# GPU-based Hierarchical Computations for View Independent Visibility


## Abstract

*With rapid improvements in the performance and programmability, Graphics Processing Units (GPUs) have fostered considerable interest in substantially reducing the running time of compute intensive problems. The solution to the view-independent mutual point-pair visibility problem (required for inter-reflections in global illumination) can, it would seem, require the capabilities of the GPUs.*

*In this paper, various ways of parallelizing the construction of the Visibility Map (V-map, a description of mutual visibility) are presented to lead the way for an implementation that achieves a speedup of 11 or more. We evaluate our scheme qualitatively and quantitatively, and conclude that parallelizing the V-map construction algorithm is eminently useful.*


## 1. Introduction

Abstractly speaking, the problem we consider in this paper assumes that we have data in the form of points, possibly in high dimensions. The data is large — approximately few lakhs of points — and we therefore are forced with a **hierarchy**. We wish to compute some **relationships** on this data at various levels in the hierarchy. More concretely, consider the motivating problems sketched below.

- SCIENTIFIC COMPUTING [1] The n-body problem is the problem of finding, given the initial positions, masses, and velocities of $n$ bodies, their subsequent motions as determined by classical mechanics. Direct simulation is often impossible; in methods such as the Barnes-Hut simulation, the volume is usually divided up into cubic cells in an octree, so that only particles from nearby cells need to be treated individually for the gravitational relationship, and particles in distant cells can be treated as a single large particle centered at its center of mass (or as a low-order multipole expansion [2]). This can dramatically reduce the number of particle pair interactions that must be computed.
- IMAGE UNDERSTANDING [14] In the city photo tourism application, a large collection of images (say, of one or more monuments, scenic points, buildings, major roads, etc.) are obtained and an attempt is made to compute the 3D structure of the various scenes to enable the navigation of the virtual tourist. The preprocessing solution strategy (grossly abbreviated here) is to collect SIFT [11] features of the images, discover relationships ("what parts, if any, of these two image are similar?"), and construct a hierarchy of the scenes.
- WEB INFORMATION RETRIEVAL [3] Blogs are to be classified as belonging to various semantic categories such as entertainment, sports, politics, and so on. However, relationships exist between all pairs of blogs, some strong, and others weak that prevent a strict partitioning of web pages in the hierarchy. A preprocessing step on the hierarchy enables focused retrieval if nodes are allowed to be linked with other nodes.
- VIEW INDEPENDENT VISIBILITY [7] In point-based graphics [6, 10, 8, 12], a scene represented as points is to be rendered from various viewpoints keeping global illumination in mind [5]. That is, a point may be illuminated by other points, which in turn are illuminated by still other points, and so on. 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 §3) 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.

A sequential implementation of the V-map given in [7] 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. 11). GPUs have evolved into an attractive [13] hardware platform for general purpose computations due to their high floating-point processing performance, huge memory bandwidth and the comparatively low cost.

**CONTRIBUTIONS:** This paper 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, matching of SIFT features, mutual relationships in blogs, point-pair visibility), we show how a hierarchical data structure can be built efficiently (see §4) 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 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 (Fig. 8).

2. The specific kernel of point-pair visibility (namely, is point $p$ visible from point $q$) proposed in [7] is analyzed, and an alternate but related formulation is given which is more suitable for GPU implementations.

The rest of the paper is organized as follows. For completeness, an overview of the NVIDIA's G80 GPU and CUDA is given in §2. The nature of tree computation for the CPU-based V-map construction algorithm is presented in §3. This tree computation algorithm is extended to the parallel architecture of the GPU in §4. Three different ways of parallelizing are presented to pave for the one chosen. Details of an "atomic" point-pair visibility query appear in §5 which is later incorporated in the parallel V-map implementation. Quantitative and qualitative results along with some GPU based optimization used to improve the algorithm's run-time efficiency are explained in §6. We follow this up with concluding remarks and future work in §7.

## 2. GPU Programming Model

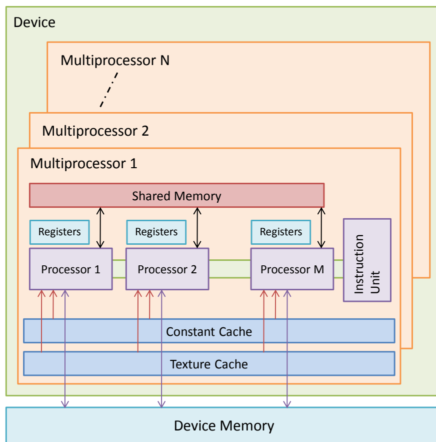



**Figure 1. Hardware Model of GPU**

NVIDIA's G80/G92 architecture GPUs are typical of current generation graphics hardware which uses a large number of parallel threads [4] to hide memory latency. Programs are written in C/C++, with CUDA specific extensions. A program consists of a host component executed on the CPU, and a GPU component. The host component issues bundles of work (GPU kernels) to be performed by threads executing on the GPU. Threads are organized as a grid of thread blocks and are run in parallel. A typical computation running on the GPU must express hundreds of threads in order to effectively use the hardware capabilities.

The G80 (Fig. 1) has $N = 16$ multiprocessors operating on a bundle of threads in SIMD fashion. All multiprocessors can talk to a large (320MB) global device memory (shown in blue). In addition, a set of 8192 registers per multiprocessor, and a total constant memory of 64kB are available. The $M = 8$ processors within each multiprocessor share 16kB of fast read-write "shared" memory (shown in red). This memory is (ironically) not shared with other (processors) in other multiprocessors. The memory access times vary considerably for these different types of memory.

From the programmers perspective, the code executing on the GPU has a number of constraints that are not imposed on host code; the major ones being *no support for dynamic memory allocation and recursion* in the kernel code.

In summary, we need to design our parallel algorithm to have large number of threads, use shared memory wisely, and get around programming constraints.

## 3. 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.

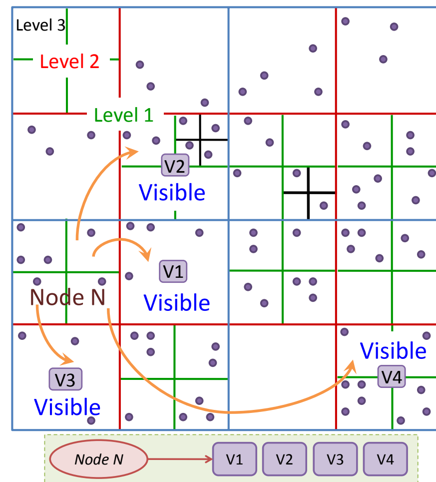



**Figure 2. Visibility links for Node N**

The visibility map (or V-map) for a tree, as defined in [7], 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

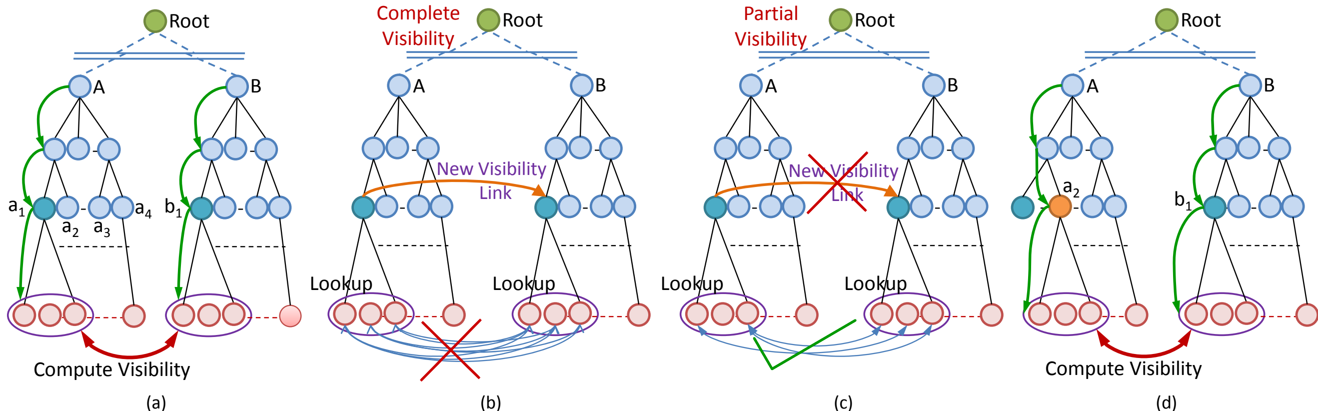



**Figure 3. The visibility map is constructed recursively by a variation of depth first search. In general, it is advantageous to have links at high level in the tree so that we can reason efficiently using coherency about the visibility of a group of points.**

same level; every point in any node in $L$ is guaranteed to be visible from every point in $p$. Fig. 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. 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.

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. 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. 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. 3(c)). The same process is repeated then for $a_2$ and $b_1$ (Fig. 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.
DISCUSSION: While the setting given above (e.g., links only at the same level) is specific to the visibility problem, other applications given in the introduction may 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 [7] 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. 3 is entirely independent of the visibility between another node pair, say $a_2$ and $b_1$.

## 4. V-map Computations on GPU

Various ways to parallelize are addressed below.

### 4.1. Multiple Threads Per Node Strategy

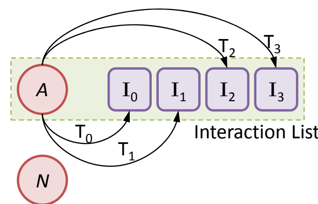



**Figure 4. Parallelism at a node level**

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, thread $T_0$ computes visibility between $A$ and $I_0$ (e.g.,. $I_0$ can be the node $B$ of Fig. 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

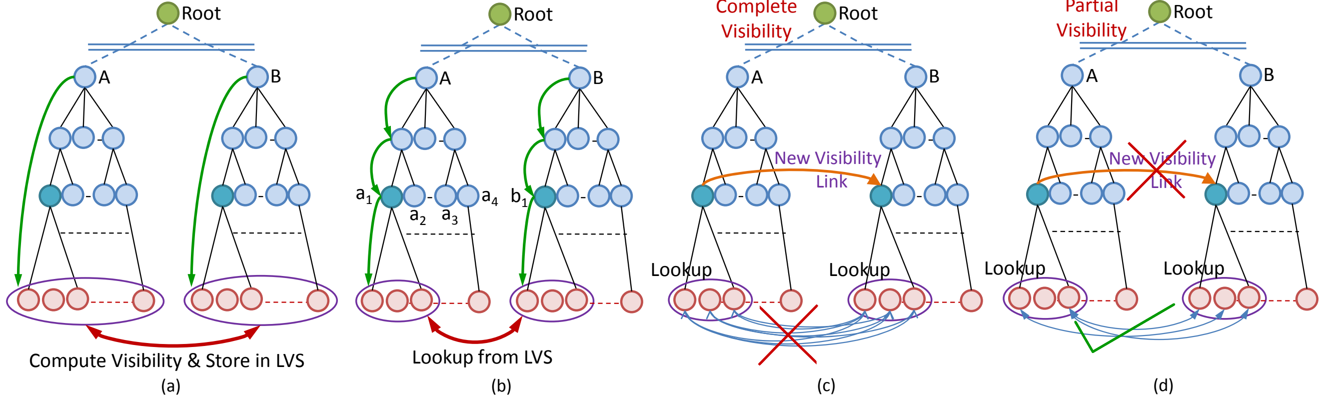



**Figure 5. The visibility map on the GPU uses thousands of threads concurrently by working at the large number of leaves (a) and stores the result in a table. The links at other levels are set based on a lookup computation.**

move to another node $N$ and repeat the same. Thus we allocate *multiple threads per node but only one per node pair.*
DISCUSSION: 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). As commented previously (see §2), 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, §4.2).

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 (Refer §2). Recursion can be implemented with a user stack; but the dynamic allocation problem persists.

## 4.2 One Thread per Node Strategy

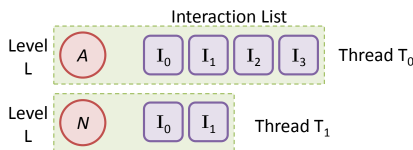



**Figure 6. Parallelism across nodes at the same level**

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 form the root. For example, as shown in Fig. 6 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.*
DISCUSSION: 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 (§4.1) of dynamic memory allocation and recursion apply to this parallel algorithm, thereby making this case undesirable.

## 4.3. 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 afterwords 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. 3(a) with Fig. 5(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 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. 5, we first compute visibility of all leaves of $A$ with respect to all leaves of $B$ and store it in the *LVS* (Fig. 5(a)). We then recurse their sub-trees, as in the CPU implementation, and look up the visibility value from the *LVS* (Fig. 5(b)) to find whether the descendants (say $a_1$ and $b_1$) are completely visible (Fig. 5(c)) or partially visible (Fig. 5(d)). We repeat the same for all other descendant pairs (say $(a_2, b_1)$).

## 5. Leaf-Pair Visibility

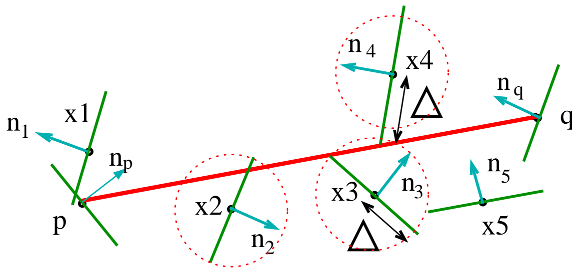



**Figure 7. Visibility between points $p$ and $q$**

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. 7) given in [7] and extend it for computing visibility between leaves.

### 5.1. Prior Algorithm

As presented in [7], 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. 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. 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 $\Delta$ 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. 7) are now the leaf's centroids (not centers) and the potential occluders are the centroids of the leaves intersecting line segment $\overline{pq}$. $\Delta$ is the distance from the centroid of an intersecting leaf to its farthest point and is different for every leaf [7].

DISCUSSION: 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 is therefore abandoned in computing potential occluders; however, the second stage of actual occluders is retained.

### 5.2. Computing Potential Occluders

At the time of construction of the octree, a sphere is initially, and implicitly constructed for every leaf node. The center of each sphere is the center of the respective leaf, and the radius is the distance from the center to the farthest point in the leaf. The spheres for internal nodes are constructed recursively using the maximum radius of their children (Fig. 9(a)). Spheres of siblings therefore might overlap, but this makes our visibility test a bit conservative without hampering the correctness of the results. Note that, only the radius of the sphere need to be stored, since the center is already present in the octree.

Consider the leaf pair $L_1$ and $L_2$. We now recursively compute the intersection status of the line segment $\overline{pq}$ ($p$ be-

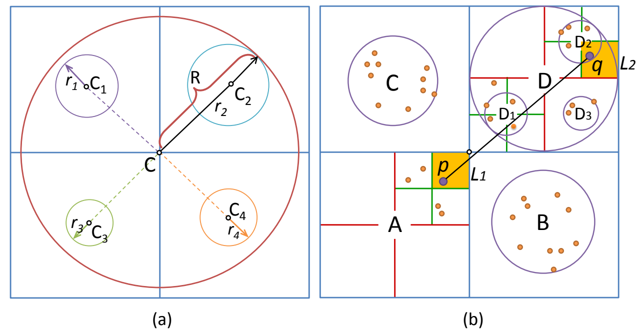
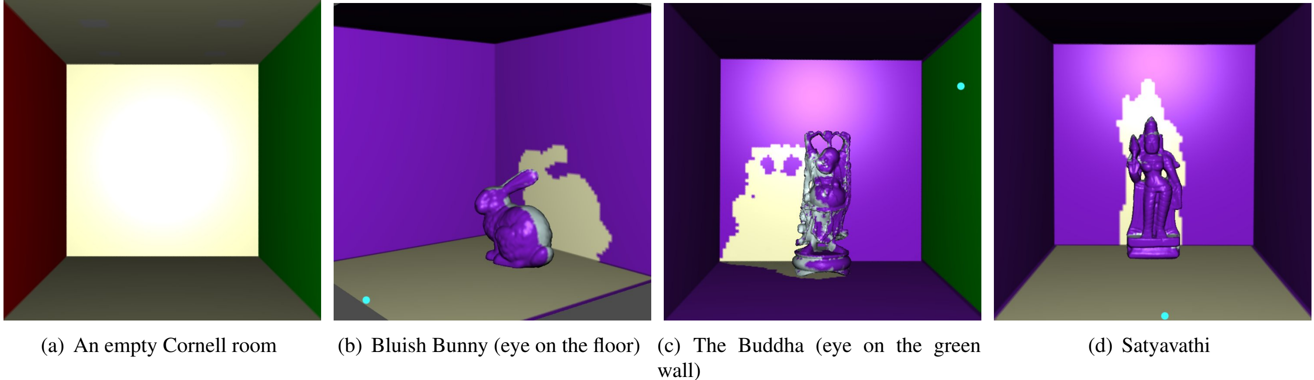



**Figure 9. (a) Constructing parent sphere from child, (b) Line Segment-Sphere Intersection test**



(a) An empty Cornell room (b) Bluish Bunny (eye on the floor) (c) The Buddha (eye on the green wall) (d) Satyavathi

**Figure 8. Visibility tests where purple color indicates portions visible to the candidate eye (in cyan)**

ing centroid of $L_1$ and $q$ of $L_2$) and the spheres of nodes of the octree, starting from the root. It is significant to note that we are *not* interested in computing the actual point of intersection, only the Boolean decision of intersection between $\overline{pq}$ and the sphere under consideration. This is achieved by performing a very simple dot-product computation. If a node contains $L_1$ or $L_2$ or intersects line segment $\overline{pq}$, we recursively perform a similar test in the sub-tree of the intersecting node. Otherwise we discard the node.

For example, we discard the sub-tree of node $C$ (Fig. 9(b)) as it does not intersect $\overline{pq}$ nor does it contain $L_1$ or $L_2$. On the other hand, we traverse the children of node $D$ and recursively repeat the same for sub-trees of children $D_1$ and $D_2$. In this we reaching the (potential occluder) intersecting leaf nodes.

Each thread then performs the same tangent-surface intersection tests (as detailed in §5.1) for their respective leaf-pairs. If mutually visible, each thread adds 1 to the corresponding location in the Boolean array *LVS* which will be eventually looked up.

## 6. Results and GPU Optimizations

The CUDA based parallel V-map construction algorithm, implemented on G80 NVIDIA GPU, was tested on several point models. We provide qualitative visibility validation and quantitative results, along with details on the GPU-based kernel optimization. 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.

### 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.

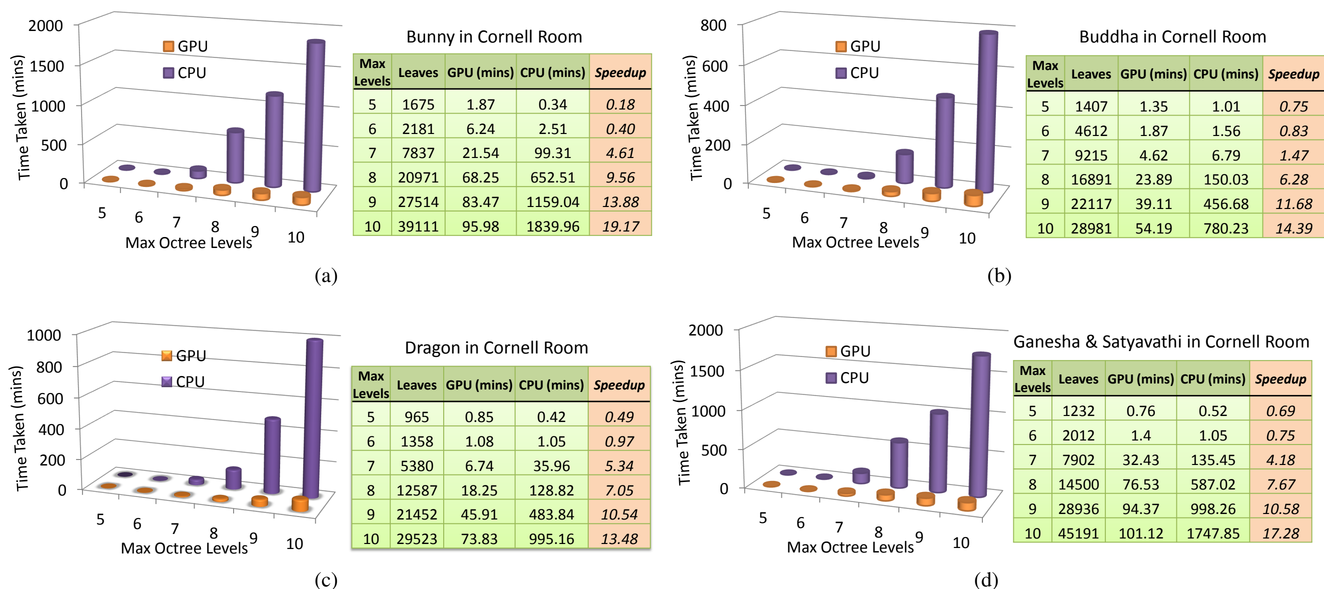
Fig. 8(a) shows a point model of an empty Cornell room with some artificial lighting to make the model visible. Note the default colors of the walls. We now introduce a bluish white Stanford bunny. In Fig. 8(b), the eye (w.r.t. which visibility is being computed) is on the floor, marked with a cyan colored 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. 8(c)), and an Indian Goddess Satyavathi (Fig. 8(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.

### 6.2. Quantitative Results

Before moving on to the timing charts of our algorithm implementation, we would like to present some of the **GPU optimization techniques** we utilized so as to improve our kernel's run-time efficiency.

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 espe-




Bunny in Cornell Room

| Max Levels | Leaves | GPU (mins) | CPU (mins) | Speedup |
|---|---|---|---|---|
| 5 | 1675 | 1.87 | 0.34 | *0.18* |
| 6 | 2181 | 6.24 | 2.51 | *0.40* |
| 7 | 7837 | 21.54 | 99.31 | *4.61* |
| 8 | 20971 | 68.25 | 652.51 | *9.56* |
| 9 | 27514 | 83.47 | 1159.04 | *13.88* |
| 10 | 39111 | 95.98 | 1839.96 | *19.17* |

Buddha in Cornell Room

| Max Levels | Leaves | GPU (mins) | CPU (mins) | Speedup |
|---|---|---|---|---|
| 5 | 1407 | 1.35 | 1.01 | *0.75* |
| 6 | 4612 | 1.87 | 1.56 | *0.83* |
| 7 | 9215 | 4.62 | 6.79 | *1.47* |
| 8 | 16891 | 23.89 | 150.03 | *6.28* |
| 9 | 22117 | 39.11 | 456.68 | *11.68* |
| 10 | 28981 | 54.19 | 780.23 | *14.39* |

Dragon in Cornell Room

| Max Levels | Leaves | GPU (mins) | CPU (mins) | Speedup |
|---|---|---|---|---|
| 5 | 965 | 0.85 | 0.42 | *0.49* |
| 6 | 1358 | 1.08 | 1.05 | *0.97* |
| 7 | 5380 | 6.74 | 35.96 | *5.34* |
| 8 | 12587 | 18.25 | 128.82 | *7.05* |
| 9 | 21452 | 45.91 | 483.84 | *10.54* |
| 10 | 29523 | 73.83 | 995.16 | *13.48* |

Ganesha & Satyavathi in Cornell Room

| Max Levels | Leaves | GPU (mins) | CPU (mins) | Speedup |
|---|---|---|---|---|
| 5 | 1232 | 0.76 | 0.52 | *0.69* |
| 6 | 2012 | 1.4 | 1.05 | *0.75* |
| 7 | 7902 | 32.43 | 135.45 | *4.18* |
| 8 | 14500 | 76.53 | 587.02 | *7.67* |
| 9 | 28936 | 94.37 | 998.26 | *10.58* |
| 10 | 45191 | 101.12 | 1747.85 | *17.28* |

**Figure 10. V-map construction times (CPU & GPU) for models with differing octree heights**

cially 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 contains $128 - 256$ 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.

Fig. 10 shows the running time of our implementation. Each graph in Fig. 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. 10(a) shows results for the Stanford bunny in the Cornell room while Fig. 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 $(16 \times 16)$ 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 (Fig. 10). However, the quality of the visibility solution is below par for octrees with heights $7$ or below (Fig. 11). Thus, to get a good acceptable accuracy in results (Fig. 8) 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.

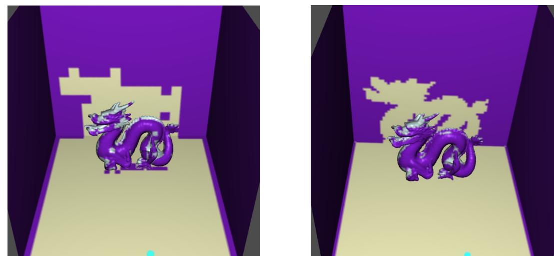


**Figure 11. Dragon viewed from the floor (cyan dot). The quality is unacceptable for octrees of heights of 7 (left) or less. The figure on the right is for an octree of height 9.**

**Figure 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.**

### 6.3. Visibility Map in Global Illumination

As a proof of applicability, we now use the parallel constructed V-map in a global illumination algorithm, where the Fast Multipole Method [2] is used to solve the radiosity kernel. Fig. 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 in Fig. 12). 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.

## 7. Final Remarks

Rapid developments in graphics hardware have made them an attractive platform for solving computationally intensive problems in varied fields. One such problem is performing various parallel computations on trees. *Parallel implementation of the View-Independent Mutual Point-Pair Visibility problem is an example problem whose solution has been implemented on the GPU in this paper.* Various ways to achieve the desired parallelism were presented, and a suitable one chosen. The core kernel computation of leaf-pair visibility was modified to perform line segment-sphere intersections for better speedup.

Blocks containing $256$ threads achieved optimal GPU performance. Visibility results were validated and speedups of up to $19$ (w.r.t CPU) were reported for octrees constructed on models till maximum depth $10$. By viewing this V-map as a 'preprocessing' step, photo-realistic global illumination rendering of complex point-based models have been shown.

The parallel V-map construction algorithm presented can also be used as an intuition to achieve parallelism for other complex problems mentioned in the introduction.

## References


[1] The N-Body Problem. `http://en.wikipedia.org/wiki/N-body_problem`. (Last seen on 30th June, 2008).

[2] J. Carrier, L. Greengard, and V. Rokhlin. A Fast Adaptive Multipole Algorithm for Particle Simulations. *SIAM Journal of Scientific and Statistical Computing*, 9:669–686, July 1988.

[3] S. Chakrabarti. *Mining the Web: Discovering Knowledge From Hypertext Data*. Morgan Kaufmann, 2002.

[4] CUDA. Nvidia Compute Unified Device Architechture Programming Guide. `http://developer.nvidia.com/cuda`. (Last viewed on 30th June, 2008).

[5] Y. Dobashi, T. Yamamoto, and T. Nishita. Radiosity for point-sampled geometry. In *Proc. of Pacific Graphics*, pages 152–159, 2004.

[6] P. Dutre, P. Tole, and D. P. Greenberg. Approximate visibility for illumination computation using point clouds. Technical report, Cornell University, 2000.

[7] R. Goradia, A. Kanakanti, S. Chandran, and A. Datta. Visibility map for global illumination in point clouds. In *ACM SIGGRAPH GRAPHITE '07*, pages 39–46. ACM, 2007.

[8] J. Grossman and W. Dally. Point sample rendering. In *Proc. of Eurographics Workshop on Rendering*, pages 181–192, 1998.

[9] A. Karapurkar, N. Goel, and S. Chandran. The Fast Multipole Method for Global Illumination. *ICVGIP*, pages 119–125, 2004.

[10] S. Katz, A. Tal, and R. Basri. Direct visibility of point sets. In *SIGGRAPH '07*, page 24. ACM, 2007.

[11] D. Lowe. Distinctive Image Features from Scale-invariant Keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[12] 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.

[13] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[14] N. Snavely, S. M. Seitz, and R. Szeliski. Photo tourism: Exploring photo collections in 3d. In *SIGGRAPH Conference Proceedings*, pages 835–846, NY, USA, 2006. ACM.