GPU-based Adaptive Octree Construction Algorithms

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, many of which work on fundamental octree based clustering. Parallelizing the construction of octrees is thus of immense importance with respect to its applicability.

This paper presents two different ways for constructing octrees on GPUs and reports average speed-ups of 100 *than their CPU counterparts. We evaluate our algorithms qualitatively and quantitatively and finally use them in a compute-intensive problem of finding radiosity based Global Illumination solution for point models using the Fast Multipole Method as a proof of its correctness and applicability.*

1 Introduction

Octree is one of the numerous hierarchical data structures, based on recursive domain decomposition used to cluster spatial data (for brevity, we assume points) in meaningful groups. Octrees have applications in vast majority of fields which are computationally intensive or problems which require quick response. More concretely, consider the application areas enlisted 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; Classic algorithms such as the Fast Multipole Method or the Barnes-Hut simulation, use the hierarchical octree structure to divide the volume into cubic cells, so that only particles from nearby cells need to be treated individually, 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). Using the hierarchical structure and spatial indices can thus dramatically reduce the number of particle pair interactions that must be computed.

VISIBLE SURFACE DETERMINATION [8, 7, 5] It is the process used to determine which surfaces and parts of surfaces are not visible from a certain viewpoint (view-dependent) or from all points in the model (view-independent). They make use of octrees to subdivide the scene's space for visibility determination to be performed hierarchically: Effectively, if a node in the tree is considered to be invisible then all of its child nodes are also invisible, and no further processing is necessary.

COLOR QUANTIZATION FOR IMAGES It is a process used for efficient compression that reduces the number of distinct colors used in an image with the intention that the new image should be as visually similar as possible to the original image. It can be viewed as a data-clustering problem where the points represent colors in the original image and the three axes represent the three color channels. The representative color of each cluster can be used for the output image. Octrees are an ideal solution for performing such clustering.

COLLISION DETECTION Highly used in physical simulations and video games, this algorithm requires to have real-time response. Object-based sub-division of space using octrees helps to check collisions directly for complex objects (as whole) rather than for each basic primitive used (for constructing that object) and thereby help speed-up the process.

Construction and traversal of the ubiquitous octree on a CPU is well understood. However, parallelizing the construction and traversal of such octrees can provide very high speed-up gains for such compute-intensive problems.

GPUs have evolved into a very attractive [10] hardware platform for general purpose computations due to their extremely high floating-point processing performance, huge memory bandwidth and their comparatively low cost. This paper is concerned with constructing octree on the GPU. A parallel implementation of octree on GPU appeared in [3] which improved on the previous algorithm of [9]. However, a complete octree ($\sum_{i=0}^{l} 8^i$, where l represents the maximum depth) with empty nodes is stored in [3], there making them *memory-inefficient*.

PRINCIPAL CONTRIBUTIONS:

1. We present an algorithm to construct octrees, in parallel, on the GPU, using CUDA. It performs *data-independent* clustering of points (useful in N-body simulations) and report upto 100 fold speed-up. Basic queries are suitably answered.

2. A different, *data-dependent* parallel octree construction algorithm (used for color quantization, collision detection, visibility determination etc) is given and 100 fold speed-ups reported.

The rest of the paper is organized as follows. For completeness, a brief overview of the NVIDIA's $G80$ GPU and CUDA is given in § 2. Details of SFCs and compressed octree are outlined in §3 which are useful for data-independent parallel octree construction algorithm presented in §4. The data-dependent parallel octree construction algorithm appears in §5. §**??** summarizes the usefulness of both our algorithms with respect to its applicability. Quantitative and qualitative results along with some run-time GPU based optimizations are explained in § 6. We follow this up with some concluding remarks and work to be done in future in § 7.

2. GPU Programming Model

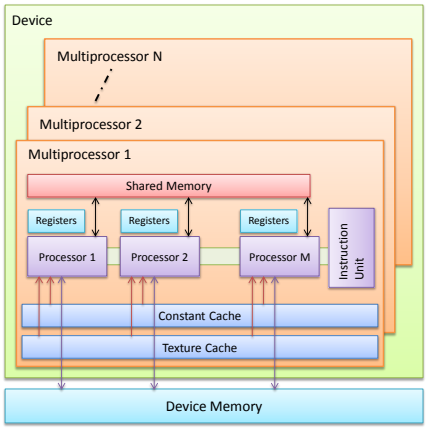


Figure 1. Hardware Model of GPU

NVIDIA's G80/G92 architecture GPUs are typical of current generation graphics hardware which uses a large number of parallel threads [2] 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

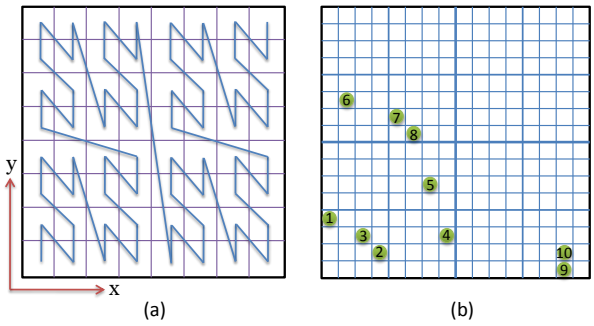


Figure 2. Z-Space Filling Curve

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 SFC and Compressed Octree

SPACE FILLING CURVES: SFC provides an easy to implement, parallelize and a good load-balanced domain decomposition technique useful in linearization of data living in 2D or 3D spaces.

Say, our spatial data lies in some d dimensional hypercube. This hypercube if bisected k times recursively along each dimension, results in 2^{dk} non-overlapping hypercells of equal size. The SFC is a mapping of these hypercells to a 1-D linear ordering. We use a 2-D Z-SFC as shown in Fig. 2(a). Fig. 2(b) shows 10 points in a 2-D space which are sequentially labeled in the Z-SFC order.

Consider a 3-D particle space of sidelength D and let its bottom left corner be at the origin. Given a point $(P_x,\ P_y,\ P_z)$ in the model, the integer coordinates of the cell to which it belongs will be $(\lfloor 2^k P_x/D \rfloor,\ \lfloor 2^k P_y/D \rfloor,\ \lfloor 2^k P_z/D \rfloor)$ [3]. The Z-SFC index of the cell is now computed by representing these

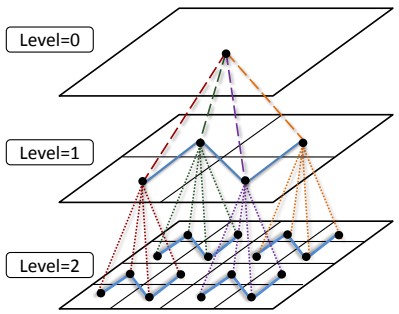


Figure 3. Octree as Multiple SFCs at Various Levels

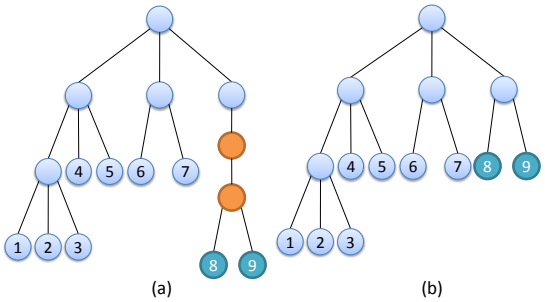


Figure 4. (a) Octree, (b) Compressed Octree

integer coordinates using k bits for each dimension and interleaving these bits. (The SFC index of, e.g. a cell with coordinates (3, 0, 2) = (11, 00, 10) is 101100).

If the computed SFC values (at any fixed resolution) are sorted, then we have the correct order to consider nodes in a bottom up traversal of an octree. *Octrees can thus be viewed as multiple SFCs at varying resolutions* (see Fig. 3). Further, removing the least d bits from the value of a cell gives the value of its parent. A linear bottom up octree construction is therefore easy if we follow the SFC order. *A nice property that follows is the resulting linearization of all cells in an octree (or compressed octree (please refer below)) sorted by the SFC order gives us its postorder traversal*. For more information on SFCs, we guide the reader to [11].

COMPRESSED OCTREES: In octrees, if the manner in which any subregion is bisected is independent of the specific location of the points within it, chains may form when many points lie within a small volume of space. An example of a chain formed due to close points labeled 8 and 9 is as shown in Fig. 4. These points can be separated only after several recursive subdivisions. Though nodes in these chains represent different volumes of the underlying space, they do not contain any extra information and hence can be compressed, thereby forming a *compressed octree – an octree without chains*.

Note that each node in a compressed octree is either a leaf or has at least two children. This ensures that *every internal node is a Least Common Ancestor (LCA) of some leaf-pair*, a property which is useful for our parallel octree construction algorithm of § 4. For more information on compressed octree please refer [4].

4 Parallel Bottom-Up Adaptive Octree

We present the parallel, data-independent, bottom-up SFC based octree construction algorithm along with some implementation details. 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. Before heading on, here are some of the intuitions behind the algorithm design.

1. BOTTOM-UP TRAVERSAL: Since every internal node in an octree has leaves in its subtree, given the leaves we can somehow decode this hierarchical inheritance information and generate the internal nodes.

2. PARALLEL STRATEGY: Each internal node can be considered as a LCA of some particular leaf pairs (in a compressed octree). Thus, given the leaves, generation of internal nodes can be parallelized since each of them can be generated independently from a leaf pair. Many leaf pairs might have the same LCA node resulting in duplicates which can be easily detected and removed.

3. Parent-Child relationship can be established and octrees can be generated from a given compressed octree using SFC indices across multiple levels.

The algorithm, with the help of Fig. 5, along with the implementation details is presented next.

1. CONSTRUCTING LEAVES

   (a) Read n points in the first n locations of an array A of size $2n-1$. As shown in Fig. 5(a), we have 8 input points in this example.

   (b) Assuming a point per leaf, for every point, *in parallel*, do

      i. Generate the $3D$ co-ordinate of leaf cell to which it belongs (§3).

      ii. Generate SFC index (§3) for the leaf cell as shown in Fig. 5(a). For in-depth parallel GPU based SFC construction algorithm, please refer [3].

   (c) Sort [6] the first n elements of array A, *in parallel*, based on SFC indices of leaves (Fig. 5(b)).

2. GENERATING INTERNAL NODES & POST ORDER: *In Parallel*, for every adjacent leaves, find their LCA using the common bits (multiple of 3; 3 being the dimension) in their SFC indices. For eg. say adjacent leaves L_1 and L_2 have their SFC indices as 100 101 1100 10 and 100 101 100 001 respectively, then the LCA is the internal node having SFC index 100 101

   (a) Allocate $n-1$ GPU threads.

   (b) For every two adjacent leaves (say at locations i and $i+1$) in array A, *in parallel*, generate the internal node and store it at location $n+i$ in array A (Fig. 5(c)).

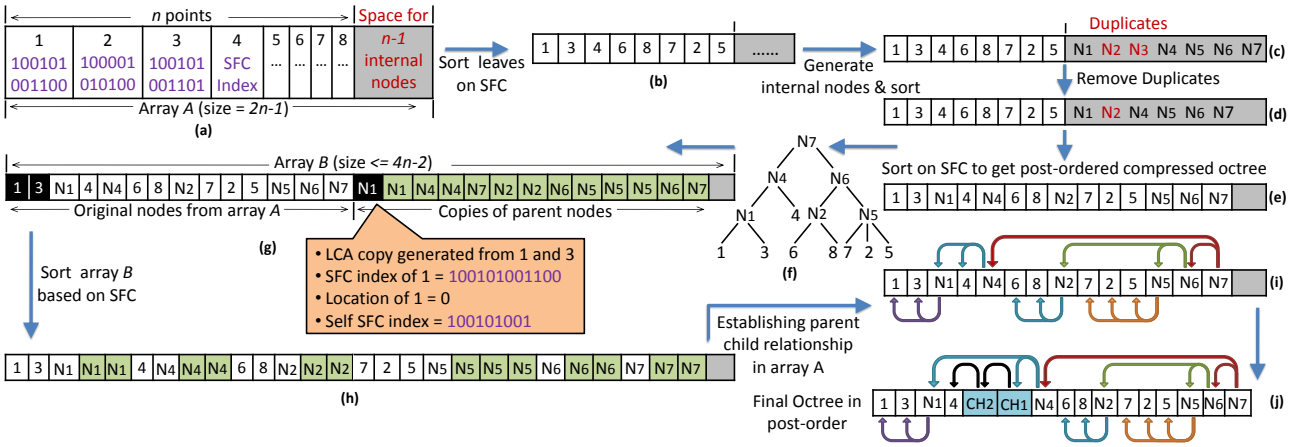


Figure 5. Algorithm 1

(c) Sort [6], *in parallel*, the internal nodes generated, across levels based on their SFC indices. To do the same, we need to establish a total order on the cells across levels. If one is contained in the other, the subcell is taken to precede the supercell; if they are disjoint, they are ordered according to the order of the immediate subcells of the smallest supercell enclosing them. Fig. 5(c) shows sorted internal nodes with duplicates (N_2 and N_3) which might be generated.

(d) Allocate $n-2$ threads for a maximum of $n-2$ consecutive internal node pairs in the later half of array A to remove the duplicates.

(e) For every two adjacent internal nodes not having same SFC indices, *in parallel*, traverse back in the later half of array A starting from the current node to look for its duplicates and eliminate them (Remove node N_3 as shown in Fig. 5(d)).

(f) Sort array A, *in parallel*, based on SFC indices across levels to get the postorder traversal of a compressed octree (§3). (Fig. 5(e)).

Here we note that there might be some empty elements at the end of array A after sorting (the *gray* shaded area in Fig. 5(e)). We can not avoid this situation since CUDA does not support dynamic memory allocation and deallocation. So, an array of maximum required size $(2n-1)$ has to be declared at compile time only.

3. PARENT-CHILD RELATIONSHIP: The compresed octree represented by this post-ordered array A is shown in Fig. 5(f). The tree is shown only for the purpose of illustration as the parent-child relationships are still not established. To generate the parent-child relationship in the compressed octree, an intuition would be *since the tree is in the post order fashion*, a LCA of every two adjacent nodes would definitely

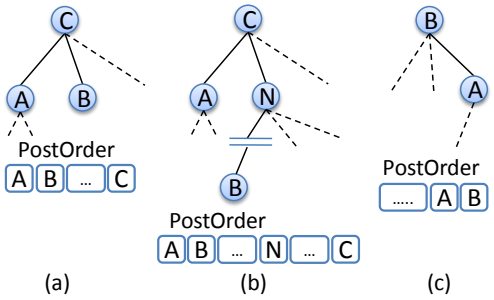


Figure 6. Computing Parent and Child

be the parent of the first node in the pair considered. Three possible cases are shown in Fig. 6.

Fig. 6(a) shows a case where both nodes A and B are siblings. Hence the LCA is the parent of A i.e. C. Fig. 6(b) is a case where B is the first node in the post-order fashion in the subtree of the node N, adjacent to A. Again their LCA i.e. C is the parent of A. Third case shown in Fig. 6(c) is where, given two adjacent nodes in post-order fashion, node B is the parent of A. Hence their LCA is B.

Thus, considering every two adjacent nodes in post-ordered compressed octree, and generating their LCA gives us the SFC index of the parent of the first node in the pair, thereby establishing the parent-child relationship. Here are the implementation steps perfomed on GPU (Fig. 5(g)).

(a) Allocate an array B twice the size of the number of leaves and internal nodes (atmost $4n-2$). Copy the first half of array B with the current post-ordered array A of leaves and nodes.

(b) Allocate threads one less than the $(NumberOfLeaves + InternalNodes)$.

(c) For every two adjacent nodes in the first half of array B, *in parallel*, do

i. Generate the LCA from the SFC indices.

ii. Copy the new node (copy of the parent of the first node in the pair considered) into the corresponding location in second half of the array B. (Generated copies of the nodes are shown in *green* in Fig. 5(g)).

iii. Write in this new node, the SFC index of the first node of the node-pair which generated it, along with the location of that node in array A. This location information will eventually give the index of the child this parent node-copy was generated from. Fig. 5(g) shows an example of the same. We expand the copy-node N_1 (in *black*) generated by leaves 1 and 3 (both shaded in *black*) and show the information it stores (Information box shaded in *orange*).

(d) Sort array B across levels, *in parallel* based on newly generated SFC indices. All the parents and their copies will come together (Fig. 5(h)).

(e) For every two adjacent nodes both having same SFC indices and atleast one of them not being a generated copy, *in parallel*, do

i. Establish the parent-child relationship. Here we see that one of the nodes is the original node and another is its copy (generated in step 3(c)(ii)). The copy will give the location of the child in array A while we get the location of the parent from the original (Fig. 5(g) and Fig. 5(h)).

ii. Scan ahead in array B and repeat step 3(e)(i) for all the copies of the original to establish the relationship between the parent and *all its children*. Step 3(e)(i) will be repeated atmost 7 times since in an octree, a parent can have atmost 8 children. Referring Fig. 5(h) and Fig. 5(i), step 3(e)(i) will be repeated twice for N_1 since we have two generated copies (and hence two children) of it. Similarly step 3(e)(i) will be repeated twice for N_4, N_2, N_6 and N_7 while thrice for N_5 since it has 3 children.

4. CREATING OCTREE FROM COMPRESSED OCTREE: We now move on to the final step of our algorithm where we need to generate octree from compressed octree. Consider two adjacent nodes, say A and B with A being the child of B, calculate the difference of octree depths between the two using their SFC indices, and finally add those many intermediate nodes in the chain between A and B. For eg. if A and B have SFC indices as 100 110 010 011 and 100, then the level difference is 3 (B is at depth 1 while A at depth 4, assuming root at depth 0).

This difference indicates a chain of nodes between A and B which are missing in the compressed octree, as shown in Fig. 7. This chain can now easily be generated, thereby giving us the final octree. The implementation steps are summarized below.

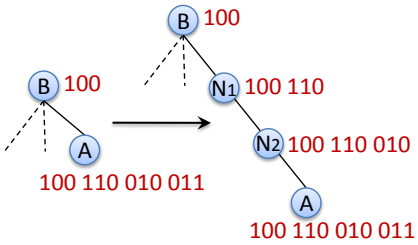


Figure 7. Compressed Octree to Octree

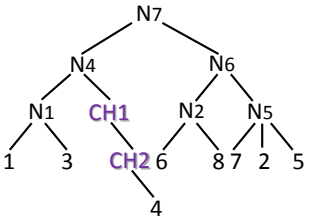


Figure 9. Final Generated Octree

(a) Allocate threads equal to size of current array A i.e. (no. of leaves + no. of internal nodes). Array is in post-order fashion.

(b) For every node calculate the level difference w.r.t its parent i.e. (level of node - level of parent - 1). This level difference gives us the count of memory needed between these two nodes.

(c) Do *parallel prefix* [6] on the level differences calculated in the step above and store at each location in A to get the total amount of memory needed to insert the internal nodes so as to make an octree (due to no support for dynamic memory allocation on GPU). While doing parallel prefix, keep track of number of nodes to be inserted before the current one, so that the index or the array location for the new node to be inserted can directly be identified.

(d) Allocate required memory for new nodes.

(e) Allocate threads equal to the size of current array A minus 1.

(f) *In parallel*, check for every node having a level difference greater than 1 with its parent, and generate new nodes to be inserted after the current node. Write them in the array location decided in step 4(c) above. As shown in Fig. 5(j), we add two *chain nodes* CH_1 and CH_2 between N_4 and 4 to get a complete octree as shown in Fig. 9.

DISCUSSION Maximum memory required for implementation is just $4n - 2$ for storing array B. It is far less than occupied by octree implementation in [3]. Our implementation is slightly slower than one presented in [3]. However, the advantage we gain due to high memory saving out-peforms the timing comparisions between them.

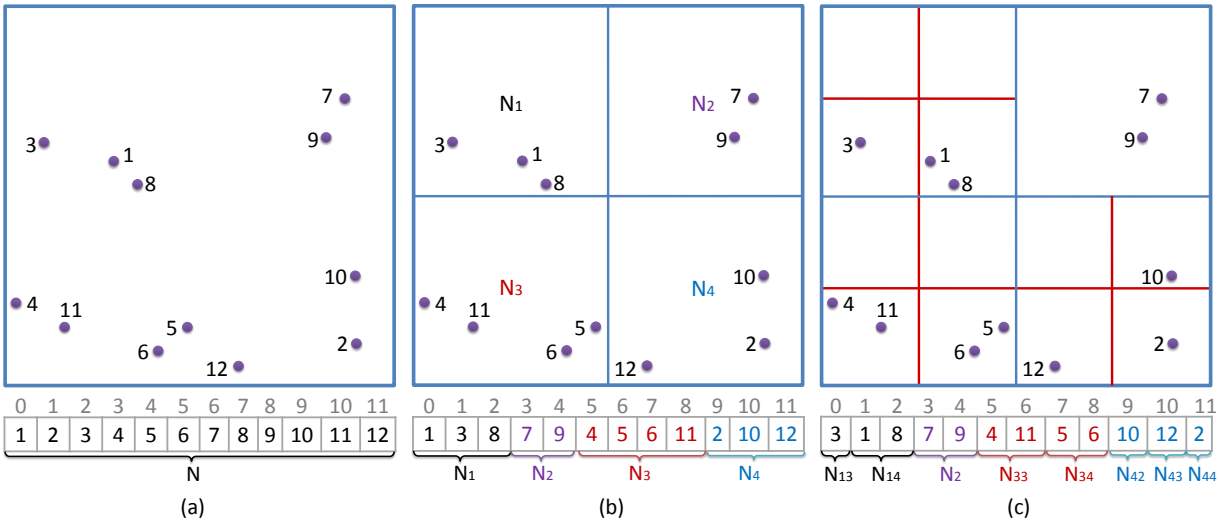


Figure 8. Spatial Clustering of Points

Further, we easily win against the same algorithm implemented on the CPU. It is a nicely load-balanced algorithm as each thread does almost the same amount of computations through out the algorithm. Here are some example queries our octree supports and solution for the same.

PARALLEL POST-ORDER TRAVERSAL: Since our output is in post-ordered form, this query is implicitly answered.

PARENT-CHILD RELATIONSHIP: For dimension d and level l, if dl is the number of bits in the SFC index representing child C_1, then the parent can be *directly* given by its first $d(l-1)$ bits.

GIVEN A POINT $(P_x,\ P_y,\ P_z)$, FIND WHICH NODE IT BELONGS TO: The co-ordinates of the desired node are $(\lfloor 2^k P_x/D \rfloor,\ \lfloor 2^k P_y/D \rfloor,\ \lfloor 2^k P_z/D \rfloor)$, where k is the number of times the space has been bisected and D is the sidelength of space enclosing all points in the model.

IS NODE C_1 CONTAINED IN NODE C_2? C_1 is contained in C_2 if and only if the SFC value of C_2 is a prefix of the SFC value of C_1

GIVEN C_2 AS A DESCENDANT OF C_1, RETURN CHILD OF C_1 CONTAINING C_2 For dimension d and level l , dl is the number of bits representing C_1.The required child is given by the first $d(l+1)$ bits of C_2.

LEAST COMMON ANCESTOR OF NODES C_1 AND C_2: The longest common prefix of the SFC values of C_1 and C_2 which is a multiple of dimension d gives us the least common ancestor.

Many other such basic queries (like *Neighbor Finding*, *Leaves in a node's sub-tree* etc.) can be supported.

5 Parallel Top-Down Adaptive Octree

We now look at a new and quite a different way to generate an octree in parallel. The problem setting is same as that in §4 but as opposed to algorithm of §4, 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. 8.

Here in Fig. 8(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. 8(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 in the array (Implementation fact: we swap the pointers, not the actual data) so that they cluster together as shown in the array of Fig. 8(b). We iteratively repeat this process till we have less than some pre-defined points (2 as in Fig. 8) in a node and term it as a leaf. Fig. 8(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. 8, 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.

Here we move down level by level, starting from the root. This intuition on building the octree can easily be extended to a parallel algorithm as each of the partitions per iteration can be generated in parallel. 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 for the next level and so on. *The degree of parallelism increases as we move down to the greater depths of the octree generation process*. The algorithm with implementation details is as sketched below.

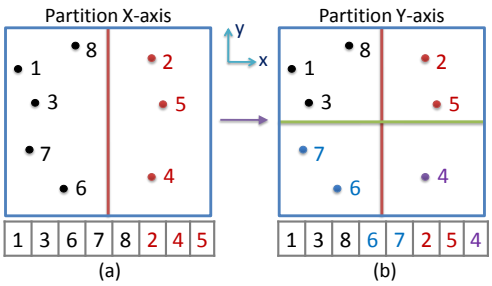


Figure 10. Spatial Clustering of Points

1. Read points in an array P of size n.
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 (can be any of x, y or z) and swap the points belonging to one side of the partition with another as shown in Fig. 10(a). We then repeat the same process and divide the 2 new partitions along the second longest axis, as in Fig. 10(b), and finally along the third. For purpose of illustration, we have shown partitioning a quadtree instead of an octree.
   (c) STOP looping when every thread encounters a leaf and hence no new partitions are generated.

Here are some of the implementation details.

MEMORY ALLOCATION: Every iteration of the algorithm creates many different number of new partitions and octree nodes. We need to allocate memory to store this new information. The problem arises here because GPU doesn't allow for dynamic memory allocation. A way to get around this is to allocate maximum possible memory. But this eventually leads to storing the whole tree ($8^0 + 8^1 + 8^2 + ... + 8^l$) till level l, and there by wasting lot of memory [3]. 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

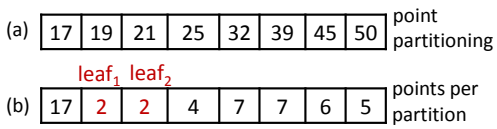


Figure 11. Partition Array of a Node

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. 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 $G92$ GPUs 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).

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 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. 11.

We have a 8 threads operating on $Node_n$ containing 50 points. Let $Node_1$, $Node_2$, ...$Node_8$ be the children of $Node_n$. As in Fig. 11, 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 $thread_0$ will write its 8 partitions at locations $1-8$, $thread_1$ at $9-16$ and so on. But since $Node_2$ and $Node_3$ are leaves, $thread_3$ will now write the new partitions at locations where $thread_1$ 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 [6] 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.

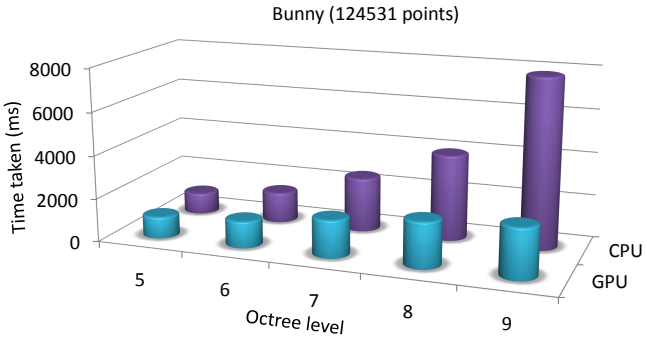
PARENT-CHILD: This relation is established while partitioning as each child partition is generated from its parent, thereby giving us our final octree.

DISCUSSION Maximum memory required for implementation is just equal to storing non-empty octree nodes, very less compared to [3]. However, it looses w.r.t time when compared to [3] but is very fast compared to the CPU implementation. As it performs data-dependent clustering, it generates a different octree compared to our first implementation. Thus they have different application areas(§ 1) and hence we dont compare them against each

other. Parent-Child, containment, range, and neighbor-finding are some example queries which it can answer.

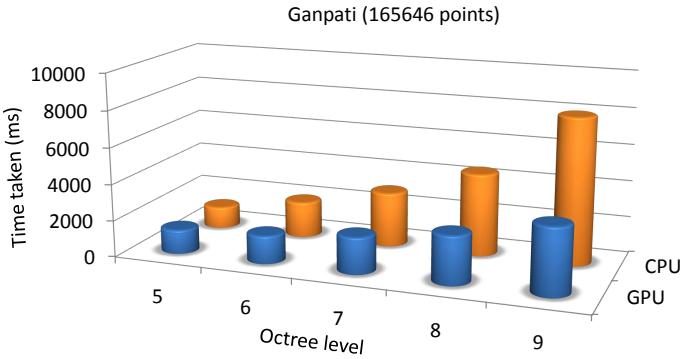
6. Results and GPU optimizations

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.



Tree level	GPU (ms)	CPU (ms)
5	1001	993
6	1231	1421
7	1742	2521
8	2117	3981
9	2323	7851

Figure 12. Top-Down Octree Construction (Bunny 124531 points) (sec. ??)



Tree level	GPU (ms)	CPU (ms)
5	1321	1200
6	1536	1981
7	2009	2997
8	2654	4521
9	3658	8001

Figure 13. Top-Down Octree Construction (Ganpati 165646 points) (sec. ??)

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.

As a proof of applicability, we used the parallel constructed octrees as a part of GPU-based global illumination algorithm for point models using the Fast Multipole Method. Fig 14 shows some results. Effects of color bleeding and soft shadows are clearly visible. 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. 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.

6.1 GPU Optimizations

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

1. LOOP UNROLLING: Any flow control instruction (if, switch, do, for, while) can significantly impact the effective instruction throughput by causing threads to diverge. Thus, significant performance improvements can be achieved by unrolling the control flow loop. 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: Obtained via an empirical study, eEach thread block must contains $128 - 256$ threads and every thread block grid no less than 64 blocks for optimal performance on $G80$ GPU. We made sure this was achieved. If the number of nodes considered 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: As every thread works only on two adjacent nodes most of the times in $\mathbf{I}_1$ or on independent partitions in $\mathbf{I}_2$, work of each thread is completely independent. This fits our situation where each thread (in any of the two implementations) on finishing its work or on making an early exit (say by encountering a leaf) simply moves on to next pair of adjacent nodes for $\mathbf{I}_1$ or a new partition for $\mathbf{I}_2$, without the need for any shared memory or synchronization with other threads. Note that to realize the full GPU load the number of nodes to be considered should be sufficiently large. With 16 multi-processors, we need atleast 64 thread-blocks, each having 128 threads to realize optimal GPU load. Thus, we realize a full GPU load for $\mathbf{I}_1$ at *even* a small enough point model (of size 8192, and assuming a point per leaf). On the other hand, for $\mathbf{I}_2$, if

Figure 14. 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.

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 ($\geqslant 6$). As we will see in Fig. **??** that GPU totally out-performs the CPU for depths $\geqslant 6$.

7 Conclusion

Rapid developments and improvements in the performance of graphics hardware has attracted much attention. Its performance is way ahead of CPUs and thus, it is being used not only for traditional graphics rendering, but for solving computationally intensive problems in varied fields. One such problem is parallel construction of octrees. We presented two different octree construction algorithms, each having its own merits and support for desired queries. When a user needs support for various different queries, the bottom-up octree construction algorithm can be used, while the top-down algorithm, although supports only a subset of those queries, has less memory requirement and is faster in terms of run-time. We compared our algorithms with their CPU equivalent counterparts and showed that they out-perform them in every department. Further, we applied both our octree algorithms to a N-body problem of computing radiosity based Global Illumination solution for point models using Fast Multipole Method and demonstrated the visually pleasing results. These algorithms can be applied to a vast variety of computationally intensive domains requiring hierarchical data structuring techniques.

References

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

[2] Nvidia CUDA Programming Guide. `http://developer.nvidia.com/cuda`.

[3] P. Ajmera, R. Goradia, S. Chandran, and S. Aluru. Fast, parallel, gpu-based construction of space filling curves and octrees. In *SI3D '08*, pages 1–1, NY, USA, 2008. ACM.

[4] S. Aluru and F. Sevilgen. Dynamic compressed hyperoctrees with applications to n-body problems. *Proceedings of Foundations of Software Technology and Theoritical Computer Science*, pages 21–33, 1999.

[5] J. Bittner. *Hierarchical Techniques for Visibility Computations*. PhD thesis, Czech Technical University, 2002.

[6] CUDPP. CUDA Data Parallel Primitives Library. `http://www.gpgpu.org/developer/cudpp`.

[7] F. Durand, G. Drettakis, and C. Puech. The visibility skeleton: a powerful and efficient multi-purpose global visibility tool. *Computer Graphics*, 31:89–100, 1997.

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

[9] S. Lefebvre, S. Hornus, and F. Neyret. *GPU Gems 2*, chapter Octree Textures on the GPU, pages 595–614. Addison Wesley, 2005.

[10] 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.

[11] H. Sagan. *Space Filling Curves*. Springer, New York, 1994.