Real-time Rendering and Dynamic Updating of 3-d Volumetric Data

Andrew Miller
Computer Vision Group, Inc.
acm@computervisiongroup.com

Vishal Jain
Computer Vision Group, Inc.
vj@computervisiongroup.com

Joseph L. Mundy
Computer Vision Group, Inc.
mundy@lems.brown.edu

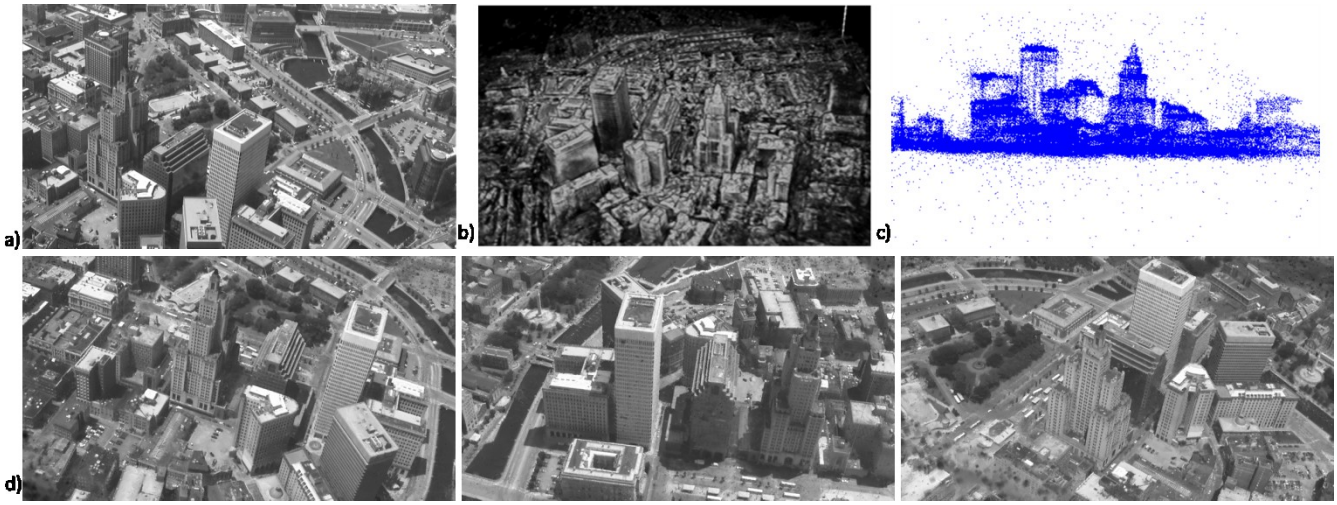


Figure 1 One of the aerial video images (a) of downtown, Providence, RI used to build the 3-d model shown in b). The locations with higher probability of being a surface are indicated as white. Compared the dense reconstruction of (b) to sparse point cloud reconstruction in (c). The rendered images of the 3-d model in b) using the proposed OpenCL implementation on GPU is shown in the bottom row d).

ABSTRACT

A dense 3-d terrain model obtained using reconstruction methods from aerial images is represented in a probabilistic volumetric framework. The choice of probabilistic representation is to represent inherent ambiguity in reconstruction of surface from images. Such probabilistic representation handles the ambiguity very well but leads to expensive dense volumetric storage. The area coverage required for building 3-d models varies from half a square kilometer to thousands of kilometers. Extensive computational resources are required for rendering and building such large models. Existing methods for rendering 3-d models typically cater to mesh models only and also lack strategies to dynamically update the models due to memory intensive operations conventionally better suited for CPUs. This paper proposes a novel OpenCL implementation catering to both GPUs and CPUs for real-time visualizing as well as updating, and dynamic restructuring of dense volumetric models for 3-d terrain. The major contributions of this paper are hybrid representation of grid and octrees, bit-based representation of octrees, randomization of data to enable parallelization of an otherwise serial strategy for subdividing octrees, real-time rendering of dense volumetric data and segmentation algorithm for minimizing global memory access in GPUs. The ideas and implementations proposed could potentially be used in different applications.

GPGPU-4, Mar 05-05 2011, Newport Beach, CA, USA

Keywords

OpenCL, volumetric data rendering, reconstructing volumetric data and dynamic restructuring of octrees.

1. INTRODUCTION

Automatic large-scale dense 3-d terrain reconstruction methods [1; 2] of any scene such as urban, rural, deserts or mountainous areas from images or other sensors require volumetric representation of 3-d data. The volumetric data being denser than meshes require high amount of storage and extensive computation. Due to the inherent ambiguity or uncertainty of the reconstruction methods, meshes of 3-d terrain cannot be obtained. Either unrealistic assumptions such as surface smoothness, *etc.* or manual intervention of skilled photogrammetrist is required to obtain a mesh of the terrain 3-d data. However, the unrealistic assumptions corrupt the model and the manual intervention limits the size and the level of details of the model. Therefore, meshes cannot be reliably obtained for high resolution 3-d terrain data.

Ambiguity in 3-d terrain models can be represented using a probabilistic volumetric framework, [1; 3]. Each discretized cell in the volume of interest contains a probability of being a surface

which varies in the range (0,1) and also contains an appearance distribution. This representation prevents any error in deciding if a cell is a surface or not if the scene evidence is ambiguous. However, such probabilistic representation at each discretized cell in the volume leads to a dense representation of a 3-d model causing an order of magnitude more data in contrast to meshes. Numerous applications such as change detection [3], persistent tracking, and virtual reality require real-time rendering, updating and processing of this volumetric 3-d model. The model size can vary from a fraction of a square kilometer to thousands of square kilometers. An example of a probabilistic model derived from aerial video is shown in Figure 1. The enormity of such models poses a challenging requirement of extensive computational resources for visualization, reconstruction, processing and updating of such models.

Graphics Processing Units (GPUs) are becoming the powerhouse for flops leading to an attractive computational resource for general purpose computing. GPUs provide an enormous flop rate through a huge array of small processors. For example, the NVIDIA GPU GeForce GTX480 has 480 processors that can perform at 1.35 Teraflops for single precision computation compared to 0.1 Teraflops provided by top of the line six-core CPU Intel i7 980x. Any computation which can be subdivided into large number of small parallel local problems is suitable for GPUs. The key challenge in General Purpose GPU (GPGPU) computing is to hide the latency of relatively low memory bandwidth with multiple threads. In GPUs, many resources such as memory, registers, *etc.* are shared among a large number of processors. Therefore, GPU implementations are optimized by reducing the utilization of the shared resources.

A recent implementation of ray-tracing on a GPU for rendering large 3-d scenes is described in [4]. The polygonal scenes are converted into compact and detailed volumetric data in the form of an octree for real-time visualization. The images are rendered by traversing along each ray though the octree. Each of the pixels in the rendered image represents a ray and the ray-tracing for all the pixels in the image is done in parallel. The surfaces are assumed to be opaque resulting in termination of the ray as soon as it hits a surface octree cell. Impressive results are achieved for rendering surfaces with great detail. Crassin *et al.* [5] proposed an approach of storing surface geometries in a multi-resolution scheme. An octree is used for easy traversal and the leaf nodes of the octree contains a small 3-d grid to store the texture information for faster rendering. The combination of an octree and 3-d grid structure is faster but also leads to excess memory use. Nevertheless, a major contribution of above approach is to enable rendering of large models. However, its limitations are: (i) it cannot handle dense volumetric data; (ii) its data structures have to be preprocessed; and (iii) dynamic updating is not allowed or is expensive.

Knoll *et al* [6] presents an approach to visualize volumetric data using iso-surfaces. The volumetric data is compressed into an octree and bundles or packets of coherent rays are cast to render a complex scene. But these methods are not efficient for a SIMD GPU architecture as they rely heavily on cross-ray dependent memory accesses. Specifically terrain rendering approaches use models based on tiled blocks and nested regular grids, quad-trees and triangle bin-trees triangulations, described in [7]. One such terrain rendering approach by Dick *et al* [8] presents a large scale terrain ray-casting based rendering approach where 3-d terrains are represented as Digital Elevation Maps (DEMs) and an orthographic texture. But unlike in the case of dense 3-d volumetric data, the approach only has single intersection along the ray passing through DEMs.

Volumetric 3-d data stored in a regular 3-d grid is very space inefficient and an octree representation can be used instead, as shown in [1]. On the other hand, ray-tracing is easier and computationally inexpensive for regular grid compared to the octree. A hybrid representation of grids and octrees is proposed in this paper to provide real-time implementation as well as provide efficient storage. Each cell of the grid stores an octree, which is opposite to the idea proposed by [5]. Furthermore octrees are represented as bit codes to save more space as opposed to conventional representations of pointer or index based trees [9].The visualization and updating of 3-d terrain model is based on ray-tracing through the volume. A full OpenCL implementation of ray-tracing for such hybrid grid-octree volumetric 3-d data has been developed and will be described below. Visualization (rendering) is read-only, while incremental model construction (update) is both read and write. The authors propose various optimizations for speeding up the rendering as well as updating along the ray.

The update of 3-d terrain models requires restructuring the octrees data by splitting cells into finer resolution or merging cells depending on the existence of a surface or open space in the scene, respectively. Such dynamic updating of the octrees requires adding or removing data stored in highly organized buffers and is inherently serial. A novel parallel solution of breaking this data buffer into multiple 1-d buffers is proposed and discussed in Section 4.4.

All the above proposed ideas have contributed to efficient storage of large probabilistic models on a GPU and CPU and a 600x speed up for visualizing and updating the model compared to a single-thread C++ implementation. The authors provide an open-source OpenCL implementation, available at [10] for the proposed approaches. The major contribution of this paper is a novel GPU-specific OpenCL implementation of : (i) hybrid grid-octree and bit representation of octrees; (ii) ray tracing for volumetric models; (iii) model update; and (iv) dynamic restructuring of octree data.

2. Hybrid Grid-Octree Representation

The 3-d terrain model for the scene shown in Figure 1 is represented by a 1536x1536x512 grid which is approximately 1.2 billion cells. By contrast, the same terrain and information when represented by a 10-level octree uses only about 12 million cells. A major drawback is that ray tracing in the octree representation is computationally expensive.

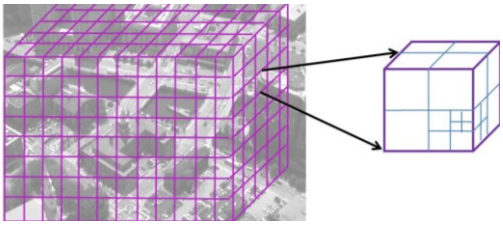


Figure 2 This figure illustrates a coarse level grid for volumetric data where each cell of the grid contains a 4-level octree.

The typical method to find the next cell along the ray during ray tracing in the case of a fixed resolution grid is to convert the exit point of the ray into the grid index and a single access is required to access the next cell along the ray. In the case of an octree, the

next cell is found by finding the neighbor along the exit face of the current cell. The neighbor is estimated by traversing the pointers of the octree. The average number of cell accesses for a 10-level tree is found to be *4*. Thus, numerous accesses to global memory of the GPU are needed before the cell of interest is loaded for ray traversal. In order to reduce expensive global memory accesses, a hybrid representation with a fixed grid defined at a low spatial resolution, and each cell (block) of the grid contains a 4-level octree to reach the full cell resolution, as shown in Figure 2. This approach serves two purposes: data compression; and reduced global memory access. A typical scene with this hybrid scheme requires roughly 2.3 million grid cells and 20 million octree cells. The average number of memory accesses is two.

Global memory access and storage is further reduced by proposing a bit-representation for each of the octrees as compared to pointer and index based schemes. The bit octree can be loaded once and stored in the local memory or registers of the GPU reducing the memory accesses to *one*, the same as the grid.

2.1 Octree Compression

Octrees are typically represented by cell data structures that contain structural information (parent and child pointers) as in [9] as well as index to the scene data, such as an appearance model. For example, a typical octree cell consists of a parent pointer, a child pointer and a data pointer.

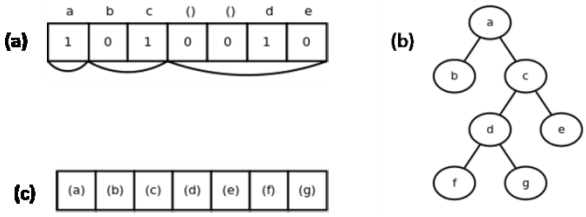


Figure 3 *Bit tree* structure and data representation (for a binary tree). The top array (a) shows bits indicating the structure of tree depicted in (b). Note that node *b* has no children, therefore its potential children must have a zero bit. The bottom array (c) shows the corresponding data items in BFS order. Note that only existing tree cells have corresponding data cells.

Each cell includes a pointer to both the parent and it's first child, and so the octree can be traversed in both a bottom up and top down manner. This representation is especially important for deep trees, as it expedites the search for the neighbors of a given cell. However, this type of cell data structure carries redundant information. Both parents and children point to each other, duplicating the same information. Furthermore, each child node points back to the same parent, which effectively repeats the same value eight times if children are placed contiguously in memory. If a tree's data is stored contiguously in memory, then storing each data pointer becomes redundant as well. Some efficient octree implementations store voxel data in conjunction with its parent cell, which removes the need to allocate storage for individual leaf voxels [4]. However, this type of data storage precludes data modularity, and forces a scene to initially allocate memory for all data associated with the octree.

In order to compress the scene's structural information, the depth of each octree is limited to four levels and is represented as a binary code. Each octree is complete, that is every node has either eight children or no children. This allows us to use a *bit tree* representation, which is an implicit data structure where each bit indicates whether or not a node has children. Assigning one bit for each potential internal node maintains a consistent structure that can represent any depth-four octree without the need for parent or child pointers. The location of a node's child bit (or a node's parent bit) can be calculated using the following formulas:

$$child(i) = 8i + 1 \tag{1}$$

$$parent(i) = \left\lfloor \frac{i-1}{8} \right\rfloor \tag{2}$$

With a depth limit of four, an octree can have at most 73 internal nodes, which allows its structure to be fully represented by 10 bytes. Reserving two bytes for a pointer to the tree's contiguous data brings the total to 12 bytes. The current implementation aligns each tree to 16 bytes, leaving the last four bytes empty.

Although each octree reserves a bit for every potential node, a data cell is only created for every existing node. In our probabilistic voxel implementation, each data cell carries two pieces of information (separated into two buffers): the parameters to a mixture of Gaussians distribution (8 bytes) and an occupancy probability (4 bytes). (see section 3 for details) Each tree's data is stored as a contiguous chunk of memory in breadth first search (BFS) order. For an arbitrary node with bit index *i*, its data location is offset by the number of nodes that come before it BFS order, which can be calculated by a simple function of the sum of the bits that occur before node *i*'s parent.

$$data\_index(i) = 8\left(\sum_{j=0}^{parent(i)-1} tree\_bit(j)\right) + 1 \tag{3}$$

2.2 Data Layout

The model is stored on the GPU in three separate buffers: a three dimensional array of block pointers (*blocks*), an array of octrees (*trees*), and an array of data cells (*data*). The *blocks* buffer

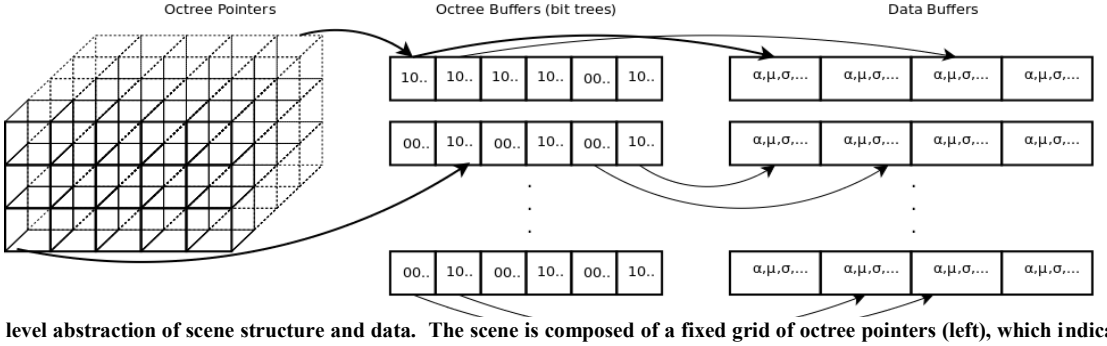


Figure 4 High level abstraction of scene structure and data. The scene is composed of a fixed grid of octree pointers (left), which indicate where the corresponding bit tree is stored in the array of octree buffers (center). The bit tree stores structural information as well as a pointer to a contiguous chunk of data. Data is always contained in the same buffer as the tree to facilitate parallel refine/merge operations.

maintains the three dimensional layout of the scene, where each element points to one octree in *trees*. Each octree defines the spatial partitioning of voxels within the block and contains a pointer to the data cell in *data* that corresponds to the tree's root. A visualization of the scene data structures is shown in Figure 3.

Separating tree structure from cell data allows for interchangeable data to be stored. For instance, multiple appearance models can be stored and used as separate *data* arrays for the same model. Though different data is stored separately, each type of data maintains the same buffer structure; indexing into the same data cell across different types of data will return data for the same voxel. Separating data from structure allows models to grow dynamically and from separate sets of data. For instance, the current implementation makes use of an occupancy probability and an appearance model stored as a mixture of Gaussians. The proposed modular data design significantly reduces model size by replacing each distribution with its expected value without having to access other data or the tree structure.

Although it may seem indirect to store a three dimensional structure of pointers to bit trees, this storage structure allows the scene to grow in parallel on the GPU. Both the node data and octree structure are dynamically updated throughout the scene reconstruction process. The reasoning for this pointer based buffer scheme will be outlined in the adaptive refinement section below.

3. Overview of Probabilistic Volumetric Framework

This section briefly describes approach of [1] to provide the readers background of probabilistic volumetric framework and for details the readers may refer to [1]. The volume of interest is assumed to be a 3-d grid of voxels for ease the understanding but extension of this to an octree have been proposed in [1]. Each voxel is assigned either of two states: a surface, S, or not a surface. The belief of a voxel, X, being a surface is denoted by a probability, $P(X \in S)$. The appearance model in each voxel is represented by a probability density $P(I|X \in S)$ which is a Mixture of Gaussian (MoG).

Each pixel of an input image, $N+1$, is back-projected into the volume along the ray, $R_{x^{N+1}}$ using a known camera, shown in Figure 5. The underlying assumption is that only one of the voxels, $V_{x^{N+1}}$, along $R_{x^{N+1}}$ produces the observed intensity in the image. The voxel $V_{x^{N+1}}$ must be un-occluded and have a high surface probability given by,

$$P^N(V_{x^{N+1}} = X_\alpha) = P(X_\alpha \in S) \prod_{\alpha' < \alpha} [1 - P(X_{\alpha'} \in S)]$$

where the product is taken over voxels closer to the sensor than voxel X_α. $p^N(I_x^{N+1}|X \in S)$ is the probability density for observed intensity for voxel X, assuming that voxel is the surface element that caused the image intensity observation.

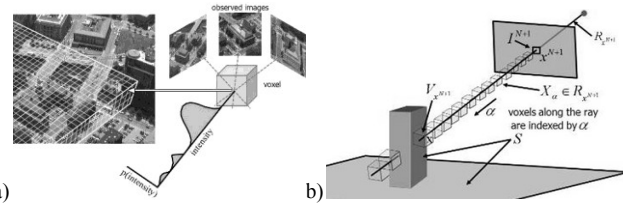
The rendering of an image pixel from the above probabilistic volumetric model is given by

$$E[I_{x^{N+1}}] = \sum_\alpha P(V_{x^{N+1}} = X_\alpha) E[p(I|X \in S)]. \quad (4)$$

The updating of a cell X, given the input image intensity $I_{x^{N+1}}$ is given by the following Bayes equation:

$$P^{N+1}(X \in S) = P^N(X \in S) \frac{p^N(I_{x^{N+1}} | X \in S)}{p^N(I_{x^{N+1}})} \quad (5)$$

The resulting 3-d model, Figure 1(b), has fine details and depicts surfaces with continuity compared to the state-of-art 3-d reconstruction based on sparse features, [11], Figure 1(c).



a) b)

Figure 5 (a) shows the grid representation of the volumetric framework and each cell of the grid stores a probability value of being a surface and an appearance distribution. (b) illustrates the interaction of a ray with the cells in the volume for rendering images as well as updating volumetric model.

4. Implementation

The cross-platform framework, OpenCL [12], is used to enable GPU model rendering and reconstruction. All major computation is done on the GPU in OpenCL kernel code with host code in C++ using the standard OpenCL API.

4.1 Ray Tracing

```
    Cast Ray: input scene, ray,
1   [t_block, t_far] = intersect_scene(ray_o, ray_d)
2   while(t_block < t_far AND visibility > .99)
3       position = ray_o + ray_d * t_block
4       blockIndex = block_index(position)
5       tree = trees[block_ptrs[blockIndex]]
6       t_exit = intersect_block(blockIndex, ray_o, ray_d)
7       t_tree = 0.0
8       while(t_tree < t_exit)
9           localPos = position + t_tree * ray_d
10          bitIndex = traverse(tree, localPos)
11          dataIndex = dataIndex(bitIndex, tree)
12          t_voxel = intersect_cell(bitIndex, ray_o, ray_d)
13          segLen = (t_voxel − t_tree) * BLOCK_LEN
14          $$cell function (update/render)$$
15          t_tree = t_voxel
16      t_block = t_exit
```

Figure 6 Ray tracing algorithm. The *pos*, *cell_min*, *ray_d*, *ray_o* and *local_ray* are vectors and operations on them are vector based. The cell level function is application specific, *e.g.*, for rendering an image, such function will estimate the appearance over all cells along the ray.

Ray tracing is a fundamental procedure which in this paper supports operations such as render, change detection, and update of the probabilistic voxel model. The scene is divided into a grid of shallow octrees, and ray tracing is carried out on both the block and tree level. The pseudo-code for the ray tracing is presented in Figure 6.

The ray tracing implementation contains two main loops: the outermost operates on the fixed grid of blocks, and the innermost

operates on an individual octree. The two functions that differ between the *bit tree* ray trace implementation and a standard octree ray trace implementation are the operations *traverse()* and *data_offset(bit_index)*. The traverse method only differs in that instead of directly storing pointers to child nodes, the bit index is calculated implicitly using Equations 1 and 2. The *data_offset* method efficiently calculates a given cell's data index (a value that would otherwise be stored in GPU memory) based on the bit tree structure and the cell's bit index. The *data_offset* method in our approach is further optimized by caching data indices in local memory as they are calculated, allowing them to be used for nodes that appear later in BFS order.

Future developments in caching entire trees and their cumulative data indices will further mitigate this computation while still enjoying the benefits of reduced data.

4.2 Rendering 3-d Model

With a reconstructed 3-d terrain model, rendering an expected image as discussed in Section 3 is given by Equation 5. This computation requires maintaining a running product of the probabilities given by

$$\prod_{\alpha'<\alpha} [1 - P(X_{\alpha'} \in S)]$$

and also maintaining a running sum of the expected intensities for each of the voxels. This is done in a selectable function which is called within the ray-tracing function as shown in line 14 of the pseudo code in Figure 6. Since each of the cells along the ray has non-zero probability, every cell can potentially contribute to the pixel value for a ray leading to numerous intersections per ray.

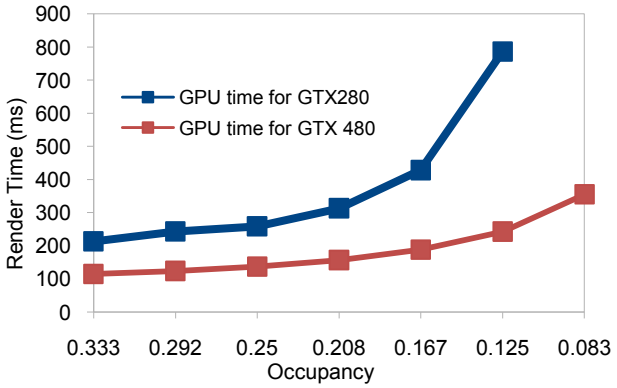


Figure 7 This plot shows the function of render time for GTX280 and GTX480 as a function of fraction of number of threads allowed per processor. Higher number of threads allow to hide latency for memory accesses.

The major optimization approaches to increase the performance of the rendering was mainly due to (i) minimizing redundant floating point operations, (ii) increasing memory bandwidth for reading octrees and (iii) reducing the usage of registers and local memory to increase the number of threads working at the same time. First, a lot of redundant floating point computation was minimized such as only the exit point of a cell was computed which would be the entry point for the next cell. Second, the memory bandwidth was increased due to small size of an octree. A 4-level octree is now encoded in bits amounting to less than 16 bytes. The bit octree is read once and stored in the local memory for faster accesses. A third optimization step increased the number of threads available and enabled the global memory latency to be masked. The performance as a function of number of threads per streaming processor is plotted in Figure 7. The rendering time decreases as the number of threads increases. The number of active threads in the GPU is determined by the amount of local memory and register memory allocated for each kernel. For example, if a workgroup of 64 threads share 4 KB of local memory, then a card with 16 KB local memory per streaming multiprocessor will be restricted to executing 4*64 threads or 8 warps. A multiprocessor on the NVIDIA GTX280 can execute 32 warps, however local memory restrictions will reduce its occupancy to 25%. Reducing the local memory and register footprint of a kernel can dramatically accelerate execution.

These optimizations led to a speed-up of factor of at-least 600 times over C++ implementation. The current approach can render 640x480 images at 30fps, 1280x720 images at 10 fps and 3500x3000 images at 1 fps.

4.3 Updating 3-d Model

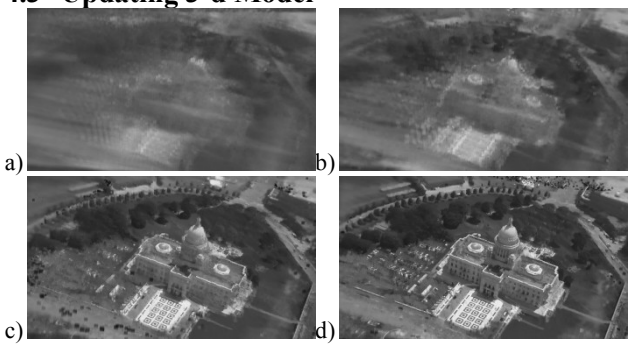


Figure 8 This figure shows the model being dynamically updated after a) 7 images, b) 14 images, c) after 56 images and d) after 119 images.

The 3-d model reconstruction approach from [1] is discussed in Section 3. The evolution of the model is shown by rendering the model from same viewpoint after some number of updates, Figure 8. The process of updating the model requires 3 ray-tracing passes as compared to one for rendering. The major bottleneck for updating is serial writing of the data associated with each octree cell The serial access is required since multiple threads may need to update the same data element.

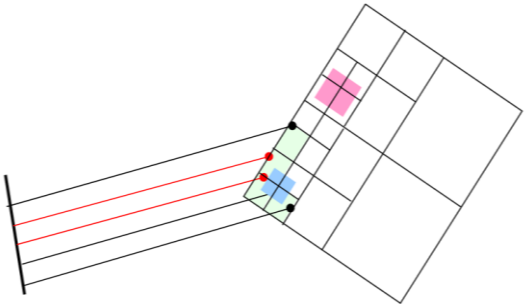


Figure 9 This figure illustrates multiple rays (shown in red) passing through the same cell in top-down view of an example.

The write operation is slower as multiple threads writing to the same location have to serialize the write instruction unlike the read operation. Multiple rays can pass through the same cell as illustrated in Figure 9, and might try to write at the same time which can overwrite data from other threads. OpenCL provides

hardware implementation of atomic functions which are serialized by the hardware. These operations are safe but slower.

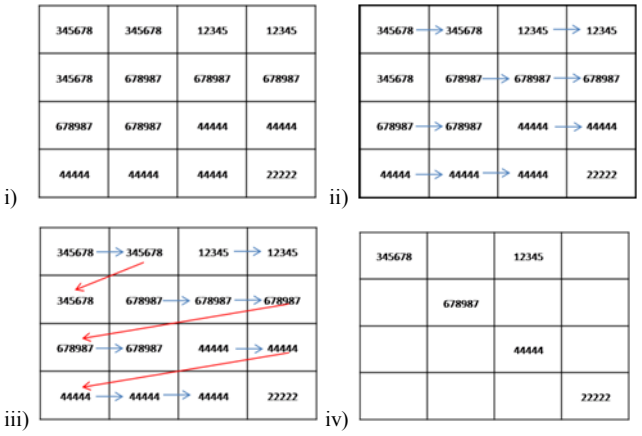


Figure 10 This figure illustrates a novel parallel algorithm to segment the threads for writing data to the same location when data is shared between multiple threads. This algorithm has worst case complexity of *O(log n)* as compare to serial algorithm with complexity of *O(n).*

Instead of performing atomic operations per thread, a strategy comprising of performing atomic operations per block of threads is proposed in this paper to achieve the desired performance. The block of threads is called workgroup and the member threads can exchange data via local memory which requires two clock cycles to read as compared to one clock cycle for registers. The threads which are writing to the same cell in the same workgroup are grouped using a novel parallel segmentation algorithm and only one of the threads of the group executes atomic operation.

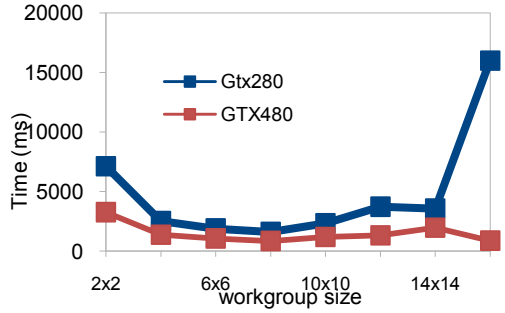


Figure 11 This plot shows the performance of caching algorithm with variation in number of threads in a workgroup for GTX280 and GTX480. Note that the optimal size seems to be 8x8 which has been used in all the results for this paper.

The parallel segmentation algorithm is described briefly using an example workgroup of 16 threads accessing different or same cells as shown in Figure 10(i). The algorithm has two stages. During the first stage each of the threads link to their right neighbor, Figure 10(ii) if they have the same cell ID. During the second stage the cells which did not link to the right neighbor will search for the cell with same id in the lower row and if found will connect to the cells shown in red arrows in Figure 10 (iii). The cells which were never linked to form the leader of their respective group shown in Figure 10 (iv). These leaders only write data to the global memory and avoid any conflicts. This approach reduces a serial implementation of *O(n)* to a parallel implementation with worst case of *O(log n)*. The performance of such caching algorithm depends on the size of the workgroup which is plotted in Figure 11. Note that the optimal workgroup size is 8x8 which has been used for the results for this paper.

Each leader performs an atomic operations which are serialized by the hardware of the GPUs. The atomic operations such as *atomic_incr* and *atomic_add* ensure that no race condition of multiple threads overwriting at the same address occurs. However, at the time of writing only int32 and int64 atomics have been implemented in OpenCL, forcing discretization of floating point numbers into integers. This quantization is done with minimal loss of accuracy by predicting practical maximum values for different quantities required for Bayesian update, Equation 5.

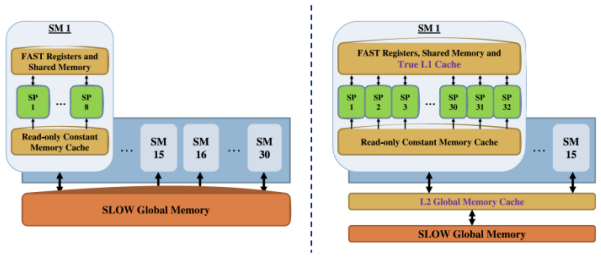


Figure 12 Reproduced figure from [13] that illustrates the difference in architecture between older generation GTX200 series (left) and newer generation GTX400 (Fermi) series cards. Note the increase in processors per streaming multiprocessor and the addition of a quicker L2 global memory cache. For updating the 3-d terrain model the L2 global memory cache significantly speeds up atomic function calls.

Using atomic functions also comes at a computational price. Serializing access to global memory drastically reduces the speed of execution on the GTX280. However, there exists an extra layer of L2 cache above global memory on the GTX480 (Fermi) cards which significantly speedup atomic functions [13].

4.4 Refine/Merge

Octree cells that have developed a sufficiently high probability of being a surface are refined into eight new cells to obtain the desired resolution. This process of refining is critical so as to develop the model to the finest resolution possible. The difference between a coarse and refined model is shown in Figure 9. Because memory cannot be dynamically allocated on the GPU, data is stored on pre-allocated buffers with a known max size. When a tree's cell needs to be refined, all of a tree's data is moved into free space on the buffer. An array of memory pointers keeps track of the free space on each buffer.

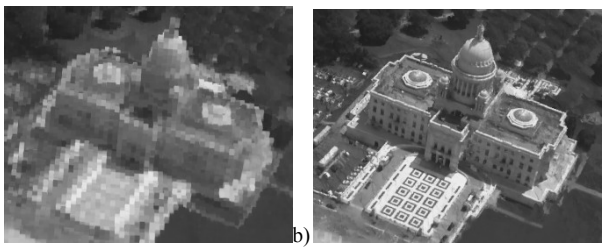


Figure 13 This figure shows the difference between a) a coarse model and b) a refined model. The refined model allows the model to built to a finer resolution.

While refining the leaves of an octree seems naturally parallel, the process is hindered by the GPUs lack of thread-safe dynamic memory allocation. Because of this limitation, we use pre-allocated buffers with empty space into which new data can expand. However, in order to refine multiple trees in parallel, it must be ensured that no two trees place data into the same spot in

global memory. To accomplish this, we implemented a simple, parallel memory allocation scheme that divides data and trees into an array of smaller buffers while keeping track of the location of free cells in each buffer. A tree and its corresponding data always lie in a buffer with the same index. The buffers can be refined in parallel (with the trees in each buffer being serially refined) without any memory conflict.

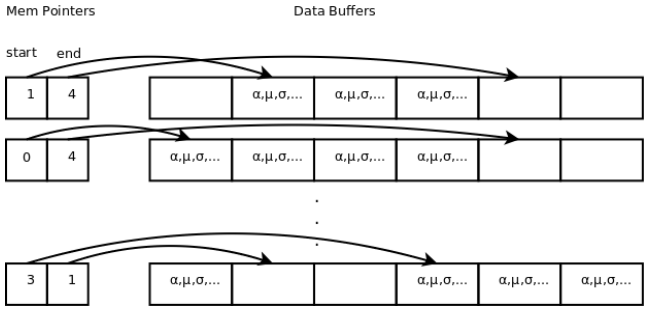


Figure 14 Memory pointers keep track of free space in each data buffer. This is necessary because all buffers must be pre-allocated before a kernel is executed.

However, splitting trees and data up into buffers can be an inefficient way to store octree information. For example, if the octrees near a large surface are all put into the same small buffer, the depth to which the trees could refine would be limited by the length of the buffer which reduces the fidelity of the model. Likewise, if octrees in empty space are placed into a buffer, they would never refine and end up wasting the majority of the pre-allocated space. This inefficiency is mitigated by randomly distributing the trees throughout the buffers. With scene sizes on the order of 2.4 million trees (192x192x64), uniformly random distribution is quite efficient

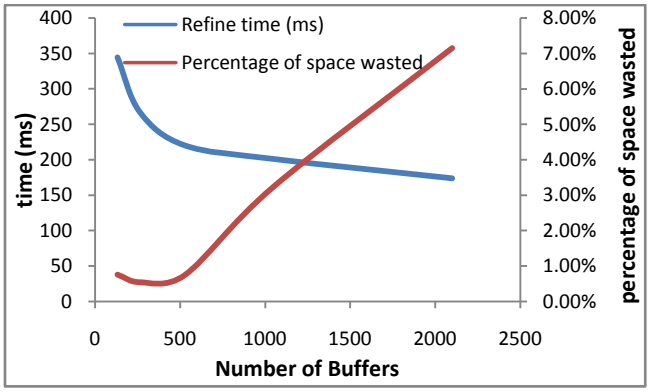


Figure 15 This plot shows the trade-off between improving the time taken at the expense of wasting storage space for refining the model in parallel. As the number of buffers are increased, the time taken decreases, but the space wasted increases.

Wasted buffer space is also reduced by merging cells. When eight sibling cells have converged to an adequately small occupancy probability, they can be merged back into the parent cell without loss of information. When this operation is done after the model has converged, significant space can be saved. For example, after the downtown scene (1512 by 1512 by 512 voxels) is updated by 300 images, one pass of cell merging results in a reduction of about 100 megabytes of space or about 10 percent of the model as it exists in the GPU global memory.

5. Speed and Memory Analysis

The performance of the OpenCL implementation has been evaluated on multiple computing resources namely Intel I7 980x (a six core processor), ATI 5870, NVIDIA GTX 280 and NVIDIA GTX 480[1]. The test model is a 480 megabyte reconstruction of downtown Providence, RI. The total number of octree cells was about 3.76e7, and the total number of highest resolution cells is 1.2e9 (i.e., 1536 by 1536 by 512). about the cell memory usages is on average 13.38 bytes/cell. The expected image render speeds vary from viewpoint to viewpoint as some views intersect more cells than others. Table 1 compares execution times of rendering, updating and refining the downtown model on six systems[1]: C++ implementation on a single core of an Intel i7 980x clocked at 3.33 GHz, OpenCL implementation on six-core of Intel i7 980x, ATI 5870 (2.72 Tflops), NVIDIA GTX200 (0.7 Tflops) and NVIDIA GTX400 series (Fermi) GPU (1.35 Tflops). Because of the variance in rendering times, one oblique view and one near nadir view are listed. Update time also varies with model complexity, so an empty update and a near converged update are both listed. The update time for the GTX280 is missing because the card has inadequate memory to update the fully converged downtown scene model.

Method		CPU (i7) C++ single threaded	CPU OpenCL (Intel i7-980x)	ATI 5870	GTX 280	GTX 480
RENDER	oblique	89.65 s	2.23s	0.27s	.213 s	.114 s
	nadir	51.45 s	1.45s	0.15s	.118 s	.062 s
UPDATE	converged	592.3 s	18.92s	n/a	n/a	1.363 s atomic 1.990 s caching
	empty	175.6 s	8.105s	4.723 s atomic 4.366 s caching	16.14 s atomic 1.81 s caching	.934 s atomic .795 s caching
Refine		1.14 s	0.045	0.943s	.52 s	.242 s

Table 1 Comparison between C++ on single core CPU and OpenCL implementation on CPU and GPU for volumetric modeling implementations[1]. Render times were compared for two different views. Update times were compared for nascent and near converged scenes. The refine comparison split 54009 leaves.

The approach in this paper is compared another state of the art GPU ray tracing approach, [4]. Although both approaches use voxel-based ray tracing with octree-based spatial partitioning, the comparison between Laine and Karras' [4] approach based on a discrete geometry representation and this papers' approach based on the much larger data requirement of the probabilistic model is not well defined. The probabilistic ray trace technique intersects with each voxel in the path of a ray, whereas Laine and Karras' stops after the first intersection. The model in [4] consists of a single deep octree, while the probabilistic model contains a fixed

[1] The OpenCL implementation was well optimized for NVIDIA graphic cards and have been run on the CPU and ATI card without any optimization for respective platforms.

grid of many shallow trees. Rendering a probabilistic model is inherently more computationally complex.

Approach	Mean number of Cells/Ray	Computational rate for Cells/Sec
Laine and Karras, [4]	41.66	4.29e9
Current implementation	422.24	3.43e9

Table 2 Comparison of two voxel octree based ray trace rendering algorithms.

Nonetheless, a comparison based on two metrics is drawn: (i) number of cells intersected per ray and (ii) number of cells computed per second. Number of cells intersected per ray captures the number of cells that each ray pierces while ray tracing. The disparity in this number is due to the denser nature of probabilistic data. The number of cells computed per second is the number of cells pierced by the entire ray tracing program as a function of time. The number of cells computed per second using the proposed approach is comparable to the rate achieved in [4] even though much more computation is done per cell for the probabilistic model. Most of the cells traversed in [4] are empty and no computation is performed.

6. Conclusion & Future Work

A GPU-based implementation of a probabilistic, voxel-based 3-d model has been described. The implementation includes a grid of octrees spatial partitioning structure coupled with an optimized ray tracing algorithm that can be used to render expected images as well as update the voxel model. An efficient bit tree representation is developed to reduce the memory size of the structure of the octree from about half to less than five percent of model information stored in GPU.

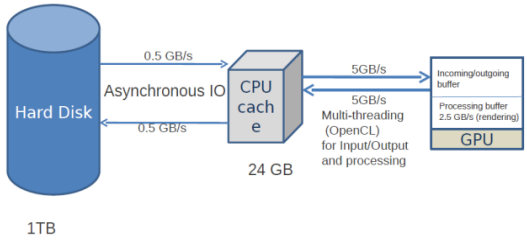


Figure 16 Proposed streaming memory setup to handle larger volumetric scenes. A hard disk serves as the first level of cache from which scene data can be loaded asynchronously into CPU memory. The CPU memory serves as the second layer of cache, storing immediately relevant scene information. The GPU simultaneously processes existing data and loads data from the CPU cache into its global memory, serving as the highest level cache.

Like all volumetric models, the large model size and limited available GPU memory leads to code design challenge. The current implementation only supports update and rendering for models that can fit entirely on the GPU, which limits model size and resolution. In order to overcome such limits, future work will involve a streaming model that simultaneously processes and reads/writes model data to the GPU. The current CPU to GPU memory bandwidth on the NVIDIA GTX 480 is 5 GB/s. The rate of data processing for rendering is roughly 4 GB/s, which indicates that the rate of transfer of the data is sufficient to match the computational time and thus hide the latency. This approach will support continuous processing on much larger models and images. Figure 16 presents a proposed streaming data system for large-scale model processing.

The authors also plan to pursue future optimizations to speed up ray tracing. Currently, internal caches of GPUs are not used which could be used more efficiently by storing the 3-d grid as 3-d image object in GPU. A heuristic method for pruning the blocks from the computation will be investigated. This approach includes accelerating traversal or skipping empty blocks. If a block remains unrefined, and its occupancy probability is vanishingly small, the proposed rendering algorithm can skip it without loading its appearance model from global memory. Perhaps the most significant improvement will be to include a multi-resolution scheme for rendering blocks at a distance. For example, if a cell back-projects to less than a pixel, traversal could terminate early. Additionally the data should be reorganized such that uniform regions use courser samples, while more textured regions will be more precisely sampled. Such level of detail (LOD) methods are in use for rendering large polygonal models.

Acknowledgements: The work proposed in this paper has been funded by DARPA contract #N10PC20126.

7. References

1. **Crispell, Daniel E.** Probabilistic Geometry and Appearance Modeling from Aerial Video. 2009.

2. **Pollefeys, M., et al.** Detailed real-time urban 3-d reconstruction from video. *International Journal of Computer Vision.* 2008, Vol. 78, 2-3.

3. **Pollard, Thomas and Mundy, Joseph L.** Change Detection in a 3-d world. *Computer Vision and Pattern Recognition, IEEE Computer Society.* 2007.

4. **Laine, Samuli and Karras, Tero.** Efficient Sparse Voxel Octrees -- Analysis, Extensions, and Implementation. *NVIDIA Technical Report.* 2010.

5. **Crassin, Cyril, et al.** GigaVoxels : Ray-Guided Streaming for Efficient and Detailed Voxel Rendering. *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D).* 2009.

6. **Knoll, A., Hansen, C. D. and Wald, I.** Coherent Multiresolution Isosurface Ray Tracing. *he Visual Computer.* 2009, pp. 209--225.

7. **Pajarola, Renato and Gobbetti, Enrico.** Survey of semi-regular multiresolution models for interactive terrain rendering. *The Visual Computer.* s.l. : Springer Berlin / Heidelberg, 2007. Vol. 23, 0178-2789.

8. **Westermann, R., Dick, C. and Kruger, J.** GPU Ray-Casting for Scalable Terrain Rendering. *Proceedings of Eurographics 2009 - Areas Papers.* 2009.

9. **Samet, Hanan.** Implementing ray tracing with octrees and neighbor finding. *Computers And Graphics.* 1989, Vol. 13, pp. 445--460.

10. VXL: computer vision libraries. [Online] vxl.sourceforge.net.

11. **Noah, Snavley, Szelisky, Richard and Seitz, Steven M.** Modeling the World from Internet Photo Collections. *International Journal of Computer Vision.* 2008, pp. 189-210.

12. The OpenCL™ Specification 1.0 (rev 48). *Khronos* . s.l. : Editor Aaftab Munshi, 2009.

13. **Alerstam E, Yip Lo WC, Han TD, Rose J, Andersson-Engels S, Lilge L.** Next-generation acceleration and code optimization for light transport in turbid media using GPUs. *Biomedical Optics Express.* 2010, Vol. 1, 2.