Quadtree Construction on the GPU: A Hybrid CPU-GPU Approach

Maria Kelly
mkelly1@sccs.swarthmore.edu

Alexander Breslow
abreslo1@swarthmore.edu

Abstract

We introduce a method for fast quadtree construction on the Graphics Processing Unit (GPU) using a level-by-level approach to quadtree construction. The algorithm is designed to build each subsequent level from the parent nodes of the previous level and thus is suitable for parallelization. Our work is motivated by the use of quadtrees for spactial segmentation of LIDAR data points for grid digital elevation models (DEM) and the need to reduce total computation time in grid DEM construction. We introduce an algorithm suitable for quadtree construction on the GPU which reduces the construction problem to a sorting task of the data points. We then describe three possible implementations of the algorithm: a purely GPU-based implementation, a CPU-based sequential implementation, and a hybrid approach, which builds the first several levels on the CPU before transferring the data to the GPU to construct the remaining levels. We find that the hybrid implementation outperforms all other implementations on sufficiently large datasets and provides significant speedup over our two sequential quadtree implementations.

1 Introduction

Fast construction of spatial data structures that index many millions of points and/or polygons is essential for computation on very large datasets. Algorithms which make use of partitioned datasets often perform orders of magnitude better than their counterparts that do not segment the data before processing. Thus both fast and effective construction of spatial data structures proves a necessity for expanding the threshold of data that may be processed within a given time frame. Consequently, novel approaches in data structure generation are necessary.

To meet this end, parallelization must be thought of as the answer. Purely sequential approaches to computation, while invaluable for many applications, are reaching their limits. The growth of processor clock speed has stagnated, meaning that computation using a single core will not significantly improve, at least not in the foreseeable future. This slow-down is largely due to limits in the material properties of silicon, meaning that further increases in clock speed lead to massive increases in power consumption and, consequently, temperature. Hence without elaborate cooling systems and power grids, silicon-based processors either melt or do not have sufficient power to attain ever-increasing speeds. Therefore until a new material is used for the fabrication of computer components, realistic processor speeds are bound at approximately 2 to 3 gigahertz. What this means for computing is that parallelism must be thought of as the future of computationally intensive computing.

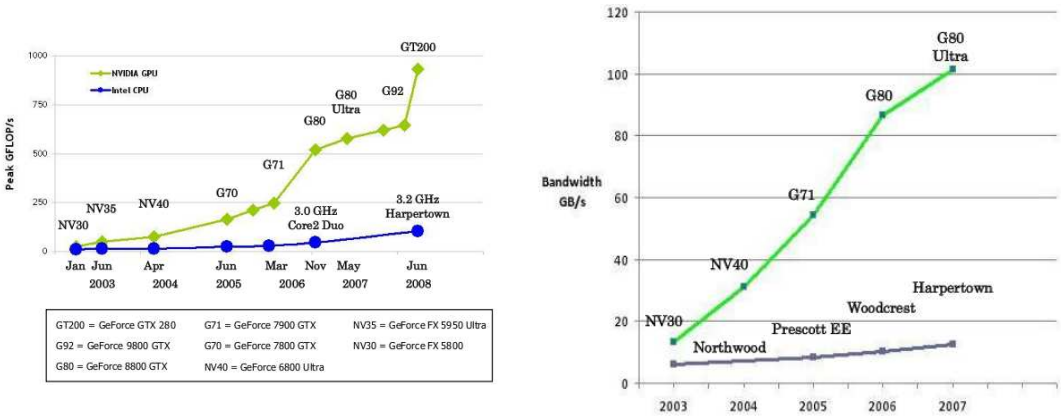

Figure 1: (Left) The maximum number of floating point operations per second (FLOPS) possible with CPU vs. GPU computing. The FLOPS achieved by NVIDIA GPU's have increased more quickly than that of the fastest CPUs. (Right) From 2003 to 2007, the bandwidth of the GPU has increased at a much faster rate than the chip bandwidth on the CPU.

Because of this, we choose to focus on an implementation for building quadtrees that makes extensive use of parallelism in order to gain significant speedup. We utilize NVIDIA's CUDA programming model for the GPU, allowing us to execute thousands of threads simultaneously on the graphics processing unit (GPU). CUDA makes use of a grid-block based architecture whereby the CPU oversees the scheduling of blocks of threads to the array of cores comprising the GPU. This allows for embarrassingly parallel applications, namely those where computation is repetitive and largely logically independent, to see speedups of up to several orders of magnitude on a consumer-level computer. Additionally, we select CUDA over traditional CPU-based thread libraries such as POSIX Pthreads or OpenMP because although multicore CPUs are now commonplace, their computational throughput is limited when compared to that of GPUs.

In a case study presented by NVIDIA in [1], NVIDIA demonstrates that, in the past several years, both floating point operations per second (FLOPS) and chip bandwidth are both better and continue to increase at much faster rate on the GPU versus the CPU (See Figure 1). Therefore, our choice of using the GPU for parallelism enables us to construct large portions of the quadtree simultaneously and take advantage of the GPU's computational superiority. This allows us to achieve considerable speedup of quadtree construction.

The next section describes related work, including initial work done with quadtree construction, the importance of quadtrees for grid digital elevation model construction, and research on constructing data structures on the GPU. In Section 3, we present our algorithm for quadtree construction and explain how we reduce the problem to a sorting task. In Section 4 we describe the three possible implementations of the algorithm and in Section 5 we present and discuss the results of implementing the algorithm on the CPU and with our hybrid CPU-GPU approach. In Sections 6 and 7 we describe future directions for the quadtree-sort algorithm approach to quadtree construction and conclude. Finally in Section 8, we present some reflections on our work, the project, and working with CUDA to program for the GPU.

2 Related Work

2.1 Quadtrees

A quadtree is a data structure used to spatially index points in R^2. The tree, like all quadtrees has a root, but specific to quadtrees is that each subsequent level can have a maximum of four children for every parent. Thus a quadtree with k levels including the root would have $4^{(k-1)}$ nodes on the kth level and $(4^k) - 1$ nodes in total. The purpose of each node is to represent a bounding box in the plane for a subset of the data points. At the root, the bounding box contains all of the points of the dataset. For quadtrees one also defines a threshold value for the maximum number of points allowed in each bounding box (segment). If the number of points in a given segment of the data is greater than the threshold value, the rectangular parent segment is split into four geometrically similar child segments.

For the purposes of geographical data, these are the Northwest, Northeast, Southeast, and Southwest quadrants of the parent's bounding box. When the number of data points in any of these segments exceeds the threshold value, then all segments exceeding the threshold are again split into four quadrants. If a segment has a number of data points less than or equal to the threshold, that segment no longer splits and becomes a leaf of the quadtree. In this way, datasets with relatively uniform data distributions may be indexed in a tree of height no greater than $O(\log_4 (datasetsize/threshold))$. Another benefit is that all segments may directly know the data points that they contain, yielding $O(n)$ access time for all the data in any segment. Consequently we feel that this is an effective way for the segmentation of data in the plane. (See Figure 2).

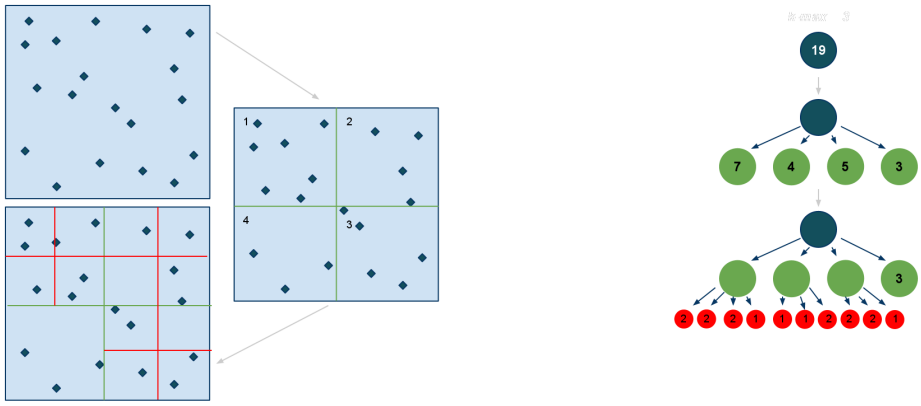

Figure 2: A sample quadtree with a threshold of $kmax = 3$. The 19 points in R^2 in the top-left bounding box are split into four children nodes having 7, 4, 5, and 3 points, respectively. Each non-terminal child is split into four child nodes until each node contains 3 or fewer points.

2.2 Grid DEM Construction Using Quadtrees

Our work is motivated by the use of quadtrees for the segmentation of LIDAR data for a given geographic region. LIDAR data is a 4-tuple consisting of an $x-$coordinate (East-West), a $y-$coordinate (North-South), a $z-$coordinate (elevation), and an intensity coordinate (strength of the returning signal). For the purposes of our experiment, we ignore the intensity. Our quadtree implementations then take the approach described in [2] in which we index data points in R3 based on their $x-$ and $y-$coordinates. This allows us to index points in 3-space without resorting to using an octree. Once the data is segmented, an interpolation algorithm may then be used on the data to interpolate

the pixel values for a digital elevation models. Because data segmentation is an important part of DEM construction, its parallelization for speedup is necessary for fast processing of multi-gigabyte datasets. This serves as the primary motivation for parallelizing the construction of our quadtree implementation.

In their paper *From Point Cloud to Grid DEM: A Scalable Approach* Argarwal, Arge and Danner propose a three part, scalable, sequential algorithm for efficient construction of grid based digital elevation models (grid DEMs) on large, multi-gigabyte datasets [2]. They describe a three phase approach consisting of data segmentation using a quadtree, neighbor finding also using said quadtree and thirdly interpolation applying whatever method one sees fit. The authors explain how large quadtrees that do not fit entirely into RAM must instead be built and processed in stages. Agarwal et al. describe incremental and level-by-level construction of quadtrees and then adopt a hybrid approach. In said approach, the authors build three levels of the quadtree at a time, with leaves containing lists of points. The algorithm is then repeated on each of these lists until the number of points in a leaf is less than or equal to some threshold value. Points that are too close to other points are discarded.

To find the neighbors of a leaf, the authors again describe an incremental approach where leaves that share a partial common edge or full edge of their bounding boxes are said to be adjacent. This is accomplished by first selecting a leaf and then tier by tier seeing if its bounding box shares partial or full edges with the the bounding boxes of other segments. If a segment is not a leaf, this process is repeated on its children until only leaf nodes are said to be adjacent to the leaf node in question. The authors' layered approach involves loading the top levels of the tree into memory and recursively processing the leaves using the incremental approach and some additional overhead.

The description of the data interpolation phase leaves something to be desired, but Agarwal et al. admit that there are multitude of data interpolation algorithms. Thus the algorithm is not data interpolation method specific.

For the purposes of our work, we choose only to focus on the segmentation phase and its parallelization. We observe that this is an essential part of the algorithm to parallelize because parallel-constructed quadtrees are not only useful to the indexing and segmentation of LIDAR data but have been shown to serve many other purposes. In fact a more generalized version of our implementation should extend to quadtrees used for image processing and compression as well as for computer graphics.

2.3 Data Structures on the GPU

Another paper which is quite relevant to our work is *Fast BVH Construction on GPUs* by C. Lauterbach et al. 2009. In their paper, they present a new algorithm for the computation of bounding volume hierarchies (BVH). BVHs are of imminent importance throughout many areas of computational geometry and computer graphics because algorithms that make use of them are often times considerably faster.

Because spatial segmentation through the use of quadtrees is a simplified two dimensional analog of bounding volume hierachies, understanding fast construction of BVHs is key to making good choices when designing an algorithm for fast construction of quadtrees. Consequently this paper demonstrates several important lessons which are relevant to our work. First, they choose to implement their algorithm in CUDA because it allows for fine-grain parallelism and simultaneous execution of thousands of threads. In their implementation, they make heavy use of shared block memory which is as fast as register accesses for single readers. This is approximately 150 to 200 times faster than global device memory, making its use a must for significant speedup.

Second, they implement a novel algorithm called the Linear Bounding Volume Hierarchy (LBVH) which reduces the problem of bounding volume hierarchies to one of sorting. This allows them to achieve speedups of BVH construction which are over an order of magnitude faster than equivalent GPU-based constructions using the Surface Area Heuristic (SAH). The authors map the geometric primitives to a space filling curve. They then sort using the primitives' relative position along the curve as the heuristic. If a parent occupies the interval $[0, max]$ on the curve, then its k children would occupy $[0, c1), [c1, c2), \ldots, [ck - 1, max]$. In this way, all primitives may be sorted without destroying ancestor's descriptions of their data members, as sorting on subintervals does not affect the relative ordering of the data on the larger curve.

These facts are of importance for our work because we aim to achieve parallel speedup by implementing an algorithm which reduces the problem of dataset segmentation into a sorting problem. Likewise, we hope this will permit our parallel implementation of quadtree construction to be more than an order of magnitude faster than both the iterative and recursive sequential, CPU-based approaches because sorting algorithms can make use of the fast, shared memory of thread-blocks.

2.4 Initial Work on Quadtrees

In their paper, *Quad Trees: A Data Structure for Retrieval on Composite Keys*, Finkel and Bentley introduce the quadtree data structure [3]. They give a brief definition and provide algorithms for insertion and balanced insertion, searching and optimization. The authors claim that the deletion of nodes and the merging of quadtrees proves to be quite difficult. However, insertion is $O(n \log n)$, searching is $O(\log n)$ and optimization for the purposes of balancing the tree is O(nlogn). These properties are quite nice because they allow quadtrees to be quickly constructed and restructured. This makes them an ideal data structure for multiple insertions.

Since the paper focuses on quadtrees using composite keys, there are additional problems which are found that do not exist in the more naive implementations for data indexation, segmentation and compression. For instance, removal of nodes is not an issue for our application because a remove of a node would mean a loss of information about the segmentation of the dataset. Additionally, merging two trees is not an issue for the data segmentation phase of our algorithm because one constructs only one quadtree. So although merging information on two datasets is difficult, this in not the object in question. Therefore the quadtree is very well suited for this application, as its primary weaknesses do not play a role in the implementation of our algorithm.

2.5 Linear Quadtrees

In his 1991 work, Fangju Wang gives a good overview of linear quadtrees and introduces the relational-linear quadtree [5]. Wang describes the linear quadtree as sorted list of region codes, essentially a sorted list of keys that map to the leaves of the tree. Leaves are encoded in binary based on the quadrants where they and their ancestors exist through the recursive splitting of the quadtree construction process. This canonical labeling takes the form of 0 for NW, 1 for NE, 2 for SE and 3 for SW. Thus a node which represents a region in an image in the upper right of the rectangular pixel buffer could have a region code of 12. Namely, this node has as its ancestor the root and the root's NW child. Since the code is 12, it occupies the SE quadrant of the root's NW child's bounding box. If a region represented by a node is the merged result of smaller regions, an X is appended to its region code for every time it has been merged. For example, 30X would be the same as 30, but because the height of the tree excluding the root is 3, we append an X.

The author also gives many of the motivations for using quadtrees specifically citing the linear quadtree's application to geographic information systems, to which our paper is oriented. Because

the linear quadtree directly stores the leaves in an array, it allows for more efficient use of memory and quicker access to data. One does not have to do a tree traversal to raster an image represented as a linear quadtree but merely iterate through the leaf list. From here, he presents the relational-linear quadtree.

3 A Sorting Algorithm Solution

To accomplish the task of parallelizing quadtree construction for the GPU, we follow the general idea from [4] that the simplest method for building a quadtree on the GPU is to reduce the problem to "a sorting problem," which closely resembles a linear quadtree. Thus, at each step of our construction algorithm, we implement a *quadtree-sort* on the dataset and let each new child node correspond to a specific interval of the dataset for which we are building the quadtree. Using this method, nodes are represented by their specific start end end points in a quadtree-sorted dataset rather than by explicitly linking them them with the specific data that they contain.

3.1 Data Structure Representation

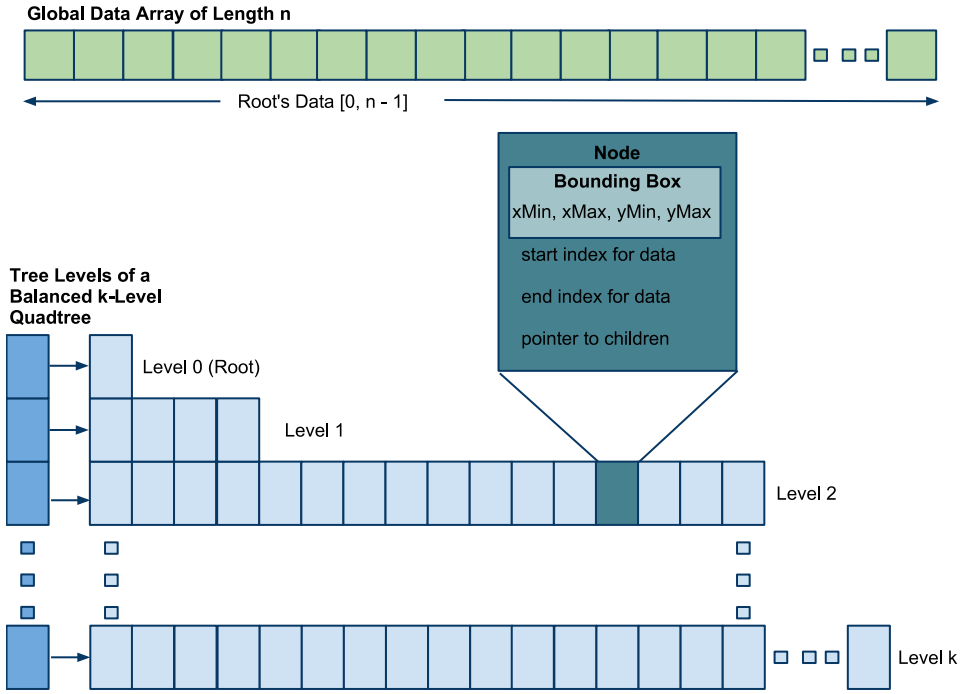

Figure 3: The quadtree data structure for the quadtree-sort method of quadtree construction is comprised of the original data-array, an array of arrays of nodes for each level. Each node consists of a bounding box, start and end indices representing their interval of the global data array and a pointer to its four children.

For the *quadtree-sort* linearized approach to quadtree construction, the actual quadtree data structure is comprised of (1) the array of datapoints and (2) an array containing one array of nodes for each level of the tree. The nodes are comprised of a two-dimensional bounding box for the data they represent, start and end indices into the array of datapoints, which represent the portion of the data for which that node is responsible, and a pointer to four children nodes. See Figure 3 for a visual representation of the data structure.

$xRange \leftarrow (xMax - xMin)$
$yRange \leftarrow (yMax - yMin)$
for all p in $[s_{parent}, e_{parent}]$ **do**
  $xIndex \leftarrow 2 * (p.x - xMin)/xRange$
  $yIndex \leftarrow 2 * (p.x - xMin)/xRange$
  $bucketIndex \leftarrow 2 * yIndex + xIndex$
end for

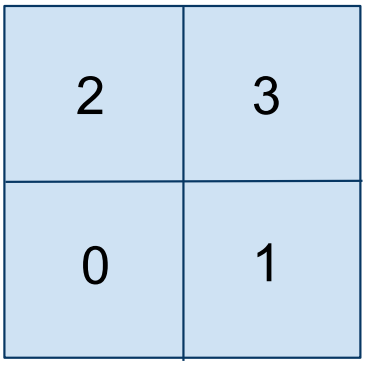

Figure 4: (Left) The algorithm for assigning child buckets to points p from the parent node, where, $xMin$, $xMax$, $yMin$ and $yMax$ are the outer vertices of the parent node's bounding box. Each point is assigned an $xIndex$ and $yIndex$ with values of either 0 or 1 which are used to compute the bucket index value between 0 and 3. (Right) A visualization of how each bucket index corresponds to the four child nodes' bounding boxes.

3.2 Construction of the Quadtree

To construct the quadtree, we must first traverse the entire data set to find the maximum and minimum values for the $x-$, $y-$coordinates of the dataset and thus determine a bounding box for the data. Then, as we build nodes for each sublevel of the tree, we initialize a new array of child nodes for the sublevel which consists of four times as many nodes as were present in the previous level. Then, for each parent node in the array of the previous level, we subdivide the node's bounding box into four equally-sized bounding boxes. After we compute the child nodes' bounding boxes, we implement *quadtree-sort*.

This sort is a modified bucket sort which consists of two stages: first, each datapoint from the parent node (the points in the data-array from s_{parent} to e_{parent}) is assigned one of four buckets corresponding to which child node's bounding box the point belongs (see Figure 4); then the points are sorted within the data-array such that all points corresponding to a given bucket are placed next to each other in the sorted data-array and that all points remain in the interval (s_{parent}, e_{parent}). Finally, we assign to each child node its start and end indices in the newly-sorted data-array.

After sorting, we obtain a new configuration of the dataset in which the children of each parent node will correspond to subintervals $[s_{parent}, e_{c1}]$, $[e_{c1}+1, e_{c2}]$, $[e_{c2}+1, e_{c3}]$, and $[e_{c3}+1, e_{parent}]$ where e_{ci} is the endpoint of the data belonging to bucket i in the sorted dataset. We continue subdividing the bounding boxes at each level and conducting a quadtree-sort of the corresponding data until the number of points in every node in the sublevel is less than or equal to the threshold value, $k - max$ (See Figure 5). If, at any level, the parent node consists of $k - max$ or fewer datapoints, its child nodes are assigned null values.

4 Three Implementation Choices

The algorithm described in the previous section for a building a quadtree level-by-level can be easily parallelized to the GPU even for large datasets by assigning one thread to every parent node. However, it is also possible to implement the algorithm in a purely sequential manner on the CPU by initializing a child node array, looping through the parent nodes of each level and building its corresponding child nodes (bounding box assignment and quadtree-sort of the parent's datapoints). In this section we describe in detail the design choices associated with these two implementations and introduce a hybrid approach to quadtree construction that utilizes both the CPU and the GPU.

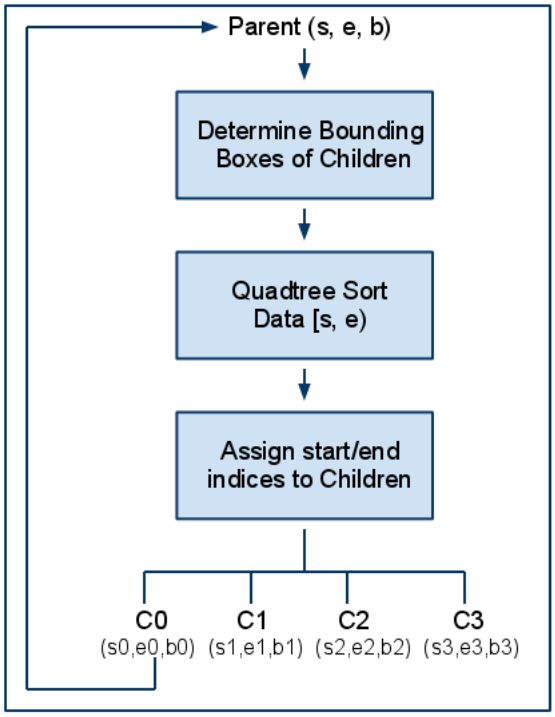

Figure 5: The quadtree-sort method of quadtree construction is comprised of three phases: bounding box construction, quadtree-sort, and assignment of start and endpoints of the sorted dataset.

4.1 Parallelization on the GPU

The quadtree-sort method for level-by-level construction of the data structure is logically suitable for parallelization. Since each parent node is distinct and does not share any data with other nodes from the same sublevel, the four children of each parent node can be computed simultaneously. Thus, in the GPU-based approach to quadtree-sort construction, we spawn one thread for each parent node from the previous level which computes and assigns the child bounding boxes and carries out quadtree-sort on the appropriate datapoints. Parallelization of quadtree construction is key for large datasets: since the number of children at each subsequent level is four times that of the previous level, the number of nodes at each level increases rapidly for large datasets and thus it is useful to be able to employ one thread for each parent node.

However, given that, in this implementation, parallelization takes place at the node level of granularity, parallelization only significantly aids performance over sequential construction once the number of parent nodes for a level increases above some threshold. In fact, because of the significant amount of time it takes to implement bucket sort on large dataset, the construction of the first several levels of the tree dominate the total construction time. Therefore, we investigate the CPU-performance of the algorithm in addition to the purely GPU-based approach.

4.2 Sequential Approach on the CPU

To implement the level-by-level quadtree-sort construction algorithm on the CPU in a sequential manner, instead of creating one thread for each parent node, we loop through the node array of parents and for each parent construct their four children by determining bounding boxes and doing a quadtree-sort on the corresponding datapoints. By constructing the quadtree on the CPU, we

are able to utilize functionality that the GPU and CUDA programming model do not allow for, most importantly, the use of non-inline functions. Therefore, in the CPU-based approach to the algorithm, we choose to use the C-standard quicksort **qsort()** to accomplish the quadtree-sort of the data. We expect that as the size of each subsequent level increases, computation entirely on the CPU in this way will become slow. However, the construction of the first few levels should be relatively quick as we are able to make use of a fast sorting algorithm.

4.3 Achieving Optimal Performance via a Hybrid Approach

In the previous sections, we described the benefits and drawbacks the entirely GPU-based and entirely CPU-based approaches. Now, we describe a hybrid CPU-GPU method that optimizes performance of the algorithm by taking advantages of the strengths of each approach. This hybrid method builds the first few levels of the quadtree on the CPU by using the sequential method described in Section 4.2. Then, at a user-defined *switch-level*, we transfer the array of parent nodes to the GPU and begin constructing the remaining levels of the tree on the GPU until every child in the current sublevel is either null or a leaf (a node containing $k - max$ or fewer datapoints).

4.4 Memory Usage and Allocation

The memory model on the GPU is entirely independent of that of the CPU and thus certain decisions about memory allocation and transfer of data from and to the GPU must be made for both the entirely GPU-based and hybrid implementations. We choose to transfer data to the GPU only once: when we first begin GPU construction. Thus, in the entirely GPU-based approach, we first copy the root and the global data-array to the GPU and in the hybrid approach, we copy the array of nodes from the previous level and the global data-array to the GPU when we reach the switch-level. The array of level arrays is updated after each level on the GPU is constructed by assigning to it the pointer to that level on the GPU. Once the entire tree is constructed, these pointers are used to copy the levels onto the CPU.

For the GPU implementations, because each level of the array remains on the GPU, we are unable to know exactly how many nodes in each level are leafs of the tree or null nodes. Thus, in building the next level, we typically overestimate the number of nodes in the level by allocating space for 4^c nodes, where c is the number of the level we're on (for example, level 0 is comprised of one node, the root). While this space-time tradeoff for memory usage works well for small datasets, for larger datasets the GPU does not have enough memory to contain these oversized level arrays.

In this situation, when the GPU does not have enough memory to allocate the next level with space for 4^c nodes, all of the level arrays stored on the GPU are copied onto the CPU and freed from the GPU memory. We then find the non-terminal nodes from the array corresponding to the parents of the level we seek to build, copy these non-terminal nodes back to the GPU, allocate space for four times as many child nodes, and construct the level on the GPU. For example, if we found that in the parent node array there were four non-terminal nodes, we copy only this subset of the array to the GPU and allocate a child array for sixteen nodes on the GPU.

After this data-transfer off of the GPU has taken place, the algorithm proceeds as before, allocated four times as many children for each node in the parent array (in the example, the next child array would consist of sixty-four children), and would execute another data-transfer step if the GPU were to once again run out of memory.

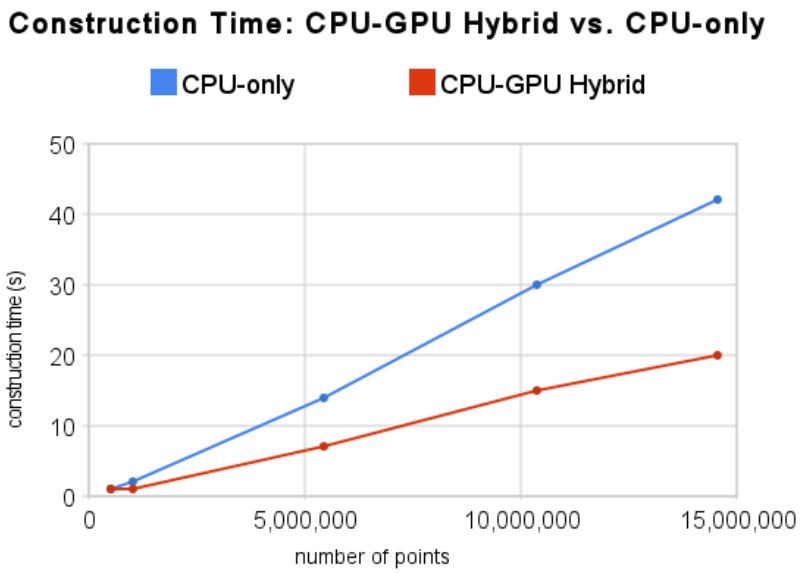

Figure 6: **Quadtree construction times in hybrid and CPU-Only implementations.** For larger datasets, the speedup of the CPU-GPU hybrid implementation is constant: the hybrid implementation is about twice as fast on average as the CPU-only implementation.

5 Experimental Results

We implement the CPU-only approach to the algorithm in C using qsort() for the quadtree sort phase of the algorithm. For parallelization on the GPU, we use NVIDIA's CUDA with C-bindings and write two independent kernels: the first for bounding box assignment and the second for quadtree sort and start/end index assignment. To evaluate the performance of the hybrid CPU-GPU model, we ran several experiments comparing it against the purely CPU-based version of the algorithm.

We ran timed-trials using datasets of LIDAR points ranging in size from several hundred thousands of points to over fourteen-million points. Each timed experiment was run for ten trials on each dataset. The significance of our results are threefold: improved performance of the hybrid model over the entirely CPU-based version, importance of choosing the optimal switch-level for GPU construction, and the significance of the difference in the amount of time for system calls between the two versions.

5.1 CPU-GPU Hybrid vs. CPU-Based Implementation

We conducted several experiments of different sized datasets ranging from just over 500,000 points to over 14 million points to determine the speedup achieved by implementing the algorithm using the hybrid approach. We timed and compared the quadtree construction portion of computation on each dataset for the hybrid and CPU-based implementations. The results of these experiments are shown in Figure 6. Our results show that the hybrid implementation of our quadtree construction algorithm exhibit speedup of approximately 2.0 over the sequential, CPU-only implementation.

5.2 Determining the Optimal Switch Level for Hybrid Construction

Throughout our experiments, we discovered that determining and using the optimal switch-level in the hybrid implementation is the key to achieving the best possible performance of the al-

gorithm. Thus, we sought to determine what switch-level or range of switch-levels are optimal for the types of problems we are dealing with. We conducted a series of experiments which time the quadtree construction phase of the algorithm for various switch-levels. The amount of time required for quadtree construction for various sized datasets using different switch-levels. We ran ten trials of each switch-level for datasets which ranged in size from about 500,000 points to over 29-million points. The results, which are depicted in Figure 7 indicate that the optimal switch-level is between level two and four. With earlier switch-levels being better for larger datasets.

Optimal Switch Levels		
Number of Points (millions)	**Switch Level**	**Construction Time (sec)**
1.0	1	1.99872
	2	1.53042
	3	**1.00388**
	4	1.99625
5.4	1	12.9965
	2	**9.00711**
	3	10.0037
	4	11.9986
10.4	1	22.0032
	2	**16.9985**
	3	18.0025
	4	21.0023

Figure 7: **Determining the optimal *switch-level***. Optimal switch-level times are shown in bold. The results of our experiments indicate that the optimal switch-level is always in the first three levels.

5.3 Differences in System Time

Finally, in conducting these timed tests on different datasets, we computed how much time was spent overall in system and user times. We find that the time involving system calls is higher in the hybrid implementation than in the entirely CPU-based algorithm: 10% of the total execution time (from reading in the data file to complete construction of the quadtree) was from system calls in the hybrid implementation versus an average of 2% of time for system calls in the CPU-only implementation.

The difference in system time required in the two implementations can be attributed to the amount of time required to transfer data between the CPU and GPU. We observe that when implemented at the optimal switch-level, the time for system calls consistently comprised 10% of the total execution time. From this result, we may infer that what makes a particular switch-level optimal is that system calls required for transfer to and from the GPU for the given switch-level remains at the ideal proportion of the total computation.

Further, this result suggests that any increase in the amount of transfer between the CPU and the GPU to reduce unused memory allocations for child nodes on the GPU (as described in Section 4.4) would increase the total time of the algorithm by using an increasing number of system calls. However, future research must be conducted to investigate whether higher amounts of data transfer would result increased computation time.

6 Future Work

There are many possible directions for future work on the quadtree-sort algorithm for quadtree construction on the GPU using a hybrid approach. One important future direction will seek to parallelize computation on the CPU. Currently, computation on the CPU, specifically looping over each parent node in a given level, is done in an entirely sequential manner. This results in the CPU-computation dominating most of the time for quadtree construction. Instead of relying on a purely sequential CPU implementation, in the future we intend to utilize the multi-core CPUs available on many modern computers to parallelize this computation. Furthermore, the process for sorting the data on the CPU could also be parallelized, which would also result in lessened computation time on the CPU.

One design choice made for this particular project was to reduce the amount of data transferred between the CPU and the GPU at any given level. This came at the expense of a significant amount of wasted space on the GPU. One possible next step for further optimizing the hybrid implementation will involve determining the costs and benefits of transferring data off of the GPU in order to determine more accurately the amount of space necessary to allocate for child nodes in the next level of the tree.

Finally, one other significant task that remains is to modify the algorithm for larger datasets whose points cannot fit into memory and thus necessitate reading points from disk in pieces and reading and writing the levels of the quadtree to disk as computation occurs. For these datasets, we seek to implement methods for transferring levels off of the GPU and CPU to files on disk which can then be read back in pieces in order to construct subsequent levels of the quadtree.

7 Conclusion

Our work seeks to extend and improve the performance of the algorithm for digital elevation modeling proposed in [2] by parallelizing the segmentation phase of the algorithm–quadtree construction. We introduce a new method for constructing quadtrees on the GPU in parallel which relies on a level-by-level approach with parallelism implemented such that the children of each parent node from the current level are computed simultaneously. The algorithm reduces the problem of quadtree construction to a sorting problem in which nodes do not explicitly contain the data they correspond to but to their start and end indices of their portion of the data in a global quadtree-sorted list of all of the datapoints.

We propose a hybrid CPU-GPU approach to the problem of quadtree parallelization which involves building the first several levels of the tree on the CPU sequentially and then transferring the data to the GPU to compute the remaining levels of the tree. We demonstrate that this model is effective in reducing the amount of time necessary for quadtree construction and that it is in fact twice as fast as implementing the algorithm entirely on the CPU.

8 Reflections on Our Work

The original goal of our project was to parallelize not only quadtree construction but also a form of interpolation for created grid DEMs involving Voronoi diagrams. Due to the extremely steep learn-curve of the CUDA programming model, including the black-box nature of computations taking place on the GPU, we ran into several difficulties along the way.

The most significant of these hurdles was in determining the best way to represent the data structure on the GPU. The memory-model of the GPU does not allow for easy manipulation and computation on pointers to pointers. Thus, we had to flatten out our data structure and eventually came to a breakthrough in the quadtree-sort data structure representation in which nodes do not contain pointers to the data they contain, but instead this data is represented by a start and end index into the global, quadtree-sorted data-array.

References

[1] CUDA Programming Guide Version 3.0.

[2] P. K. Agarwal, L. Arge, and A. Danner. From point cloud to grid DEM: A scalable approach. In Andreas Riedl, Wolfgang Kainz, and Gregory Elmes, editors, *Progress in Spatial Data*

Handling. 12th International Symposium on Spatial Data Handling, pages 771–788. Springer-Verlag, 2006.

[3] R.A. Finkel and J.L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4(1), 1974.

[4] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. Fast BVH construction on GPUs. *EUROGRAPHICS 2009*, 28(2), 2009.

[5] Fangju Wang. Relational-linear quadtree approach for two-dimensional spatial representation and manipulation. *IEEE Transactions on Knowledge and Data Engineering*, 3(1), 1991.