heritage structures (Figure 1.4) are useful for a variety of reasons – be it preservation, renovation, or simply viewing in a museum under various lighting conditions. *We wish to see the effects of Global Illumination (GI) – the simulation of the physical process of light transport that captures inter-reflections – on point clouds of not just solitary models, but an environment that consists of*

such hard to segment entities.



Global Illumination effects are the results of two types of light reflections and refractions, namely *Diffuse* and *Specular*.

1.2.1 Diffuse and Specular Inter-reflections

Diffuse reflection is the reflection of light from an uneven or granular surface such that an incident ray is seemingly reflected at a number of angles. The reflected light will evenly spread over the hemisphere surrounding the surface (2π steradians) i.e. they reflect light equally in all directions.

Specular reflection, on the other hand, is the perfect, mirror-like reflection of light from a surface, in which light from a single incoming direction (a ray) is reflected into a single outgoing direction. Such behavior is described by the law of reflection, which states that the direction of incoming light (the incident ray), and the

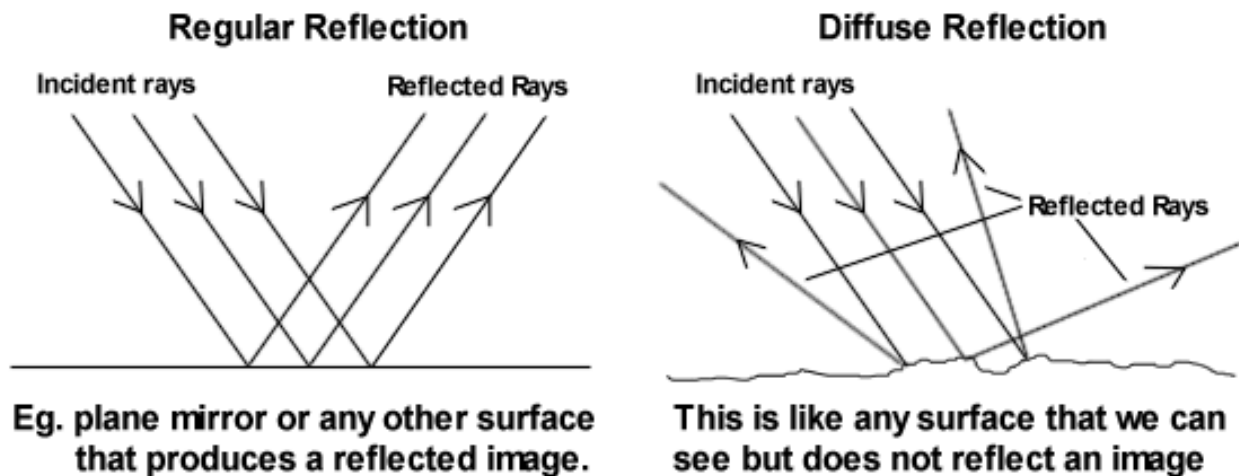


Figure 1.6: Specular (Regular) and Diffuse Reflections

direction of outgoing light reflected (the reflected ray) make the same angle with respect to the surface normal, thus the angle of incidence equals the angle of reflection; this is commonly stated as $\theta_i = \theta_r$.

The most familiar example of the distinction between specular and diffuse reflection would be matte and glossy paints as used in home painting. Matte paints have a higher proportion of diffuse reflection, while gloss paints have a greater part of specular reflection.

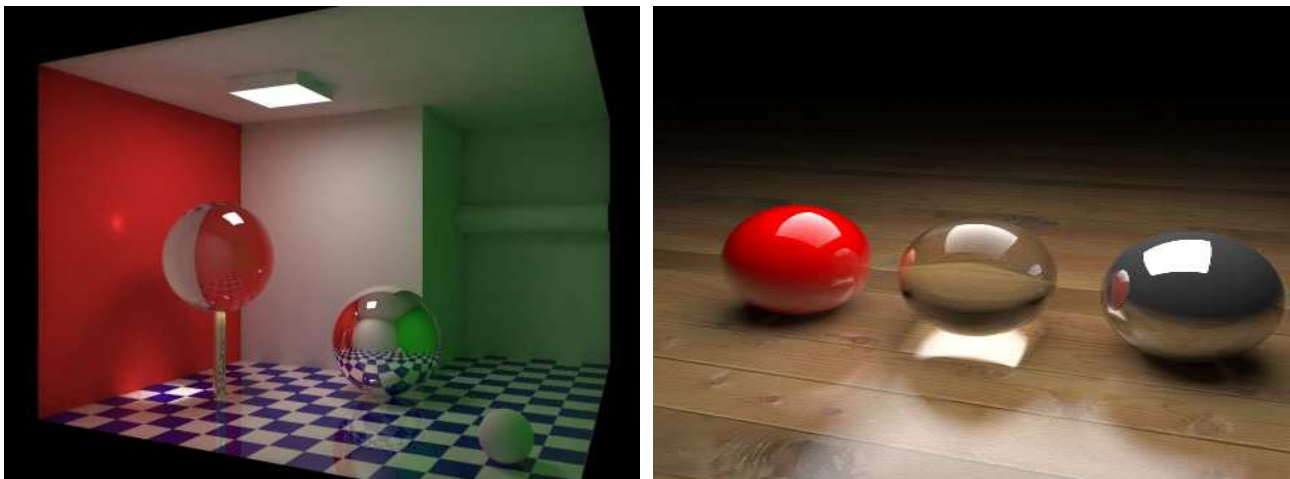


Figure 1.7: Left: Colors transfer (or "bleed") from one surface to another, an effect of diffuse inter-reflection. Also notable is the caustic projected on the red wall as light passes through the glass sphere. Right: Reflections and refractions due to the specular objects are clearly evident

Due to various specular and diffuse inter-reflections in any scene, various types of global illumination effects may be produced. Some of these effects are very interesting like color bleeding, soft shadows, specular

highlights and caustics. *Color bleeding* is the phenomenon in which objects or surfaces are colored by reflection of colored light from nearby surfaces. It is an effect of diffuse inter-reflection. *Specular highlight* refers to the glossy spot which is formed on specular surfaces due to specular reflections. A *caustic* is the envelope of light rays reflected or refracted by a curved surface or object, or the projection of that envelope of rays on another surface. Light coming from the light source, being specularly reflected one or more times before being diffusely reflected in the direction of the eye, is the path traveled by light when creating caustics. Figure 1.7 shows color bleeding and specular inter-reflections including caustics. *Radiosity* and *Ray-Tracing* are two basic global illumination algorithms used for diffuse and specular effects generation (respectively).

Interesting methods like statistical photon tracing [Jen96], directional radiance maps [Wal05], and wavelets based hierarchical radiosity [GSCH93] have been invented for computing a global illumination solution. A good global illumination algorithm should cover both diffuse and specular inter-reflections and refractions, *Photon Mapping* being one such algorithm. Traditionally, all these methods *assume a surface* representation for the propagation of indirect lighting. Surfaces are either explicitly given as triangles, or implicitly computable. The lack of any sort of connectivity information in point-based modeling (PBM) systems now *hurts* photo-realistic rendering. This becomes especially true when it is not possible to correctly segment points obtained from an aggregation of objects (see Figure 1.4) to stitch together a surface.

There have been efforts trying to solve this problem [WS05], [Ama84, SJ00], [AA03, OBA⁺03], [RL00]. Our view is that these methods would work *even better* if fast pre-computation of diffuse illumination could be performed. *Fast Multipole Method* (FMM) provides an answer. We [GKCD07] provided an efficient solution to the above mentioned problem on the CPU. We used a FMM-based radiosity kernel to provide a *diffuse* global illumination solution to any input scene given in terms of points.

1.3 Fast computation with Fast Multipole Method

Computational science and engineering is replete with problems which require the evaluation of pairwise interactions in a large collection of particles. Direct evaluation of such interactions results in $O(N^2)$ complexity which places practical limits on the size of problems which can be considered. Techniques that attempt to overcome this limitation are labeled N-body methods. The N-body method is at the core of many computational problems, but simulations of celestial mechanics and coulombic interactions have motivated much of the research into these. Numerous efforts have aimed at reducing the computational complexity of the N-body method, particle-in-cell, particle-particle/particle-mesh being notable among these. The first numerically-defensible algorithm [DS00] that succeeded in reducing the N-body complexity to $O(N)$ was the Greengard-Rokhlin Fast Multipole Method (FMM) [GR87].

The algorithm derives its name from its original application. Initially developed for the fast evaluation of

potential fields generated by a large number of sources (e.g. the gravitational and electrostatic potential fields governed by the Laplace equation), this method has been generalized for application to systems described by the Helmholtz and Maxwell equations, and to name a few, currently finds acceptance in chemistry[BCL⁺92], fluid dynamics[GKM96], image processing[EDD03], and fast summation of radial-basis functions [CBC⁺01]. For its wide applicability and impact on scientific computing, the FMM has been listed as one of the top ten numerical algorithms invented in the 20th century[DS00]. The FMM, in a broad sense, enables the product of restricted dense matrices with a vector to be evaluated in $O(N)$ or $O(N \log N)$ operations, to a fixed prescribed accuracy ϵ when direct multiplication requires $O(N^2)$ operations. Global illumination problem requires the computation of pairwise interactions among each of the surface elements (points or triangles) in the given data (usually of order $> 10^6$) and thus naturally fits in the FMM framework.

Besides being very efficient ($O(N)$ algorithm) and applicable to a wide range of problem domains, the FMM is also highly parallel in structure. Thus implementing it on a parallel, high performance multi-processor cluster will further speedup the computation of diffuse illumination for our input point sampled scene. Our interest lies in a design of a parallel FMM algorithm that uses static decomposition, does not require any explicit dynamic load balancing and is rigorously analyzable. The algorithm must be capable of being efficiently implemented on any model of parallel computation. We exploit the inherent parallelism of this method to implement it on the data parallel architecture of the GPU to achieve multifold speedups. Further, the same parallel implementation on the GPU, designed for point models, can also be used for triangular models.

1.4 Parallel computations using the GPU

The graphics processor (GPU) on today's video cards has evolved into an extremely powerful and flexible processor. The latest GPUs have undergone a major transition, from supporting a few fixed algorithms to being fully programmable. High level languages have emerged for graphics hardware, making this computational power accessible. NVIDIA's CUDA [CUDA] programming environment offers the familiar C-like syntax which makes programs simpler and easier to build and debug. CUDA's programming model allows its users to take full advantage of the GPU's powerful hardware but also permits an increasingly high-level programming model that enables productive authoring of complex applications. The result is a processor with enormous arithmetic capability [a single NVIDIA GeForce 8800 GTX can sustain over 330 giga-floating-point operations per second (Gflops)] and streaming memory bandwidth (80+ GB/s), both substantially greater than a high-end CPU.

Architecturally, GPUs are highly parallel streaming processors optimized for vector operations. The programmable units of the GPU follow a single instruction multiple-data (SIMD) programming model. For efficiency, the GPU processes many elements in parallel using the same program (kernel). Each element is independent from the other elements, and in the base programming model, elements cannot communicate with each

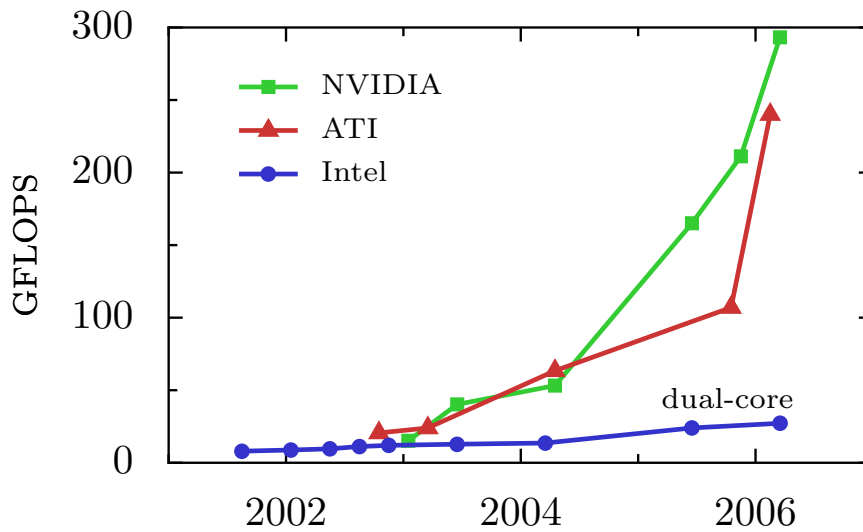


Figure 1.8: GPUs are fast and getting faster [OLG⁺07]

other. All GPU programs must be structured in this way: many parallel elements, each processed in parallel by a single program. Each element can operate on 32-bit integer or floating-point data with a reasonably complete general-purpose instruction set. Elements can read data from a shared global memory (a “gather” operation) and, with the newest GPUs, also write back to arbitrary locations in shared global memory (“scatter”).

With the rapid improvements in the performance and programmability of GPUs, the idea of harnessing the power of GPUs for general-purpose computing has emerged. Problems, requiring heavy computations can be transformed and mapped onto a GPU to get fast and efficient solutions. This field of research, termed as *General Purpose GPU (GPGPU) Computing* has found its way into fields as diverse as databases and data mining, scientific image processing, signal processing, finance etc.

The GPU is designed for a particular class of applications which give more importance to throughput than latency and have large computational requirements and offer substantial parallelism. Many specific algorithms like bitonic sorting, parallel prefix, matrix multiplication and transpose, parallel Mersenne Twister (random number generation) etc. have been efficiently implemented using the GPGPU framework.

One such algorithm which can harness the compute capabilities of the GPUs is parallel Fast Multipole Method. FMM, if divided at a high level, consists of five sequential passes:

1. Octree Construction
2. Interaction List Construction
3. Upward pass on the Octree
4. Downward pass on the Octree

5. Final Summation of Energy

Upward Pass, Downward pass and Final Summation stages are the ones which take more than 97% of the run time. Hence we *first* implemented these 3 stages on the GPU while the Octree Construction and Interaction List Construction stages were performed on the CPU. These will eventually be implemented on GPU as well. We have used the latest Nvidia's G80/G92 architecture GPUs with CUDA as the programming environment.

1.5 Octrees and FMM

The FMM enables an answer to the N-body problem of global illumination to be evaluated in just $O(N)$ or $O(N \log N)$ operations. This is mainly possible because of the underlying hierarchical data structure, the *octree*.

1.5.1 Octrees

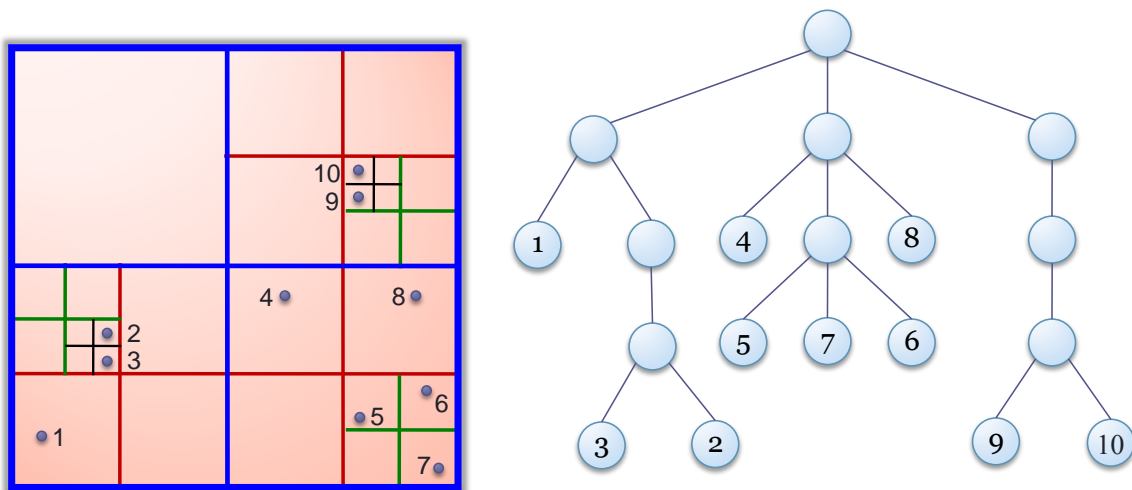


Figure 1.9: A quadtree built on a set of 10 points in 2-D.

Octrees are hierarchical tree data structures that organize multidimensional data using a recursive decomposition of the space containing them. Such a tree is called a *quadtree* in two dimensions, *octree* in three dimensions and *hyperoctree* in higher dimensions. Octrees can be differentiated on the basis of the type of data they are used to represent, the principle guiding the decomposition and the resolution which can be fixed or variable. In practice, the recursive subdivision is stopped when a predetermined resolution level is reached, or when the number of points in a subregion falls below a pre-established constant. This results in the formation of an *adaptive octree*. An example is shown in figure 1.9. In this report we present a novel algorithm for constructing octree in parallel on a GPU based on spatial clustering of points in a top down fashion (ch. 3). This

parallel octree construction algorithm will potentially be combined with the parallel FMM implementation on the GPU.

1.5.2 Visibility between Point Pairs



Figure 1.10: Example showing importance of visibility calculations between points [GKCD07]

Even a good and efficient global illumination algorithm would not give us correct results if we do not have information about mutual visibility between points. For example, in Fig. 1.10, shadows wouldn't have been possible if there wasn't any visibility information. An important aspect of capturing the radiance (be it a finite-element based strategy or otherwise) is an object space *view-independent* knowledge of visibility between point pairs. *Visibility calculation between point pairs is essential as a point receives energy from other point only if it is visible to that point.* But its easier said than done. Its complicated in our case as our input data set is a point based model with *no connectivity* information. Thus, we do not have knowledge of any intervening surfaces occluding a pair of points. Theoretically, it is therefore impossible to determine exact visibility between a pair of points. We, thus, restrict ourselves to **approximate visibility**. We provided a view-independent visibility solution for global illumination for point models in [GKCD07] [Gor07] using Visibility Map (V-map). However, this CPU-based sequential implementation of V-map takes considerable amount of time and hence not very useful for practical applications. We exploit the inherent parallelism in the V-map construction algorithm and attempt to make it work faster with multi-fold speed-ups. Parallel implementation of V-map on GPU [GAC08] offer considerable performance improvements (in terms of *speed*) and has been

detailed in this report.

1.6 Problem Definition and Contributions

After getting a brief overview of the topics, let us now define our problem and contributions.

Problem Definition: *Capturing interreflection effects in a scene when the input is available as point samples of hard to segment entities.*

- Computing a mutual visibility solution for point pairs is one major and a necessary step for achieving good and correct global illumination effects (**Done**).
- Inter-reflection effects include both diffuse (**Done**) and specular effects like reflections, refractions, and caustics. Capturing specular reflections is a part of work to be done in the coming year, which essentially, when combined with the diffuse inter-reflection implementation, will give a complete global illumination package for point models.
- We compute diffuse inter-reflections using the **Fast Multipole Method(FMM)** (**Done**).
- Parallel implementation of visibility and FMM algorithms on Graphics Processing Units(GPUs) so as to achieve speedups for generating the global illumination solution (**Done**).
- Have a parallel octree construction algorithm which could be potentially combined with a parallel FMM algorithm on GPUs (**Done**).

1.7 Overview of the Report

Having got a brief overview of the keyterms, let us review the approach in detail in the subsequent chapters. The rest of the report is organized as follows. Chapter 2 presents an introduction to the FMM algorithm for Radiosity Kernel and our parallel implementation of the same on the GPU. We provide a step by step overview of different kernel functions for each phase of the FMM algorithm along with efficient speed results. We then move on to a parallel octree implementation on GPU in chapter 3. This implementation can be combined with the parallel, GPU-based FMM algorithm. Chapter 4 discusses our GPU-based, parallel V-map construction algorithm and reports multi-fold speed-ups. Finally, chapter 5 summarizes the work done in the course of this year and outlines possible avenues for future research. Capturing specular effects (reflections, refractions, caustics) for point models so as to give a complete global illumination package is a part of work to be done in future.

Parallel FMM on the GPU

2.1 Fast computation with Fast Multipole Method

The FMM, in a broad sense, enables the product of restricted dense matrices with a vector to be evaluated in $O(N)$ or $O(N \log N)$ operations, when direct multiplication requires $O(N^2)$ operations. The Fast Multipole Method [GR87] is concerned with evaluating the effect of a “set of sources” \mathbb{X} , on a set of “evaluation points” \mathbb{Y} . More formally, given

$$\mathbb{X} = \{x_1, x_2, \dots, x_N\}, \quad x_i \in \mathbb{R}^3, \quad i = 1, \dots, N, \quad (2.1)$$

$$\mathbb{Y} = \{y_1, y_2, \dots, y_M\}, \quad y_j \in \mathbb{R}^3, \quad j = 1, \dots, M \quad (2.2)$$

we wish to evaluate the sum

$$f(y_j) = \sum_{i=1}^N \phi(x_i, y_j), \quad j = 1, \dots, M \quad (2.3)$$

The function ϕ which describes the interaction between two particles is called the “kernel” of the system (e.g. for electrostatic potential, kernel $\phi(x, y) = |x - y|^{-1}$). The function f essentially sums up the contribution from each of the sources x_i .

Assuming that the evaluation of the kernel ϕ can be done in constant time, evaluation of f at each of the M evaluation points requires N operations. The total complexity of this operation will therefore be $O(NM)$. The FMM attempts to reduce this seemingly irreducible complexity to $O(N + M)$ or even $O(N \log N + M)$. Three main insights that make this possible are:

1. **Factorization** of the kernel into source and receiver terms
2. Most application domains do not require that the function f be calculated at very high accuracy.
3. FMM follows a **hierarchical structure** (*Octrees*)

Details on the theoretical foundations of FMM, requirements subject to which the FMM can be applied to a particular domain and discussion on the actual algorithm and its complexity as well as the mathematical

apparatus required to apply the FMM to radiosity are available in [KC03][KGC04] and [Gor06]. Five theorems with respect to the core radiosity equation are also proved in this context. *In our case, this highly efficient algorithm is used for solving the radiosity kernel and getting a diffuse global illumination solution.*

Besides being very efficient and applicable to a wide range of problem domains, the FMM is also highly parallel in structure. Considerable research efforts have thus been directed at developing parallel implementations of the adaptive FMM. With rapid improvements in performance and programmability, GPUs have fostered considerable interest in doing computations that go beyond computer graphics and are being used for general purpose computations. GPUs may be viewed as data parallel compute co-processors that can provide significant improvements in computational performance especially for algorithms which exhibit sufficiently high amount of parallelism. FMM is one such algorithm.

Recently, several researchers have reported the use of GPUs, either in isolation or in a cluster to speed up the N-Body problem using direct algorithms (*not FMM*), in which the interaction of every pair of particles is considered [NHP07]. While impressive speedups are reported, these algorithms require $O(N^2)$ memory to utilize $O(N^2)$ available parallelism and is limited by memory bandwidth. Our interest lies in design of a parallel FMM algorithm suited to modern day NVIDIA's G80/G92 GPU architecture using CUDA. We discuss such an algorithm in this chapter. It uses only a static data decomposition and does not require any explicit dynamic load balancing, either within an iteration or across iterations.

2.2 Parallel FMM computations on GPU

The FMM algorithm for Radiosity kernel [KC03] that we use is consistent with the single precision floating point arithmetic on the GPU. While there are softwares to emulate double precision on the GPU, their use is reported to show a decrease of the computational speed by up to 10 times (see the e.g., in [GST05]). And while we will still see an acceleration relative to the CPU, this will not be as dramatic. GPU manufacturers envision in the closest future the release of GPUs with double precision hardware (both ATI and NVIDIA have announced that this feature will be released in mid 2008). In this case, single precision algorithms can be modified accordingly and the fastest methods for high precision computations can be implemented and tested, without writing artificial libraries.

Thus, with the currently existing GPUs, computations with 3, 4, or 5 digit accuracy are appropriate, which cover a broad class of practical needs. FMM achieves the user-defined accuracy using a truncation number p , which essentially signifies number of terms to be considered from the infinitely long series expansion of the kernel function required for separating the source and receiver terms. Computations with a truncation number $p = 8$ and higher using 4 byte floats can produce a heavy loss of accuracy, overflows/underflows (due to summation of numbers of very different magnitude) and cannot be used for large scale problems. On the other

hand, computations with relatively small truncation numbers, like $p = 3, 4,$ and 5 are stable, and can produce the required 3, 4 or 5 digits of accuracy for problems with number of particles of the order $\approx 10^6$, and we focus on this range of truncation numbers in our implementation.

The different parallelization strategies used in our implementation are quite similar to those used by [GD07]. One important thing to note here is that our FMM kernel for radiosity is far more complex which makes the FMM implementation highly difficult. As such the number of terms p in the truncated series expansion of the radiosity kernel is chosen to be 3. It produces results with sufficiently good amount of accuracy (error less than 10^{-4}) acceptable for showing good Global Illumination effects.

2.3 Implementation Details

The Fast Multipole Method consists of the following five phases:

- Octree Construction
- Generating visible interaction lists
- Upward Pass
- Downward Pass
- Final Summation

Our parallel FMM algorithm specifically solves the last three phases (Upward pass, Downward pass and Final summation stage) on the GPU. We assume, as a part of pre-processing step, that we have been given an octree constructed for the input 3D model along with the interactions lists for each of the octree nodes (containing only visible nodes). The octree can be constructed on the CPU or on the GPU (using algorithms discussed in chapter 3), while the visible interaction lists construction happens on the CPU. These 2 phases will eventually be implemented on GPU and combined with the rest of the algorithm.

INPUT: A 3D model with its defined octree and visible interaction lists.

OUTPUT: A Global Illumination solution for the given model.

Our input octree is a long one dimensional array with each level of octree stored one after the other (starting from the *root*). The parent-child relationship is established using the array indices.

We also define four one dimensional arrays, each corresponding to one of the interaction list's type (far, near, multipole, local). The size of each of these arrays is the sum total of the number of nodes in the interaction

lists of every node (for e.g. $\text{size}(\text{far cell list}) = \sum_i \text{size}(\text{far cell list of each node } i)$). The relationship between each node and each of its interaction lists is defined by storing in it the start and the end indices of each of its interaction list in the four global interaction list arrays.

A 3D input point model is stored as a single point array with its necessary attributes (co-ordinates, normal, diffuse surface color, emmissivity, gaussian weights). In case of triangular models, they are converted to points using gaussian quadrature weights theory [KC03].

In the next section we present the different kernel functions implemented for upward, downward and direct summation passes of the FMM algorithm. Each kernel is executed, for different levels of the octree, in CUDA on a one dimensional grid of one dimensional blocks of threads. By default each block contains 128 threads, which was obtained via an empirical study of optimal thread-block size. This study also showed that for good performance the thread block grid should contain not less than 64 blocks for a 16 multiprocessor configuration. If the number of nodes at a level is not a divisor of the block size, only the remaining number of threads is employed for computations of the last block.

2.3.1 Upward Pass

2.3.1.1 Step 1: Generating Multipole Expansion Co-efficients for the Leaves

We need to calculate, in parallel, for each leaf in the octree, the multipole, or S expansion of all particles (sources) contained in the node about the center of the node. The expansions from all particles (sources) in the node are consolidated in a single expansion by summing the coefficients corresponding to each particle (source).

One solution for parallelization here is to assign each thread to handle one source expansion. A drawback of this method is that after generation of expansions they need to be consolidated, which will necessitate data transfer to GPU global memory, unless they form a block of threads handled by one processor. The block size for execution of any subroutine in GPU can be defined by the user, but it is fixed during execution. In the FMM each node may have different number of particles (sources). Thus if a node is handled by a block of threads then threads could be idle, which, of course, reduces the utilization efficiency. GPU speedups compared to the serial CPU code in this case are in range 2-5, which appear to be rather low when compared with the performance of other steps.

The efficiency of this step of calculating multipole expansions substantially increases when we adopt a different parallelization model of having *one thread per node*. In this case one thread performs expansion for each of the sources in the leaf and consolidates these expansions. So one thread produces full multipole expansion for the entire leaf. The advantage of this approach is that the work of each thread is completely independent and so there is no need for shared memory. This perfectly fits the situation when each leaf may

have different number of sources, as the thread that finishes work for a given leaf simply takes care of another leaf, without waiting or need for synchronization with other threads. The disadvantage of this approach is that to realize the full GPU load the number of boxes should be sufficiently large. Indeed, if an optimal thread block size is 128 and there are 16 multiprocessors (so we need at least 64 blocks of threads to realize an optimal GPU load), then the number of nodes should be at least 8192 for a good performance. Note that at maximum leaf level = 4 we have at most $8^4 = 4096$ leaves, and for maximum level = 5 this number becomes $8^5 = 32768$ leaves. So the method can work efficiently only for large enough problems which is the case with us.

1. For every level of octree, starting from the last level of octree, upto the root do
 - (a) Allocate threads equal to number of nodes at the current level
 - (b) For every thread, *in parallel*, Do
 - (c) If current node is a leaf Then
 - i. Calculate the multipole expansion of all particles (sources) contained in the current leaf about the center of that leaf.
 - ii. Consolidate each of these expansions in a single expansion at the current leaf's center.

The time spent for this step usually does not exceed a couple of percent of the overall FMM run time and also the FMM on the GPU is efficient only for relatively large problems.

2.3.1.2 Step 2: Generating Multipole Expansion Coefficients for the Internal Nodes

We need to calculate, in parallel, for each level $l = l_{max} - 1, \dots, 2$, for each node b at that level, the multipole, or S expansion coefficients $M(b)$ due to all particles in that node by translating and aggregating the multipole expansion coefficients of all its children.

We have not yet come upon an optimal strategy for this subroutine. The current version is based on the fact that the resulting multipole, or S expansions for the parent nodes can be generated independently. So, each thread can be assigned one parent node. However, the work load of the GPU in this case becomes very small for low l_{max} and more or less reasonable speedups can be achieved only if several threads are allocated to process a parent node. Since each parent node in the octree has at most 8 children and for each child the multipole-to-multipole, or $S|S$ translation can be performed independently, we used a two dimensional 64×8 blocks of threads and one dimensional grid of blocks. In this setting each parent node was served by 8 threads, with the thread id in y varying from 0 to 7 for identification of the child nodes.

1. For every level of octree, starting from the second last level, upto the root do

- (a) Allocate a 2D grid of threads equal to number of nodes at the current level times 8 (idx is parent and $idy = 0 - 7$ are children nodes. For empty children threads remain idle)
- (b) For every thread, *in parallel*, Do
 - i. If current node is a non-leaf node Then
 - ii. Translate one S coefficient corresponding to the child idy to the center of current node and write to the shared memory based on idx, idy
- (c) Synchronize the threads
- (d) For every thread with $idy = 0$, *in parallel*, Do
 - i. Sum up all the coefficients with the same idx and store in variable sum
 - ii. Write the result back to the global memory corresponding to the current (parent) node.

We do not expect to achieve much speedup in this step. The complexity of depends only on the number of children for each parent node. For non-adaptive structure this number is equal to 8^l for level l . When $l_{max} = 3$ which has only 512 children and 64 parent nodes at most, the efficiency of *translation/per thread* parallelization is low. We have already mentioned earlier that for the current GPU architecture sizes involving 8192 parallel processes or more can be run at full efficiency. Even for $l_{max} = 4$, the full load is not achieved. Thus, for $l_{max} \geq 5$ we expect GPU to gain speedups over the CPU code, which includes computations not only for l_{max} , but for all levels from l_{max} to 3.

Note that the upward pass is a very cheap step of the FMM and normally takes not more than 1% of the total time. This also diminishes the value of putting substantial resources and effort in achieving high speedups for this step.

2.3.2 Downward Pass

We repeat the following steps for each level of the octree, starting from level 2 to the maximum level l_{max} . Downward pass and the final summation phases are combined into a single phase.

2.3.2.1 Step 1: Multipole to Local Translations

For each node, in parallel, translate and aggregate the multipole, or S expansion coefficients of every node in the far cell interaction list of the current node into local, or R expansion coefficients about the current node's center.

1. Allocate threads equal to number of nodes at the current level
2. For every thread, *in parallel*, Do

- (a) For each node A in the far cell list of current node Do
 - i. Translate the multipole expansion coefficients of A into local, or R expansion coefficients about the center of current node.
 - ii. Aggregate each of these expansions in a single expansion at the current node's center.

2.3.2.2 Step 2: Local List Translations

For every node, in parallel, in addition to converting the multipole expansion coefficients of all nodes in the interaction list into local expansion coefficients at the node's center, the local expansion coefficients obtained from the individual particles contained in the local interaction list are also aggregated.

1. Allocate threads equal to number of nodes at the current level
2. For every thread, *in parallel*, Do
 - (a) For each node n in the local list of current node Do
 - i. Obtain the local expansion coefficients obtained from the individual particles contained in n about the center of current node.
 - ii. Aggregate each of these expansions in a single expansion at the current node's center and add up to its existing local expansion coefficients.

2.3.2.3 Step 3: Local to Local Translations

In addition to multipole-to-local and local-list translations, we further need to calculate, in parallel, for each node b at current level, the local, or R expansion coefficients about its center by translating and aggregating the local expansion coefficients from its parent.

1. Allocate threads equal to number of nodes at the current level
2. For every thread, *in parallel*, Do
 - (a) Obtain the local expansion coefficients from its parent node about the center of current node.
 - (b) Add up to the existing local expansion coefficients about current node's center.

This step is very similar to the step 2 of upward pass. For parallelization of this step, the *one thread per node* strategy is used.

2.3.2.4 Step 4: Evaluate Local Expansion at Points

Evaluate, in parallel, the local expansions at individual points in each of the leaves, from the corresponding leaf's center.

1. Allocate threads equal to number of nodes at the current level
2. For every thread, *in parallel*, Do
 - (a) If current node is a leaf Then
 - i. Obtain the radiosity values at individual points in the current leaf, from the local expansion coefficients at current leaf's center.

This step is very similar to the multipole expansion generator discussed above in step 1 of the upward pass. For parallelization of this step, the *one thread per node* strategy is used. The performance of this step is approximately the same as of the multipole expansion generator.

2.3.2.5 Step 5: Near Cell List Translations

For every node in parallel, evaluate the near neighbor interactions (if current node is a leaf) between the points in the current node and every point in each of the nodes in its near cell interaction list. This, and the remaining steps, are a part of the final summation phase.

1. Allocate threads equal to number of nodes at the current level
2. For every thread, *in parallel*, Do
 - (a) If current node is a leaf Then
 - i. For each node n in the near cell list of current node Do
 - A. For all points in current leaf
 - B. For all points in n
 - C. Evaluate the radiosity interaction directly
 - D. Add up the evaluated value to the existing radiosity values of the points

2.3.2.6 Step 6: Multipole List Translations

For every node, in parallel, in addition to evaluating the near neighbors and local expansion coefficients at each particle, we also evaluate the multipole expansion coefficients of all nodes in the multipole interaction list.

1. Allocate threads equal to number of nodes at the current level

2. For every thread, *in parallel*, Do

(a) If current node is a leaf Then

i. For each node n in the multipole list of current node Do

A. Translate the multipole expansion coefficients of n from its center to individual points of current leaf

B. Add up the evaluated value to the existing radiosity values of the points

This scheme for downward pass is efficient on CPU, but may not be the best for GPU, where the cost of one random access to global memory is equal up to 150 float operations, and instead of reading precomputed data GPU may rather compute them at higher rate. Moreover, for low l_{max} the GPU kernel may even run slower than the serial CPU!. However, even in this case it is not recommended to switch between the CPU and GPU, since such a switch involves the slowest memory copying process (CPU-GPU), and if possible all data should stay on the GPU global memory. *Performance improves a lot* as the size of the problem and, respectively, the maximum level of the octree increases and for $l_{max} = 8$ the time ratio reached 20 or so.

In the above steps we explained the kernel functions implemented for the upward, downward and final summation passes of the FMM algorithm. Each point has 3 primary colors associated with it viz. Red, Green and Blue. Hence, we run all the above steps thrice, corresponding to each color. Further, we converge to the final Global Illumination solution by iterating all the above steps (for all colors) three times (Empirical evidences prove that the solution converges to a good extent in 3 iterations).

2.4 Results

Now we compare the Fast Multipole Method on CPU and GPU based on the visual quality of output result showing global illumination effects and running time.

2.4.1 Quality Comparisons

First we compare the results obtained on the GPU for quality with the corresponding implementation on the CPU. To compare CPU and GPU implementations we use 3-d point models of bunny and Ganesha in a Cornell room each having four light sources on the ceiling. As we can see in Fig. 2.1 the *CPU-GPU results look identical*. Global illumination effects like color bleeding and soft shadows are also clearly visible. Note that for FMM the visual quality of result does not depend on the kind of GPU used (NVIDIA 8800 GTS or Quadro FX 3700). The GPU should just support CUDA and have enough memory (> 256 Mbs).

Note that we converge to the final Global Illumination solution shown in Fig. 2.1 by performing the FMM algorithm (Upward and then Downward passes) for all colors (RGB) three times. We can see that the solution

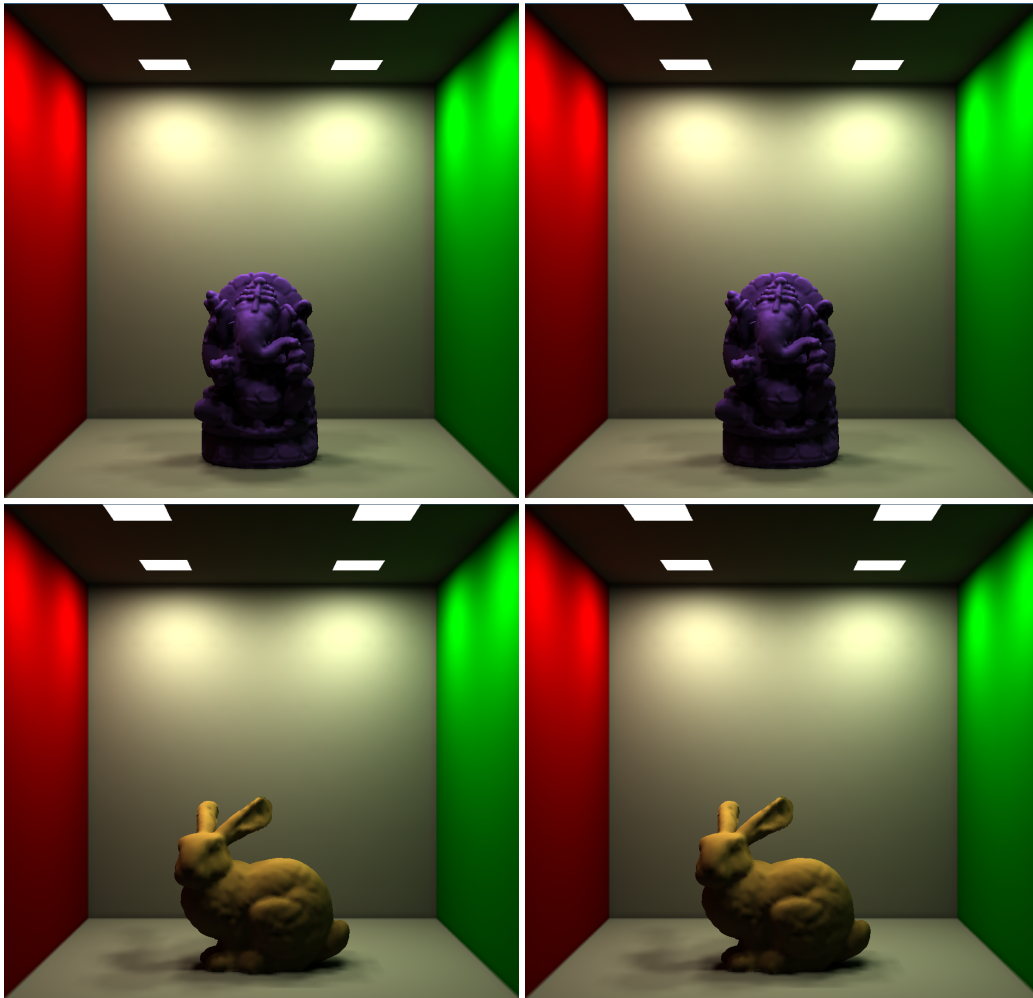


Figure 2.1: Top Left: A Cornell Room with the Ganesha’s point model on CPU. Top Right: Corresponding GPU result. Bottom Left: A Cornell Room with the Bunny’s point model on CPU. Bottom Right: Corresponding GPU result. Both the results assume 50 points per leaf.

converges to a good extent in 3 iterations.

2.4.2 Timing Comparisons

The timing calculations are done on a machine having a dual core AMD Opteron 2210 processor with 2 Gbs of RAM, NVIDIA GeForce 8800 GTS with 320 Mbs of memory and Fedora Core 7 (x86_64) installed on it. The total time taken by the upward and downward passes of the FMM algorithm for all 3 iterations and all 3 colors RGB is shown in the results below. The time taken by each iteration is approximately same.

2.4.2.1 Upward Pass (for all 3 iterations, 3 colors and $p=3$)



Figure 2.2: Point models rendered with diffuse global illumination effects of color bleeding and soft shadows. Pair-wise visibility information is essential in such cases. Note that the Cornell room as well as the models in it are input as point models.

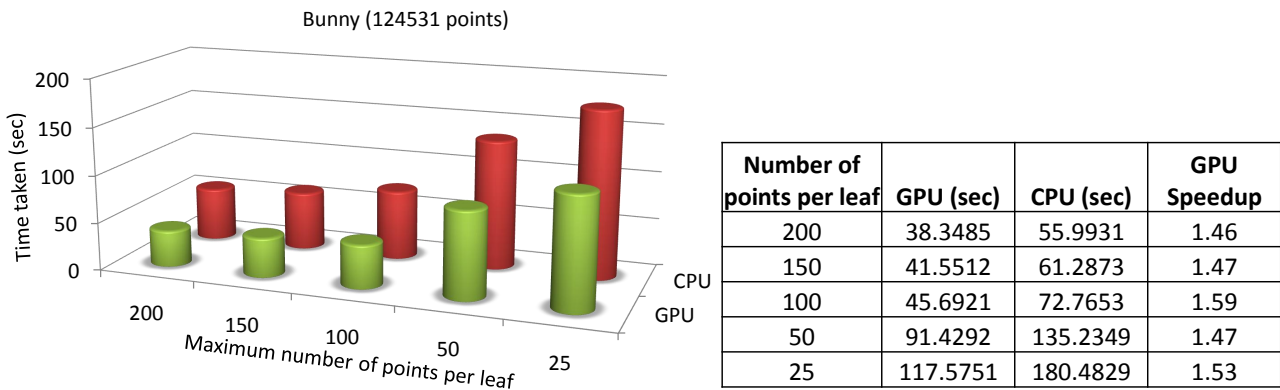


Figure 2.3: FMM Upward Pass : Bunny with 124531 points

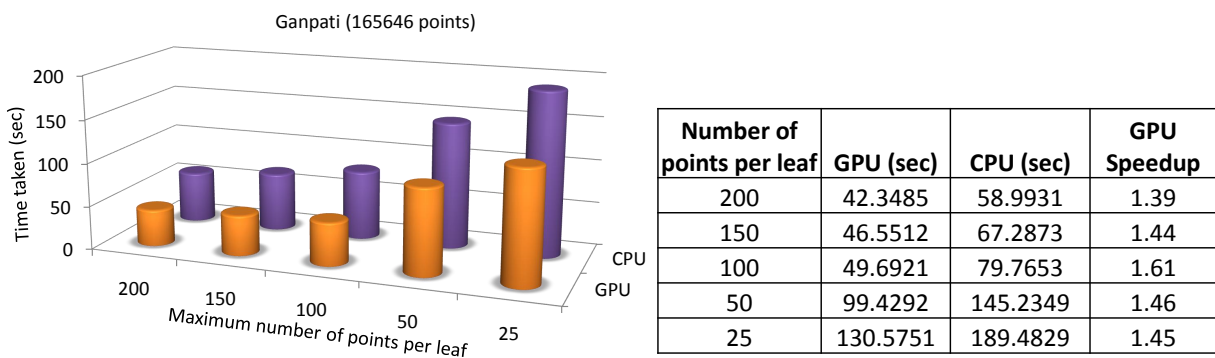
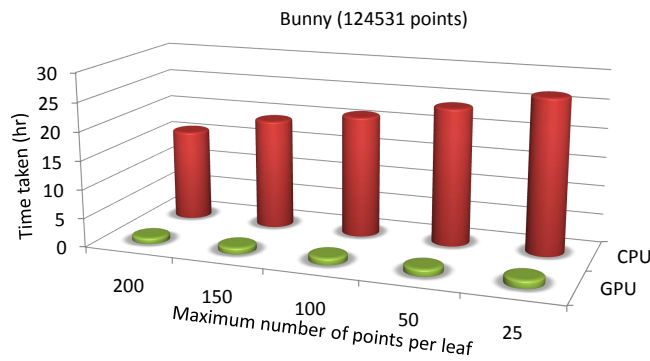
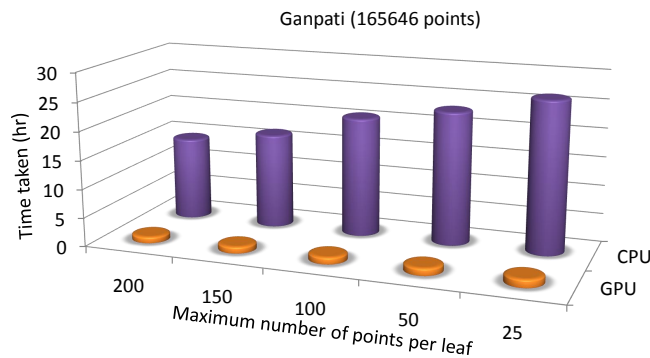


Figure 2.4: FMM Upward Pass : Ganpati with 165646 points



Number of points per leaf	GPU (hr)	CPU (hr)	GPU speedup
200	1.01	15.96	15.8
150	1.09	19.18	17.6
100	1.16	21.11	18.2
50	1.21	23.81	19.5
25	1.30	25.87	19.9

(a)



Number of points per leaf	GPU (hr)	CPU (hr)	GPU Speedup
200	1.11	14.54	13.1
150	1.16	16.58	14.3
100	1.21	20.81	17.2
50	1.28	23.15	18.1
25	1.41	26.37	18.7

(b)

Figure 2.5: Downward Pass (a) Bunny with 124531 points (b) Ganpati with 165646 points

2.4.2.2 Downward Pass (for all 3 iterations, 3 colors and p=3)

Thus, we see that the GPU outperforms the CPU by factors of 13-20 in the downward pass of the FMM algorithm. We also see that the upward pass of the FMM algorithm consumes less than 1% of the time taken by the downward pass. Thus, the speedup achieved in the upward pass does not play an important role in the overall FMM speedup. *The overall speedup achieved is the speedup achieved in the downward pass.*

Octrees

Octree is one of the numerous hierarchical data structures, based on recursive domain decomposition, that are used for representing spatial data. Its development has been motivated to a large extent by a desire to save storage by aggregating data having identical or similar values. However, the savings in execution time that arise from this aggregation are often of equal or greater importance. Using octree as the base hierarchical structure for FMM implementation is one of the important reasons for its success with respect to pulling down the run-time complexity of the algorithm. We discuss, in this chapter, a parallel octree construction implementation on the GPU which can eventually be combined to parallel FMM implementation on GPU. It is a top-down approach using spatial clustering of points. Note that the octree construction phase does not consume more than 1% of the overall FMM run-time and hence it is insignificant if the octree is implemented on the CPU or the GPU.

3.1 Octrees: Introduction

Octrees can be differentiated on the basis of the type of data they are used to represent, the principle guiding the decomposition and the resolution which can be fixed or variable. Let us look at a top-down method to construct octrees.

Consider a hypercube enclosing n multidimensional points. The domain enclosing all the points forms the root of the octree. This is subdivided into 2^d subregions of equal size by bisecting along each dimension. Each of these regions that contain at least one point is represented as a child of the root node. The same procedure is recursively applied to each child of the root node terminating when a subregion contains at most one point. The resulting tree is called a region octree to reflect the fact that each node of the tree corresponds to a non-empty subdomain. An example is shown in Fig. 3.1. In practice, the recursive subdivision is stopped when a predetermined resolution level is reached, or when the number of points in a subregion falls below a pre-established constant. This forms an *adaptive octree*. A *non-adaptive octree* is formed when the maximum

resolution is fixed.

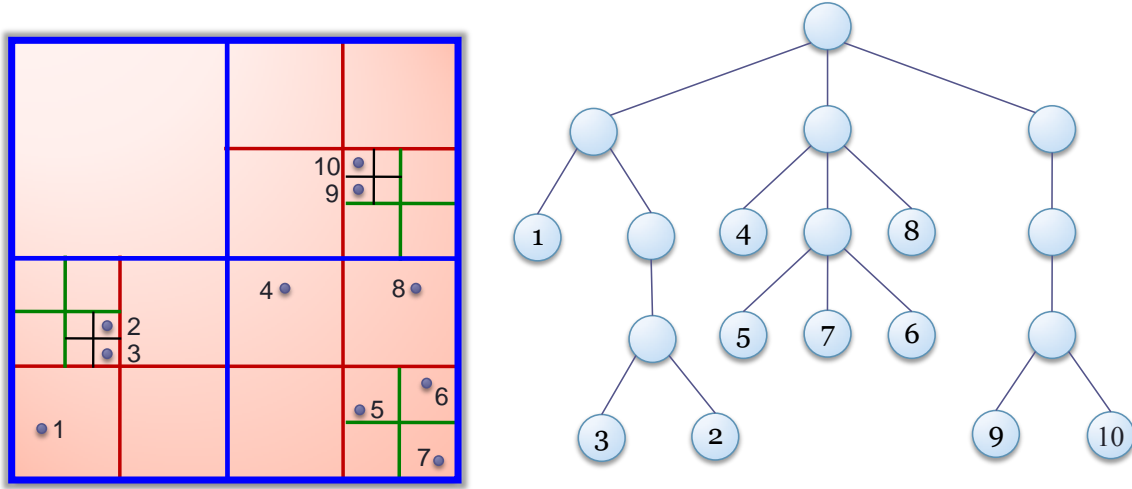


Figure 3.1: A quadtree built on a set of 10 points in 2-D.

3.2 Parallel Memory Efficient Top-Down Adaptive Octree on the GPU

INPUT: n points belonging to some 3-d domain, maximum points per leaf of an octree.

OUTPUT: Octree represented using L arrays one for each level with parent-child relationships established.

PROBLEM SETTING: For brevity we assume that the data of interest is available as points in a domain. For eg., these could be the points belonging to some 3-D point model of say, a Stanford bunny, or might represent centroids of triangular patches of some 3-D mesh. We make no assumption on the number of points in the model. However, memory limitations of the GPU might possibly result in multiple points within a cell.

This is a *top-down* parallel adaptive octree generation algorithm. The intuition behind this algorithm is to *iteratively cluster* the points belonging to the same node together, starting from the root till we construct the leaves. As each cluster generation is independent of the other, on each iteration, the cluster generation process can be parallelized. An example to explain the same is shown in Fig. 3.2.

It is implemented using the latest NVIDIA GPUs featuring support for *atomic operations* like atomic add/subtract, atomic increment/decrement, atomic max/min etc [CUDA]. An operation is atomic in the sense that it is guaranteed to be performed without interference from other threads.

Here in Fig. 3.2(a), we see an array of points enclosed in some space. We now try to cluster these points based on their locations with respect to nodes of the octree. Assume the space enclosing the points to be the root of the octree. We now divide the root into its children as shown in Fig. 3.2(b). Here we see that points 1, 3, 8 belong to child N_1 of root, points 7, 9 belong to child N_2 and so on. Hence we swap these points accordingly

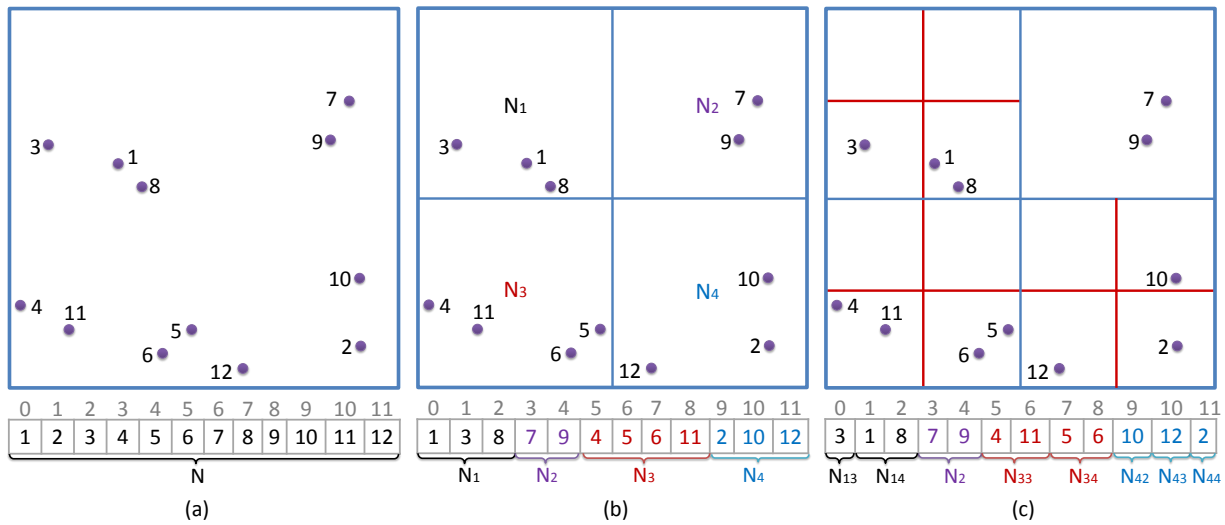


Figure 3.2: Spatial Clustering of Points

in the array (Implementation fact: we swap the pointers, not the actual data) so that they cluster together as shown in Fig. 3.2(b). We iteratively repeat this process till we have less than some pre-defined points (2 as in Fig. 3.2) in a node and term it as a leaf. Fig. 3.2(c) shows this recursion and the final point array after all the swaps. The octree nodes generated now just need to store the *start and end bounds* defining their cluster of points in the point array. For eg., node N_1 , as shown in Fig. 3.2, stores its *start bound* as array location 0 and end bound as 2, while node N_4 stores them as 9 and *end bound* as 11. Further, node N_{34} , child on N_3 , stores bounds as 7 and 8 and so on for all the nodes.

This intuition on building the octree can easily be extended to a parallel algorithm. As we can see, we move down level by level. Thus, on every iteration, we swap the points and create new *partitions* in the point array. An important thing to note is that *all these partitions can be generated independently of one another and thus can be parallelized*. Hence, initially for the *root* we have a single thread generating 8 new partitions corresponding to 8 of its children. We then have *maximum* of 8 threads generating maximum of 64 new partitions corresponding to 64 grand-children on the *root* (*maximum* of 8 because some nodes might turn to leaves and won't be divided further and their thread stops); then *maximum* of 64 threads and so on. *The degree of parallelism increases as we move down to the greater depths of the octree generation process.*

Having got a brief overview, we now present the algorithm with some implementation details.

INPUT: n points belonging to some 3-d domain

OUTPUT: Octree with parent-child relationships established

1. Read points in an array P of size n .

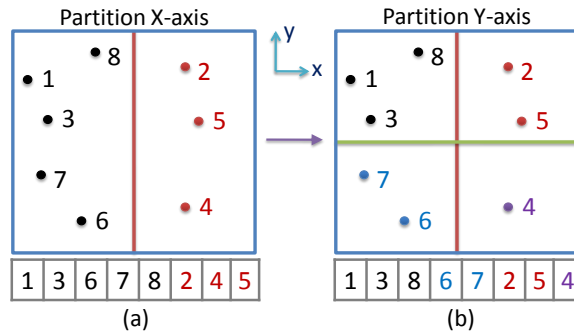


Figure 3.3: Spatial Clustering of Points

2. Initialize the *root* node of the octree as containing all points of P . Set the bounds defining cluster of points belonging to the root as 0 and $n - 1$.
3. Now loop on current step
 - (a) Allocate threads equal to the number of partitions. ($Num_Threads = 1$ initially for the root and then increases as we iterate)
 - (b) For every thread, *in parallel*, do
 - i. STOP the thread if the current partition is a leaf.
 - ii. ELSE, create 8 new partitions and 8 new octree nodes. Record the respective partition bounds in the nodes created. To create 8 new partitions, we first divide the current partition along the longest axis (x , y or z) and swap the points belonging to one side of the partition with another as shown in Fig. 3.3(a). We then repeat the same process and divide the 2 new partitions along the second longest axis, as in Fig. 3.3(b), and finally along the third. For purpose of illustration, we have shown partitioning a quadtree instead of an octree.
 - (c) STOP the loop when every thread running encounters a leaf and hence no new partitions are generated.

Here are some of the implementation details.

1. MEMORY ALLOCATION: Every iteration of the algorithm creates many new partitions and octree nodes. We need to allocate memory to store this newly generated information. The problem arises here because GPU doesn't allow for dynamic memory allocation. One way to get around this is to allocate maximum possible memory. But this eventually leads to storing the whole tree ($8^0 + 8^1 + 8^2 + \dots + 8^l$) till level l , and there by wasting a huge amount of memory [AGCA08].

A better solution is to *pre-compute*, in the current iteration, the number of nodes which will be generated at the next iteration. We can thus allocate *only the desired memory* before the next iteration starts.

This can be achieved by setting a global *Num_Leaves* variable. This will be used to count the leaves which are formed in the current iteration and hence these won't be partitioned further. Every thread, after creating the partitions, checks whether any of the 8 partitions is a leaf or not. If YES (For eg. 2 of the 8 are leaves) it increments the global *Num_Leaves* variable by those many leaf-counts (For eg. $Num_Leaves+=2$). We use *atomic increments* available in latest G92 architecture of GPU so that every thread increments it by a desired amount and the final outcome is the total number of leaves at current level. The new global memory allocated then would be $(Nodes\ at\ current\ level - Num_Leaves)*8$. 8 here refers to the 8 new partitions generated by each thread.

2. INDEXING MECHANISM: We know that the partitions generated by the iteration will be partitioned further in the next iteration, provided they don't represent a leaf. Thus, there might be threads in between which are stopped as they represent a leaf. Hence, a proper indexing and offset mechanism must be installed so that the threads know where to write the new partitions in the global array, as shown in Fig. 3.4.

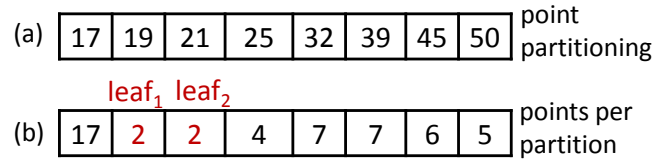


Figure 3.4: Partition Array of a Node

We have a $Node_n$ with say 50 points. Let $Node_1, Node_2, \dots, Node_8$ be the children of $Node_n$. As in Fig. 3.4, points 1 – 17 belong to $Node_1$, 18 – 19 to $Node_2$ and so on. Let us assume that a leaf is formed when the node has 3 or less points. Thus $Node_2$ and $Node_3$ are leaf nodes. Hence the memory allocated for next iteration is $(8 - 2) * 8 = 48$ for 48 new partitions. So $thread_1$ will write its 8 partitions at locations 1 – 8, $thread_2$ at 9 – 16 and so on. But since $Node_2$ and $Node_3$ are leaves, $thread_4$ will now write the new partitions at locations where $thread_2$ was suppose to write i.e. 9 – 16 and the remaining threads will follow the offset. So every node must know how many leaves are present before itself in the array. One can find this using a simple parallel prefix sum [CUDb][HSO07][Har] on the array.

Thus, the new location to write new partitions is, say for node A is $(original\ location\ to\ write - 8 * Number\ of\ leaves\ before\ A)$. This gives a unique indexing for every thread and memory is allocated only as much as desired.

3. PARENT-CHILD: This relationship is established while partitioning itself as every child partition is generated from its parent, thereby giving us our desired octree.

DISCUSSION: Maximum memory required for implementation is just equal to storing non-empty octree nodes, very less compared to [AGCA08]. However, it loses w.r.t time when compared to [AGCA08] but is very fast compared to the CPU implementation. As it performs data-dependent clustering, octree generated can be used in example application areas like color quantization, collision detection, visibility determination. Parent-Child, containment, and neighbor-finding are some example queries which it can answer. Answer to these queries is required while performing FMM.

3.2.1 GPU Optimizations

To improve the GPU kernel's performance, we utilize several optimization techniques enlisted below.

1. **LOOP UNROLLING:** Loop unrolling achieves a modest speedup compared to our initial implementation. We found that especially the loops with global memory accesses (as it is the case in our algorithm) in them benefit a lot from unrolling.
2. **OPTIMAL THREAD AND BLOCK SIZE:** Each thread block must contain 128 – 256 threads and every thread block grid no less than 64 blocks for a 16 multiprocessor configuration for optimal performance, which was obtained via an empirical study. We made sure this was achieved. If the number of nodes at a level is not a divisor of the block size, only the remaining number of threads is employed for computations of the last block.
3. **OPTIMAL OCTREE DEPTHS:** The efficiency of our kernel is substantial as every thread works on an independent partition. The advantage achieved is that the work of each thread is completely independent eliminating the need for any shared memory. This perfectly fits our situation where each thread on finishing its work or on making an early exit (say by encountering a leaf) simply moves on to a new partition without the need for synchronizing with other threads. Note that to realize the full GPU load the number of nodes to be considered should be sufficiently large. Indeed, if an optimal thread block size is 128 and there are 16 multiprocessors (so we need at least 64 blocks of threads to realize an optimal GPU load), then the number of nodes should be at least 8192 for a good performance. If the octree is built till depth 4, we have at most $8^4 = 4096$ leaves and for depth 5 this number becomes $8^5 = 32768$. Thus, GPU works to its full potential at a small enough octree depths and the efficiency increases as we move down to greater depths (≥ 6). GPU totally out-performs the CPU for depths ≥ 6 .

3.3 Results

In this section we compare our implementation of octree on the GPU with the corresponding implementation on the CPU based on running time. We use 3-d points models of bunny and Ganesha in a Cornell room as

inputs to create the octree.

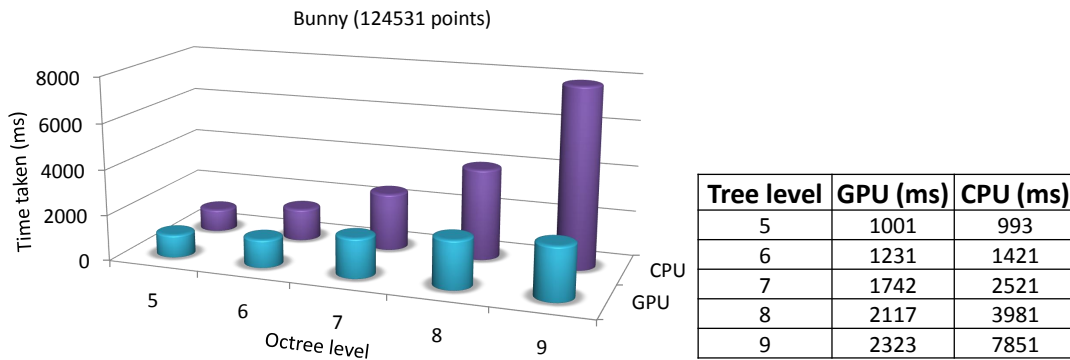


Figure 3.5: Top-Down Octree Construction (Bunny 124531 points) (sec. 3.2)

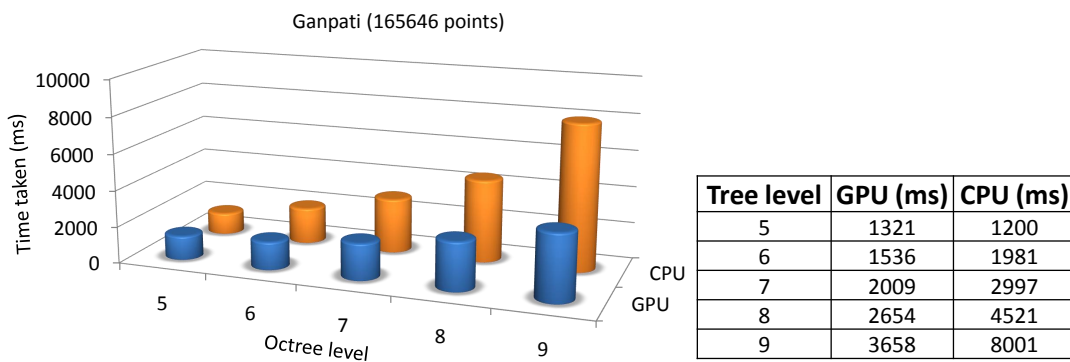


Figure 3.6: Top-Down Octree Construction (Ganpati 165646 points) (sec. 3.2)

We see that the GPU outperforms the CPU at higher levels. We implemented the top-down GPU-based parallel octree construction algorithm using the latest NVIDIA GPUs featuring support for atomic operations like atomic increment/decrement etc. These GPUs have G92 architecture. The machine used has a Intel Core 2 Duo 1.86 GHz with 2 Gbs of RAM, NVIDIA Quadro FX 3700 with 512 Mbs of memory and Fedora Core 7 (x86_64) installed on it.

View Independent Visibility using V-map on GPU

In point-based graphics [DTG00, KTB07, GD98, MGPG04], a scene represented as points is to be rendered from various viewpoints keeping global illumination in mind [DYN04]. That is, a point may be illuminated by other points, which in turn are illuminated by still other points, and so on. Inter-reflections in such scenes requires knowledge of visibility between point pairs. Computing visibility for points is all the more difficult (as compared to polygonal models), since we do not have any surface or object information. The visibility function is highly discontinuous and, like the BRDF, does not easily lend itself to an analytical FMM formulation. Thus the nature of this computation is $\theta(n^2)$ for n primitives, which depends on the geometry of the scene. A visibility preprocessing step for finding view independent mutual point-pair visibility is useful before the costly rendering equation is solved. The visibility map (V-map) data structure (see sec. 4.2) was introduced for this purpose. The basic idea here is to partition the points in the form of an octree. When large portions of a scene are mutually visible from other portions, a visibility link is set so that groups of points (instead of single points) may be considered in discovering the precise geometry-dependent illumination interaction. Ray shooting and visibility queries can also be answered in sub-linear time using this data structure.

4.1 GPU-based V-map Construction

A sequential implementation of the V-map [Gor07] [GKCD07] takes hours (for octrees with height 8 or more). Reducing the octree height (to say 7 or below) in the interests of time yields unacceptable results (Fig. 4.1).

Our work is concerned with computing the V-map data structure on the GPU. Specifically,

1. If a black-box “kernel” is available to compute the relationships (e.g., gravitational interaction, point-pair visibility), we show how a hierarchical data structure can be built efficiently (see sec. 4.3) on the GPU using CUDA. For example, our V-map data structure shows 11 fold speedup (averaged over various models and octree heights 8 or more). While a point model of a dragon placed inside a point modelled



Figure 4.1: Dragon viewed from the floor (cyan dot). The quality is unacceptable for octrees of heights of 7 (left) or less. The figure on the right is for an octree of height 9.

Cornell room, sub-divided with octree height of 8, takes more than a couple hours, on the CPU, for visibility computation, GPU performs the same in some minutes.

2. The specific kernel of point-pair visibility (namely, is point p visible from point q) proposed in [GKCD07] is analyzed, and an alternate but related formulation is given which is more suitable for GPU implementations.

4.2 The Visibility Map

We assume that the data of interest is available as points; for example, these could be the points belonging to some 3-D point model of say, the Stanford bunny, or might represent centroids of triangular patches of some 3-D mesh. Given a 3D point model, we sub-divide the model space using an adaptive octree (leaves appear at various depths). The *root* represents the complete model space and the sub-divisions are represented by the descendants in the tree; each node represents a volume in model space.

The visibility map (or V-map) for a tree, as defined in [GKCD07], is a collection of visibility links for every node in the tree. The *visibility link* for any node p is a list L of nodes at the same level; every point in any node in L is guaranteed to be visible from every point in p . Fig. 4.2 shows the link structure for some node N . Combining all such link structures defined for every node gives the complete V-map of the tree.

CPU V-MAP CONSTRUCTION: Consider a node-pair AB in Fig. 4.3 for illustration to see if we should set the visibility link between the two. The same procedure is repeated for all node-pairs in the octree so as to define a complete V-map.

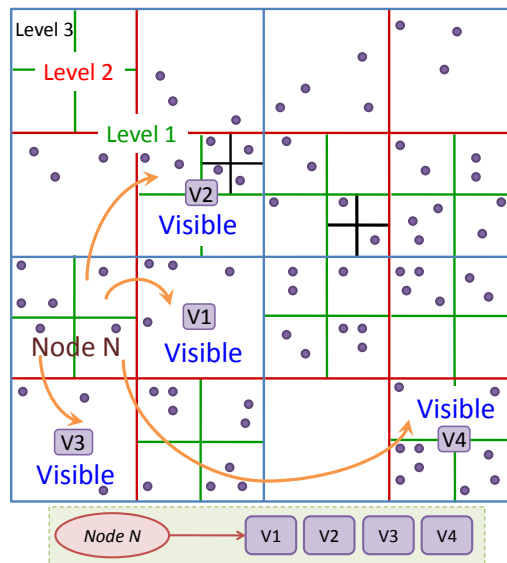


Figure 4.2: Visibility links for Node N

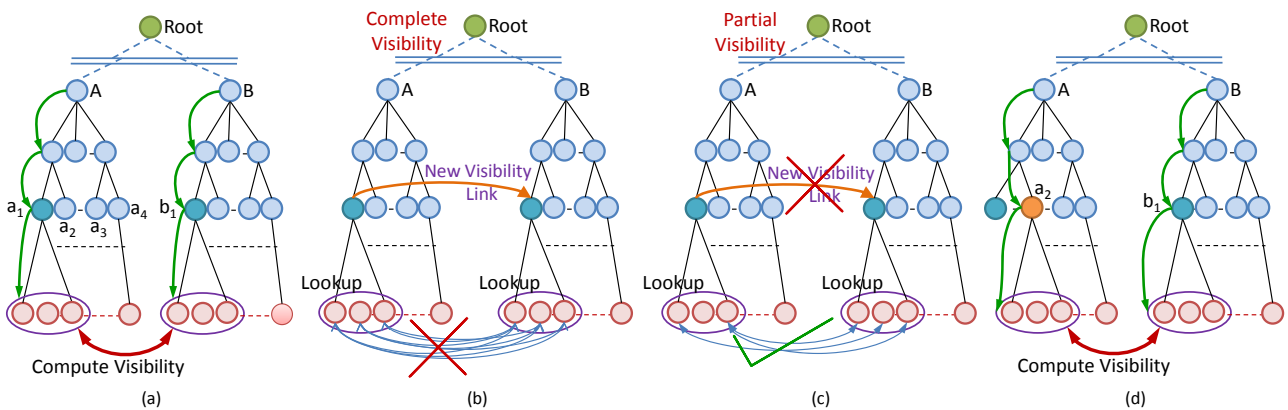


Figure 4.3: The visibility map is constructed recursively by a variation of depth first search. In general, it is advantageous to have links at high level in the tree so that we can reason efficiently using coherency about the visibility of a group of points.

To establish the visibility link between A and B we need to check if all the points in the leaves defined by A are visible to all the points in B . In doing so, we might as well take advantage of this work, and set the visibility links of all descendants of A with respect to all descendants of B . In Fig. 4.3(a) we see (green arrows to the extreme left) how we recursively go down and compute the visibility between a_1 and b_1 with the help of their leaves. This information is now propagated upwards in the sub-tree depending on the type of relation established at the leaves. In our example, Fig. 4.3(b), if all leaves of a_1 are visible to all leaves of b_1 , then the *visibility link* between a_1 and b_1 is set (using *dynamic memory allocation*) with no links between their leaves. However, if only some leaves of a_1 are visible to some leaves of b_1 , then its a case of *partial* visibility and no new link is propagated upwards between a_1 and b_1 (Fig. 4.3(c)). The same process is repeated then for a_2 and

b_1 (Fig. 4.3(d)) and then for rest of the descendants. It is easily observed that recursion (not available on the GPU) is a natural way of efficiently constructing the V-map.

With the setting given above (e.g., links only at the same level) we restrict the pairing of nodes to a subset of the all node pairs set. It is common in the scientific computing literature to define this subset as an *interaction list*. Every node has its own interaction list and its cardinality may vary from node to node.

As mentioned earlier, a sequential implementation of the V-map construction given in [GKCD07] takes hours (for octrees with height 8 or more). Employing octrees of greater height, it is therefore desirable to exploit the inherent parallelism in the V-map construction algorithm and get both quick and accurate results. Parallelism stems from the fact that the visibility between a node pair, say a_1 and b_1 in Fig. 4.3 is entirely independent of the visibility between another node pair, say a_2 and b_1 .

4.3 V-map Computations on GPU

Various ways to parallelize the V-Map computations on the GPU are addressed below.

4.3.1 Multiple Threads Per Node Strategy

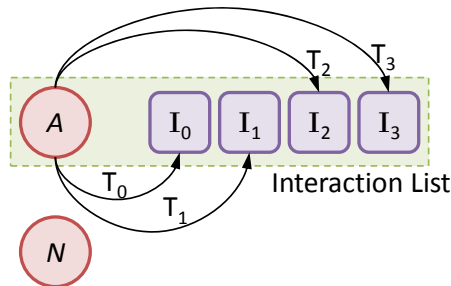


Figure 4.4: Parallelism at a node level

One of the intuitive ways to parallelize the algorithm is to make each thread compute the visibility between any node A with a node in its interaction list. For example, as shown in Fig. 4.4, thread T_0 computes visibility between A and I_0 (e.g., I_0 can be the node B of Fig. 4.3), thread T_1 computes visibility between A and I_1 and so on. Once visibility between A and all nodes in its interaction list is computed, we move to another node N and repeat the same. Thus we allocate *multiple threads per node but only one per node pair*.

The number of threads running concurrently is the size of the interaction list. The degree of parallelism here is limited by the size of the interaction list of a node which might be quite small (generally in tens or hundreds). To unleash the power of GPU we need thousands of threads running concurrently, which is not the case here. However, the threads per node strategy can be combined with other strategies (see below, for example, sec. 4.3.2).

A more serious limitation is that each thread has to perform *recursion* as well as *dynamic memory allocation* for setting up links at the descendant levels. This is *not* possible on the current GPUs. Recursion can be implemented with a user stack; but the dynamic allocation problem persists.

4.3.2 One Thread per Node Strategy

Another intuitive way to compute visibility is to let each thread compute visibility between a node and all the nodes in its interaction list, going down the octree level by level, starting from the root. For example, as shown in Fig. 4.5 thread T_0 computes visibility between a node A and all the nodes in its list, thread T_1 computes visibility between a node N and all the nodes in its list and so on. Thus we allocate *only one thread to compute visibility between a node and its entire interaction list*.

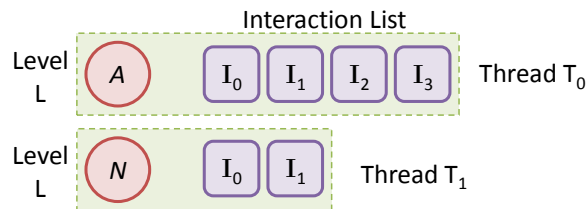


Figure 4.5: Parallelism across nodes at the same level

Note that all nodes at a particular level are considered concurrently before moving on to the next level. Thus the degree of parallelism is equal to the number of nodes at a particular level considered and changes with every level. The performance of the algorithm increases as we go down the octree, as the number of nodes per level tends to increase with greater octree depths (*root* being at depth 0).

One of the drawbacks of this parallel algorithm is the fact that it does not utilize the commutative nature of visibility. That is, if node N_1 is visible to node N_2 , then N_2 is also visible to N_1 . Although such cases can be detected, and threads made to stop execution, almost half of the threads would be wasted. Further, the same limitations (sec. 4.3.1) of dynamic memory allocation and recursion apply to this parallel algorithm, thereby making this case undesirable.

4.3.3 Multiple Threads per Node-Pair

Here we consider a node A and say node B belongs to its interaction list. We compute the visibility between all leaves of A with all leaves of B , *in parallel on the GPU* (and afterwards repeat the same for other node-pairs in the tree). The recursive part is computed on the CPU which uses traditional dynamic memory management.

To achieve the same, we introduce a minor modification to the original CPU-based algorithm (compare Fig. 4.3(a) with Fig. 4.6(a)). In the CPU-based approach, we first recurse in the sub-trees of A and B and then having reached the leaf level, compute the leaf-leaf visibility. In contrast, in our GPU implementation, we first

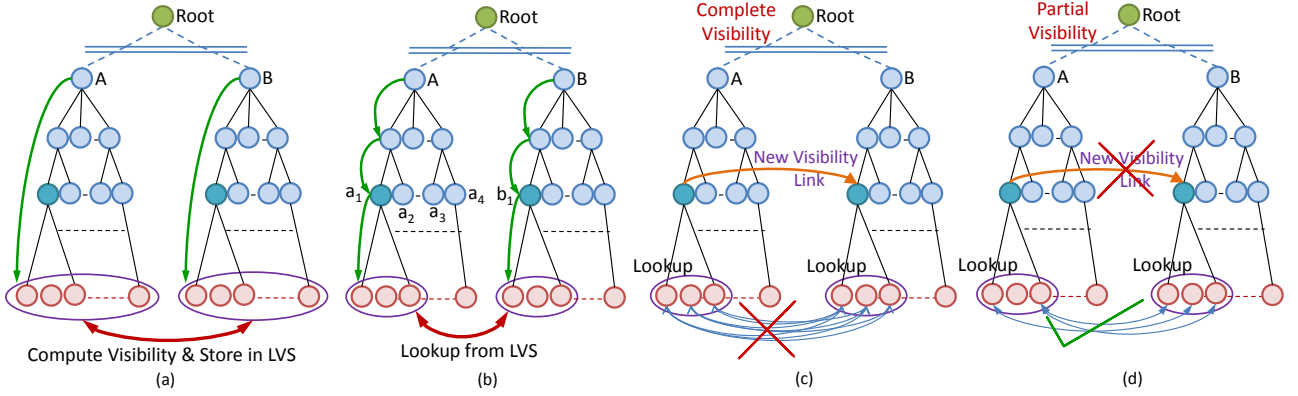


Figure 4.6: The visibility map on the GPU uses thousands of threads concurrently by working at the large number of leaves (a) and stores the result in a table. The links at other levels are set based on a lookup computation.

compute all leaf-pair visibility between two nodes and store it in a Boolean array *LVS* (Leaf Visibility Status). The CPU does the standard recursion, and having reached the leaf level, use a simple look-up to *LVS* to find the already computed answer.

For example, in Fig. 4.6, we first compute visibility of all leaves of *A* with respect to all leaves of *B* and store it in the *LVS* (Fig. 4.6(a)). We then recurse their sub-trees, as in the CPU implementation, and look up the visibility value from the *LVS* (Fig. 4.6(b)) to find whether the descendants (say a_1 and b_1) are completely visible (Fig. 4.6(c)) or partially visible (Fig. 4.6(d)). We repeat the same for all other descendant pairs (say (a_2, b_1)). The algorithm is summarized below.

1. Traverse the octree, starting from the root, on the CPU as follows
2. For every node considered, say *A*, compute visibility between *A* and every node in its interaction list as follows
 - (a) For every node *B* in the interaction list of *A*, allocate nm threads on the GPU. n and m are the number of leaves in *A* and *B* respectively.
 - (b) *In Parallel*, compute the leaf-leaf visibility of n leaves of *A* with m leaves of *B*. Store the result in respective locations in a boolean array *LVS* of size nm (Fig. 4.6(a)).
 - (c) Recurse through the sub-trees of *A* and *B*. Having reached the leaves, look-up the visibility value from respective location in *LVS* and establish the visibility links (Fig. 4.6(b)).
 - (d) Propagate the links upwards as shown in Fig. 4.6 (Fig. 4.6(c),(d)).

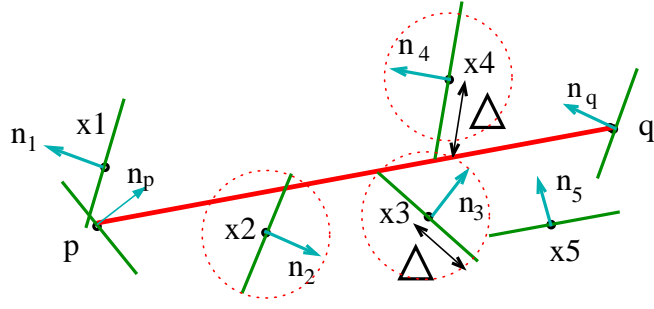


Figure 4.7: Visibility between points p and q

4.4 Leaf-Pair Visibility

Having presented the strategy for constructing, *in parallel*, a global V-map for a given tree, we now discuss the leaf-leaf visibility algorithm performed by each thread. We build on the atomic point-pair visibility algorithm (Fig. 4.7) given in [GKCD07] and extend it for computing visibility between leaves.

4.4.1 Prior Algorithm

As presented in [GKCD07], to compute visibility between any two points p and q , we check if they face each other. If yes, we then determine a set of potential occluders using a 3D Bresenham's line algorithm [E.62]. Bresenham's algorithm outputs a set Y of points which are collinear with and between p and q . Going down the octree recursively, all points from the leaves containing any point from Y is added to a set X . The Bresenham step-length is based on the sampling resolution of the original point dataset.

The potential occluders are pruned further based on the tangent plane intersection tests. In Fig. 4.7, the set X consists of *potential* occluders points $(x_1, x_2, x_3, x_4, x_5)$. In fact, only points x_2 and x_3 are considered as *actual* occluders. Point x_1 is rejected as the intersection point of the tangent plane lies outside segment \overline{pq} , point x_4 because it is more than a distance Δ away from \overline{pq} , and point x_5 as its tangent plane is parallel to \overline{pq} . If K (a parameter) of actual occluders are found, q is considered invisible to p .

Instead of point pairs p and q , one may also use leaves, and apply the same idea. p and q (of Fig. 4.7) are now the leaf's centroids (not centers) and the potential occluders are the centroids of the leaves intersecting line segment \overline{pq} . Δ is the distance from the centroid of an intersecting leaf to its farthest point and is different for every leaf [GKCD07].

Applying this idea for the GPU model, we see that in this way of computing leaf-leaf visibility, we require each thread to recursively go down the octree for finding the potential occluders between the leaf pair considered. As octree height increases, the size of leaves becomes small, thereby reducing the step length of the 3D Bresenham's Line algorithm by a significant amount. This drastically increased the load and the computations as it requires more recursive traversals of the octree. In our GPU implementation, the Bresenham approach

LVS which will be eventually looked up.

4.5 GPU Optimizations

To improve the GPU kernel's performance, we utilize several optimization techniques enlisted below.

1. **ASYNCHRONOUS COMPUTATIONS:** Asynchronous kernel launches were made by overlapping CPU computations with kernel execution. Thus, while the CPU is busy recursing the sub-trees of nodes to set visibility links at different levels of octree, our GPU kernel is busy performing leaf-pair visibility computations for the next node pair whose sub-trees will eventually be visited by CPU when the kernel finishes its part.
2. **LOOP UNROLLING:** Any flow control instruction (if, switch, do, for, while) can significantly impact the effective throughput by causing threads to diverge. Thus, major performance improvements can be achieved by unrolling the control flow loop. We found that especially the loops with global memory accesses (as is the case in our algorithm) can benefit from unrolling.
3. **OPTIMAL THREAD AND BLOCK SIZE:** Obtained via an empirical study, each thread block must contain 128 – 256 threads and every thread block grid no less than 64 blocks for optimal performance on G80 GPU.
4. **OPTIMAL OCTREE HEIGHTS:** As every thread works on single leaf-pair, multiple threads are independent. Each leaf pair may have a different number of *potential occluders* to be considered. The thread that finishes work for a given leaf-pair simply takes care of another leaf-pair, without the need for any shared memory or synchronization with other threads. To effectively use the GPU, the number of leaf-pair should be sufficiently large. With 16 multi-processors, we need at least 64 thread-blocks, each having 256 threads to utilize the GPU. Thus the number of leaf pairs considered concurrently should be at least 16384 for good performance.

4.6 Results

The CUDA based parallel V-map construction algorithm, implemented on G80 NVIDIA GPU, was tested on several point models. In this section we provide qualitative visibility validation and quantitative results. Note that all input such as the models in the room, the light source, and the walls of the Cornell room are given as points.

4.6.1 Visibility Validation

We validate our proposed method here using an *adaptive octree structure*. We remark that the user divides the octree adaptively depending on the input scene density. Increasing or decreasing the levels of subdivision for a given scene is essentially a trade-off between quality of the visibility (user driven), and the computational time.

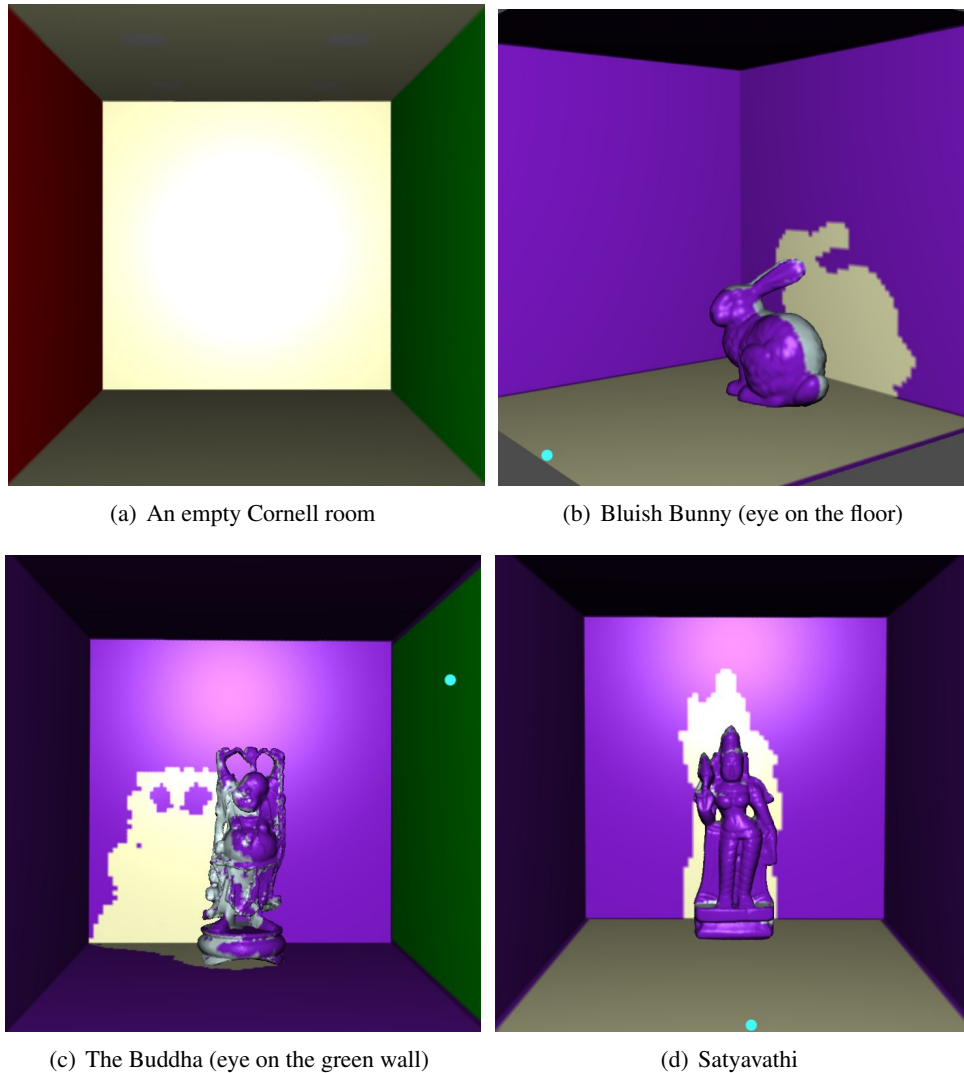


Figure 4.9: Visibility tests where purple color indicates portions visible to the candidate eye (in cyan)

Fig. 4.9(a) shows a point model of an empty Cornell room with some artificial lighting to make the model visible. Note the default colors of the walls. We now introduce a bluish white Stanford bunny. In Fig. 4.9(b), the eye (w.r.t. which visibility is being computed) is on the floor, marked with a cyan colored dot. The violet (purple) color indicates those portions of the room that are visible to this eye. Notice the “shadow” of the bunny on the back wall. The same idea is repeated with the eye (marked in cyan) placed at different locations for various different point models (all bluish white in color) of the Buddha (Fig. 4.9(c)), and an Indian Goddess

Satyavathi (Fig. 4.9(d)). We found that an octree of height 8 gave us accurate visibility results, however, it is more prudent to go to higher depths such as 9 or 10.

4.6.2 Quantitative Results

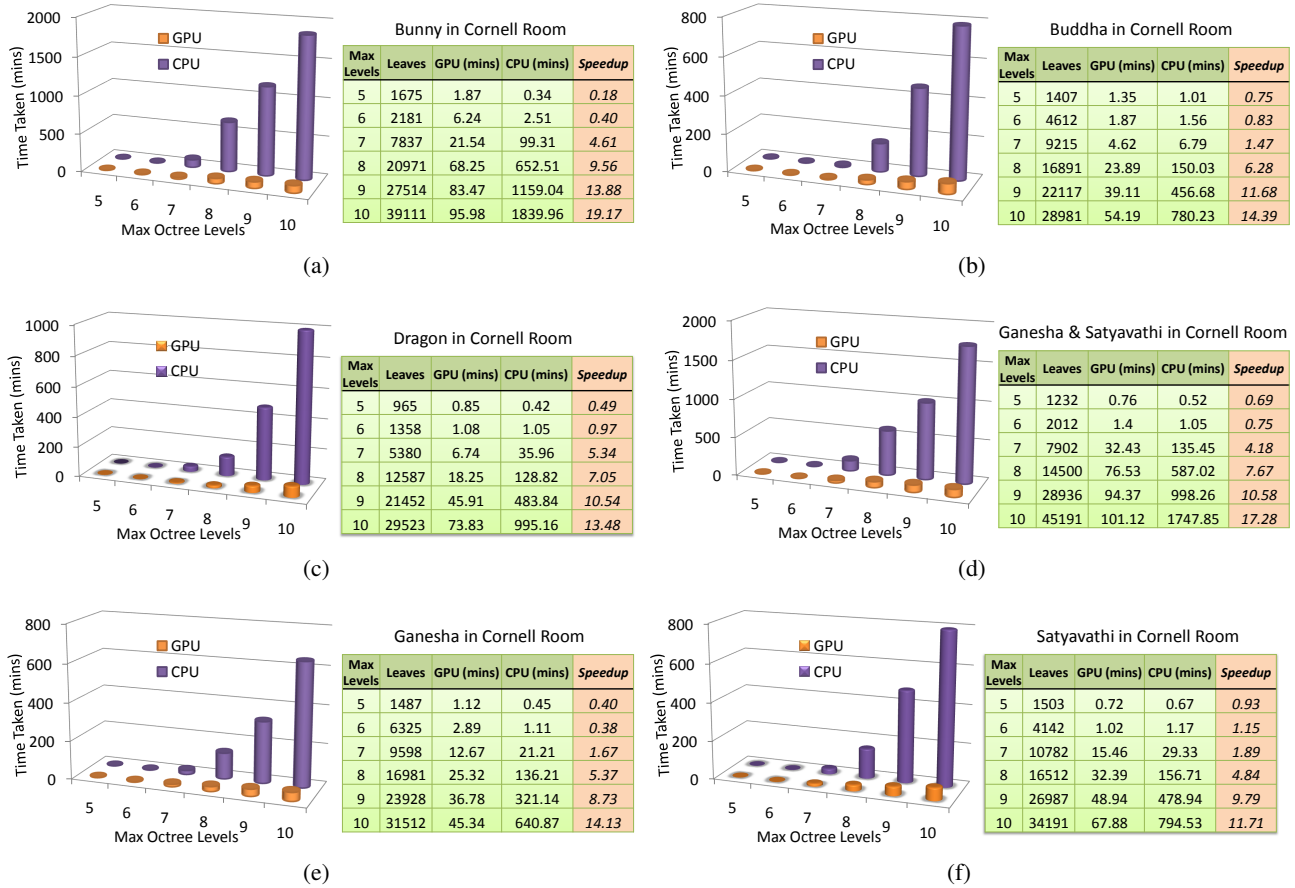


Figure 4.10: V-map construction times (CPU & GPU) for models with differing octree heights

Fig. 4.10 shows the running time of our implementation. Each graph in Fig. 4.10 refers to a particular model, and shows the only-CPU, and CPU-GPU combo running time for various octree levels (5-10). For example., Fig. 4.10(a) shows results for the Stanford bunny in the Cornell room while Fig. 4.10(b) shows the same for Buddha. The table also shows the number of leaves (at various depths) in the various adaptive octrees. This is an important parameter on which the degree of parallelism indirectly depends. The running times tabulated, also depends on the number of threads per block. A block size of (16×16) gives the best results with 256 threads per block.

The CPU and GPU have almost identical run times if the model has octree height of 6 or below. For best throughput from the GPU we need at least 16384 leaf-pairs to be considered concurrently which is generally not the case for octrees built till level 6. The GPU starts out performing the CPU for octrees with greater heights

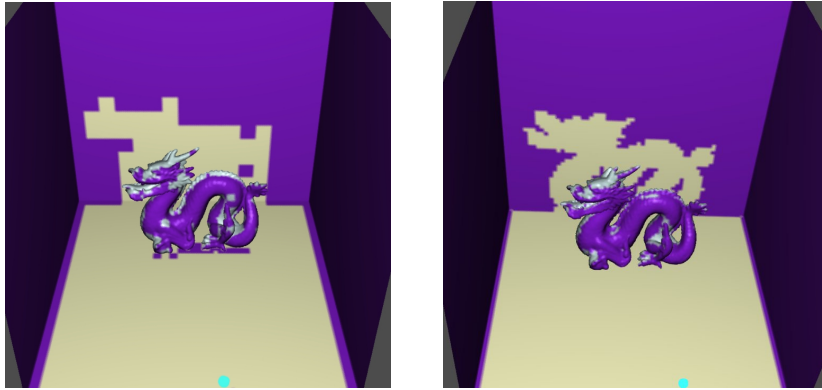


Figure 4.11: Dragon viewed from the floor (cyan dot). The quality is unacceptable for octrees of heights of 7 (left) or less. The figure on the right is for an octree of height 9.

(Fig. 4.10). However, the quality of the visibility solution is below par for octrees with heights 7 or below (Fig. 4.11). Thus, to get a good acceptable accuracy in results (Fig. 4.9) and throughput from GPU, we use octree heights of at least 8. Speedup is also given in the tables. *We achieve an average speed-up (across all models) of 15 when the input models are divided with octree of height 10.* Thus, we see that the CUDA implementation of V-map construction algorithm is efficient and fast. Once constructed, they allow for an *interactive walkthrough* of the point model scene.



Figure 4.12: Point models rendered with diffuse global illumination effects of color bleeding and soft shadows. Pair-wise visibility information is essential in such cases. Note that the Cornell room as well as the models in it are input as point models.

As a proof of applicability, we now use the parallel constructed V-map in a global illumination algorithm, where the Fast Multipole Method is used to solve the radiosity kernel. Fig. 4.12 shows results with the color bleeding effects and the soft shadows clearly visible. The V-map also works well even in the case of aggregated input models (e.g., point models of both Ganesha and Satyavathi placed in a point model of a Cornell room). Note that the input is a single, large, *mixed* point data set consisting of Ganesha, Satyavathi, and the Cornell room. These models were not taken as separate entities nor were they segmented into different objects during the whole process.

Conclusion and Future Work

Point-sampled geometry has gained significant interest due to their simplicity. The lack of connectivity touted as a plus, however, creates difficulties in many operations like generating global illumination effects. This becomes especially true when we have a complex scene consisting of several models, the data for which is available as hard to segment aggregated point-based models. Inter-reflections in such complex scenes requires knowledge of visibility between point pairs. Computing visibility for point models becomes all the more difficult, than for polygonal models, since we do not have any surface or object information.

Point-to-Point Visibility is arguably one of the most difficult problems in rendering since the interaction between two primitives depends on the rest of the scene. One way to reduce the difficulty is to consider clustering of regions such that their mutual visibility is resolved at a group level. Most scenes admit clustering, and the *Visibility Map* data structure we propose enables efficient answer to common rendering queries. We extended, in this report, the novel, provably efficient, hierarchical, visibility determination scheme for point based models to the highly parallel structures of modern day GPUs. By viewing this visibility map as a ‘preprocessing’ step, photo-realistic global illumination rendering of complex point-based models have been shown. By extending the V-map construction algorithm on the GPU, efficient speed-ups have also been reported.

Further, we have used the *Fast Multipole Method (FMM)* as the light transport kernel for inter-reflections, in point models, to compute a description – *illumination maps* – of the diffuse illumination. Parallel implementation of FMM is a difficult task with data decomposition and communication efficiency being the major challenges. We discussed one such algorithm which uses only a static data decomposition on octrees and offers communication efficiency. We exploited the parallel computing power of GPUs for implementation of the *Fast Multipole Method* based radiosity kernel as well as the point-pair visibility determination algorithm using *Visibility Maps* to provide an efficient, *fast* inter-visibility and *diffuse* global illumination solution for point models. Further, a complete global illumination solution for point models should cover *not just* diffuse but specular (reflections, refractions, and caustics) effects as well. Achieving these specular effects is a part of the work to be done in the current year, thereby gaining a complete global illumination package for point models.

References

- [AA03] Anders Adamson and Marc Alexa. Ray tracing point set surfaces. In *SMI '03: Proceedings of the Shape Modeling International 2003*, page 272, Washington, DC, USA, 2003. IEEE Computer Society. 6
- [AGCA08] Prekshu Ajmera, Rhushabh Goradia, Sharat Chandran, and Srinivas Aluru. Fast, parallel, gpu-based construction of space filling curves and octrees. In *SI3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 1–1, New York, NY, USA, 2008. ACM. 27, 29
- [Ama84] John Amanatides. Ray tracing with cones. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 129–135, 1984. 6
- [BCL⁺92] J. A. Board, J. W. Causey, J. F. Leathrum, A. Windemuth, and K. Schulten. Accelerated molecular dynamics simulation with the parallel fast multipole method. *Chemistry Physics Letters*, 198:89–94, 1992. 7
- [CBC⁺01] J. C. Carr, R. K. Beatson, J. B. Cherrie, T. J. Mitchell, W. R. Fright, B. C. McCallum, and T. R. Evans. Reconstruction and representation of 3d objects with radial basis functions. *Proceedings of ACM SIGGRAPH*, pages 67–76, August 2001. 7
- [CUDa] CUDA. NVIDIA CUDA Programming Guide. <http://developer.nvidia.com/cuda>. 7, 25
- [CUDb] CUDPP. CUDA Data Parallel Primitives Library. <http://www.gpgpu.org/developer/cudpp>. 28

- [DS00] J. Dongarra and F. Sullivan. The top ten algorithms. *Computing in Science and Engineering*, 2:22–23, 2000. 6, 7
- [DTG00] Philip Dutre, Parag Tole, and Donald P. Greenberg. Approximate visibility for illumination computation using point clouds. Technical report, Cornell University, 2000. 31
- [DYN04] Yoshinori Dobashi, Tsuyoshi Yamamoto, and Tomoyuki Nishita. Radiosity for point sampled geometry. In *Pacific Graphics*, 2004. 4, 31
- [E.62] Bresenham J. E. Bresenham’s line drawing algorithm, 1962. 37
- [EDD03] A. Elgammal, R. Duraiswami, and L. Davis. Efficient kernel density estimation using the fast gauss transform with applications to color modeling and tracking. *IEEE Transactions on PAMI*, 2003. 7
- [GAC08] R. Goradia, P. Ajmera, and S. Chandran. Gpu-based hierarchical computation for view independent visibility. *Accepted at ICVGIP, Indian Conference on Vision, Graphics and Image Processing*, 2008. 10
- [GD98] J. P. Grossman and William J. Dally. Point sample rendering. In *Rendering Techniques*, pages 181–192, 1998. 31
- [GD07] Nail A. Gumerov and Ramani Duraiswami. Fast multipole methods on graphics processors. *Astro GPU*, 2007. 14
- [GKCD07] R. Goradia, A. Kanakanti, S. Chandran, and A. Datta. Visibility map for global illumination in point clouds. *Proceedings of ACM SIGGRAPH GRAPHITE, 5th International Conference on Computer Graphics and Interactive Techniques*, 2007. 4, 6, 10, 31, 32, 34, 37
- [GKM96] L. Greengard, M. C. Kropinski, and A. Mayo. Integral equation methods for stokes flow and isotropic elasticity. *Journal of Computational Physics*, 125:403–414, 1996. 7
- [Gor06] Rhushabh Goradia. Fmm-based illumination maps for point models. *Second Progress Report, Ph.D.*, 2006. 13
- [Gor07] Rhushabh Goradia. Global illumination for point models. *Third Progress Report, Ph.D.*, 2007. 10, 31
- [GR87] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73:325–348, 1987. 6, 12

- [GSCH93] Steven J. Gortler, Peter Schröder, Michael F. Cohen, and Pat Hanrahan. Wavelet radiosity. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 221–230, New York, NY, USA, 1993. ACM Press. 6
- [GST05] Dominik Göddeke, Robert Strzodka, and Stefan Turek. Accelerating double precision FEM simulations with GPUs. In *Proceedings of ASIM 2005 - 18th Symposium on Simulation Technique*, Sep 2005. 13
- [Har] M. Harris. Parallel Prefix Sum (Scan) with CUDA. <http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.htm>. 28
- [HSO07] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (scan) with cuda. In Hubert Nguyen, editor, *GPU Gems 3*. Addison Wesley, August 2007. 28
- [Jen96] H. W. Jensen. Global illumination using photon maps. *Eurographics Rendering Workshop 1996*, pages 21–30, June 1996. 6
- [KC03] A. Karapurkar and S. Chandran. Fmm-based global illumination for polygonal models. Master's thesis, Indian Institute of Technology, Bombay, 2003. 3, 13, 15
- [KGC04] A. Karapurkar, N. Goel, and S. Chandran. Fmm-based global illumination for polygonal models. *Indian Conference on Computer Vision, Graphics, and Image Processing*, pages 119–125, 2004. 13
- [KTB07] Sagi Katz, Ayellet Tal, and Ronen Basri. Direct visibility of point sets. In *SIGGRAPH '07*, page 24. ACM, 2007. 31
- [LPC⁺00] Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, Jonathan Shade, and Duane Fulk. The digital michelangelo project: 3D scanning of large statues. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 131–144. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000. 2
- [MGPG04] N. J. Mitra, N. Gelfand, H. Pottmann, and L. Guibas. Registration of point cloud data from a geometric optimization perspective. In *Symposium on Geometry Processing*, pages 23–31, 2004. 31
- [NHP07] L. Nyland, M. Harris, and J. Prins. *GPU Gems 3*, chapter Fast N-Body Simulation with CUDA, pages 677–696. Addison Wesley, 2007. 13

- [OBA⁺03] Yutaka Ohtake, Alexander Belyaev, Marc Alexa, Greg Turk, and Hans-Peter Seidel. Multi-level partition of unity implicits. *ACM Trans. Graph.*, 22(3):463–470, 2003. 2, 6
- [OLG⁺07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007. 8
- [PKKG03] Mark Pauly, Richard Keiser, Leif P. Kobbelt, and Markus Gross. Shape modeling with point-sampled geometry. *ACM Trans. Graph.*, 22(3):641–650, 2003. 2
- [PZvBG00] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 335–342. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000. 2
- [RL00] Szymon Rusinkiewicz and Marc Levoy. QSplat: A multiresolution point rendering system for large meshes. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 343–352. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000. 2, 6
- [SJ00] G. Schaufler and H. Jensen. Ray tracing point sampled geometry. In *Eurographics Rendering Workshop Proceedings*, pages 319–328, 2000. 6
- [Wal05] Ingo Wald. High-Quality Global Illumination Walkthroughs using Discretized Incident Radiance Maps. *Technical Report, SCI Institute, University of Utah, No UUSCI-2005-010 (submitted for publication)*, 2005. 6
- [WS03] Michael Wand and Wolfgang Straer. Multi-resolution point-sample raytracing. *Graphics Interface*, pages 139–148, 2003. 2
- [WS05] Ingo Wald and Hans-Peter Seidel. Interactive Ray Tracing of Point Based Models. In *Proceedings of 2005 Symposium on Point Based Graphics*, 2005. 4, 6