Fast GPGPU Based Quadtree Construction

Joshua Gluck
Advisor: Andrew Danner

May 1, 2014

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 is thus suitable for parallelization. Our work is motivated by the use of General Purpose GPU (GPGPU) techniques for large data sets generally, and for the use of quadtrees for spatial segmentation of lidar data points for grid digital elevation models (DEM) in particular. We introduce an algorithm suitable for quadtree construction on the GPU which reduces the construction problem to bucket sort. We then describe possible implementations and refinements to the algorithm: utilization of multiple threads on a single quadtree node, null-node pruning, and a hybrid CPU-GPU approach for extending our solution past the limits of GPU memory. We find that our fully implemented algorithm outperforms a CPU approach by a factor of between $5\times -12\times$ for sufficiently large datasets.

1 Introduction

There has been an increasing trend in the field of Computer Science to working on problems which involve very large datasets. The size of data available to Computer Scientists has expanded at a significantly faster rate than CPU processor speed. Therefore, designing scalable solutions for processing these ever increasing large datasets is essential in big data research areas. One such field, Geographic Information Science (GIS), which focuses on the study and analysis of spatial data for a variety of purposes, including terrain modeling, requires processing extensive spatial data sets to generate accurate solutions. Significant research has been done to show that doing so efficiently requires the use of spatial data structures to index these data sets [2]. Examples of such data structures include quadtrees, octrees, B-trees, and k-d trees[9] [11] [14]. These data structures are necessary because algorithms which make use of the partitioned nature of the data in these data structures often perform significantly better than their counterparts that do not partition the data prior to processing.

We take as a given the importance of spatial data structures, and instead focus on the construction of these data structures. This is an important task because the creation of these data structures also represents a significant cost to processing extensive spatial datasets. While the costs involved are quickly amortized by the faster querying of the partitioned data compared to unpartitioned data, the fact remains that the construction of the initial data structure represents a major cost, particularly for the testing stages for larger projects in which the data structure may be built and rebuilt repeatedly. In light of this, determining and implementing methods for faster creation of these data structures represent important steps for ensuring either expanding the extent of data that can be processed within a given timeframe, or reducing the timeframe required to process a given size of data. We focus primarily on fast initial construction of the quadtree data structure, which is used to index points in R^2. Dynamic quadtrees, in which additional data points are added/removed after construction and spatial data structures generally, while important and well-studied, [16] are outside the scope of this work. A more in depth discussion of the mechanics of quadtrees will be provided in the related works section.

2 Parallelism

Parallelization represents a key tool for the fast creation of spatial data structures for large data sets. Purely sequential approaches to computation have been invaluable for many applications; however, as mentioned previously, processor clock speeds have remained relatively stagnant compared to the growth in the size of the datasets being processed for spatial applications. Furthermore, this relative slow-down is largely due to limitations in the material properties of silicon, meaning further increases in processor clock speed lead to massive increases in power consumption and therefore temperature. Without elaborate (and expensive) cooling systems and power grids, silicon-based processors either melt-down or cannot draw enough power to successfully run at higher speeds. Short of some new breakthrough in processor design, such as a material replacement for silicon, realistic processor speeds are bound to the 2 to 3 gigahertz range.

Instead of attempting to make individual CPUs faster, parallelism represents the future for computationally intensive computing. Parallelism involves the use of multiple processors simultaneously to solve a single computational problem. Instead of attempting to make a single processor faster, the work is evenly (or as close to evenly as possible) divided among multiple processors. Ideally, this would allow for a linear speed up of computation time in relation to the number of processors being utilized. In reality, parallelism has two major draw backs. First, parallel solutions often require additional work to be done, in the form of communication, synchronization, or other tasks, which reduce the efficiency of the computation as well as its overall speedup. Secondly, due to the need for communication and synchronization, a relatively simple computational problem can have a complex and difficult to implement parallel solution. However, in the first case, while computational efficiency might drop, parallelism still means that a given task can be completed in far shorter time.

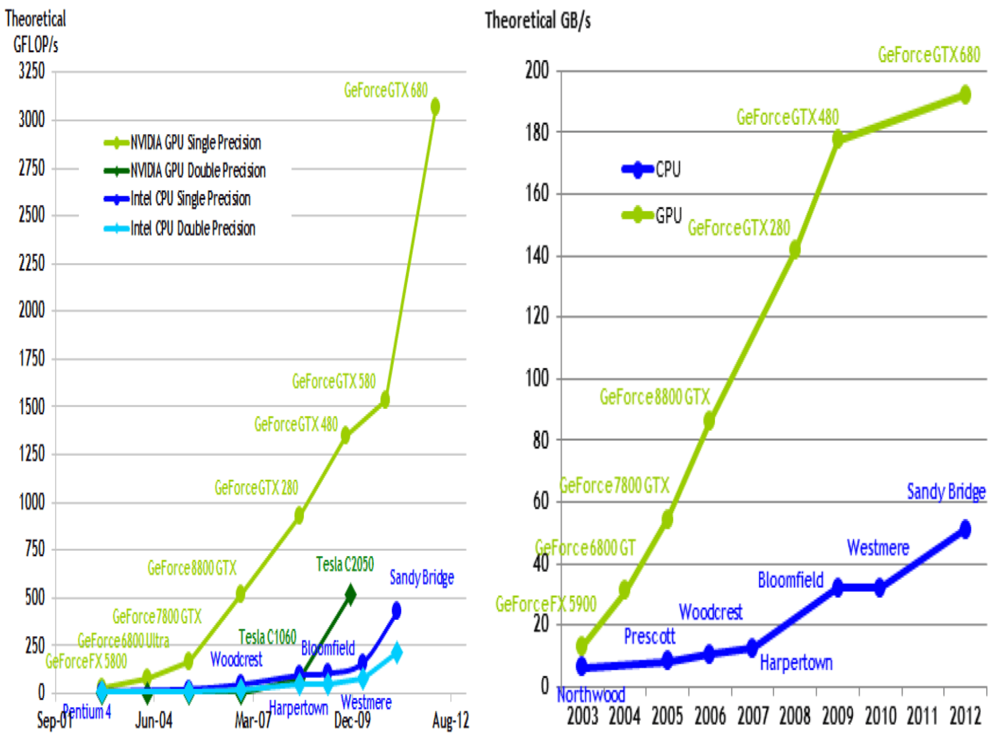

Figure 1: (Left) The maximum number of floating point and double precision operations per second possible with CPU vs. GPU computing. In both categories, NVIDIA GPU's have widened their lead on the fastest CPUs. (Right) From 2003 to 2012, the memory bandwidth of NVIDIA GPUs has increased at a faster rate than the bandwidth of the CPU [1].

The second case represents a major element of the motivation for our work. Because of this, we chose to focus on implementing a quadtree construction algorithm which makes extensive use of parallelism in order to increase performance as much as possible.

Having determined that parallelism was the solution, we then had to make the decision as to which parallelism paradigm we would use. We considered a number of options, such as using the POSIX Pthreads library to exploit the multiple CPUs available in most modern computers [7], a multihost framework using the OpenMPI [8] message passing framework, for utilizing the cores of many different computers simultaneously, and finally NVIDIA's CUDA programming model for the graphics processing unit (GPU) [1]. We settled on the last choice, for reasons that will be discussed below.

2.1 Graphics Processing Unit

In order to understand the advantages and disadvantages of a GPU framework for parallelism, it is necessary to have a basic understanding of the GPU generally and NVIDIA's CUDA programming model specifically. Fundamentally, a GPU differs from a CPU on a number of different levels. First, whereas most computers will generally have somewhere between one and 20 CPU cores, a GPU has hundreds to thousands. Second, each individual GPU core is much smaller, slower, and has a more limited instruction set available to it than a CPU core. For example, when working within the CUDA framework, memory on the GPU must be allocated by the CPU, and all data transfers between CPU and GPU memory must be initiated by the CPU. This has made GPU's ideal for the task for which they were originally developed, computer graphics, which involves a large number of vector/matrix functions and processing; however, it also makes the GPU a potentially useful tool for other tasks, including processing large, but relatively uncomplicated, spatial datasets.

Additionally, CUDA makes use of kernels, which are CPU called function descriptors which take as arguments the number of 'blocks' to be used on the GPU, and the number of 'threads' to be used within each block. Within a block, a group of threads (for our purposes 32) known as a 'warp' execute the same instruction at the same time. Blocks and threads represent two distinct levels of parallelism on the GPU, and achieving maximum performance requires the use of both threads and blocks. First, threads have access to shared local memory which acts similarly to a cache in the CPU memory hierarchy, in contrast to the larger but slower global GPU memory. Second, there are a number of synchronization primitives available in CUDA to ensure that computation occurs in a certain order among threads within a block (although these can be computationally expensive to run). CUDA makes no guarantees as to the order in which blocks run. As will be discussed in later sections of this paper, we make use of the advantages of both block and thread level parallelism. For more information on GPGPU techniques, see [12] and [13].

With at least a partial background of the mechanics of GPUs and the CUDA framework in hand, we can discuss our reasons for, and against using this framework. The enormous number of processing cores available on the GPU allows for embarrassingly parallel applications, defined as those which have computation which is repetitive and logically independent, or applications which perform a significant degree of embarrassingly parallel computation, to see speedups of up to several orders of magnitude on regular consumer-level technology. This potential for massive speed-ups on consumer-level technology was the primary motivation for choosing CUDA over traditional CPU-based thread libraries, such as POSIX Pthreads or OpenMPI, because although multicore computers are now commonplace, their potential throughput is still limited when compared to that of GPUs.

In a case study presented by NVIDIA in [1], it was demonstrated that both floating point operations per second(FLOPS) and chip bandwidth are both better and continue to increase at a much faster rate on the GPU versus the CPU(see Figure 1). In light of this, the choice of using the GPU for parallelism was made to enable the construction of large sections of the quadtree simultaneously by taking advantage of the GPU's relative computational superiority, to achieve impressive speed gains.

While the GPU certainly has a greater computational potential than sequential or multi-core CPU programming, there are some drawbacks. First, fully utilizing the GPU's processing speed is difficult for problems which involve any non-embarrassingly parallel element, which quadtree creation does, as will be explained in later sections. Second, the memory space available to the GPU is limited compared to the CPU. This problem is particularly acute given the reason, the large and increasing size of spatial data sets, for creating the quadtree in parallel. Our solution is a hybrid approach, which makes use of the CPU to preform pre-processing on data sets which are too large for the GPU, and then pass this data on in chunks of a size that the GPU can handle. This reduces the relative speed gain of our implementation for these larger data sets, but also allows us to scale to these larger data sets while maintain significant performance gains.

The next section describes related work, including a detailed description of quadtree design, the importance of quadtrees for tasks such as grid Digital Elevation Model (DEM) construction, and research on constructing data structures, including some preliminary work on quadtree construction, 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 the implementations of the algorithm and in Section 5, we present, compare, and discuss the results of algorithms described in Section 4. In Sections 6 and 7, we present our conclusions and possible future directions the creation of quadtrees on the GPU using the quadtree-sort algorithm approach outlined in Section 3.

3 Related Work

3.1 Quadtrees

A quadtree is a data structure used to index points in R^2. The quadtree has a root node, like all trees, and each subsequent level of the quadtree can have four children nodes for each parent node. Therefore, a k-level quadtree, including the root, would have at most 4^{k-1} nodes on the kth level, and $(4^k - 1)/3$ nodes in total. Each node of the quadtree represents a bounding box in the plane, which contains some subset of the data points. At the root, the bounding box contains all of the points of the dataset. Our quadtrees also have a user-defined threshold value for the maximum number of points allowed in each bounding box. If the number of points in a given bounding box is greater than the threshold value, the parent node is split into four geometrically similar child segments, each representing a disjoint bounding box covering one fourth the area of the parent node. For a more in depth description of quadtrees, see [11].

When working with geographical data, these child segments are the Northwest, Northeast, Southeast, and Southwest quadrants of the parent's bounding box. If the number of data points in any of these child nodes exceeds the threshold value, then all nodes which exceed the threshold will be split recursively. If a node contains a number of data points

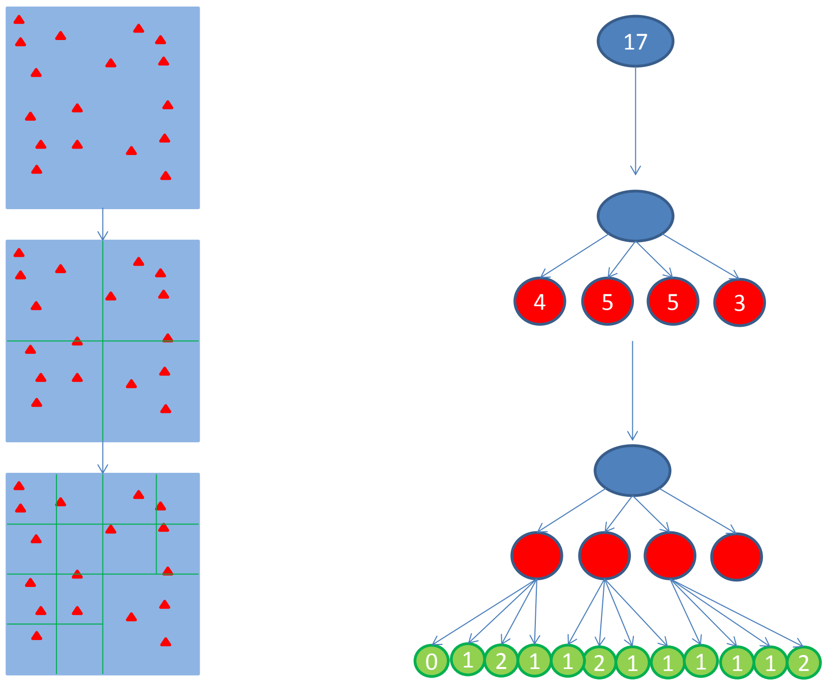

Figure 2: A sample quadtree with z = 3. The 17 points in R2 in the top most bounding box are split into four children nodes having 4, 5, 5, and 3 points, respectively. Each non-terminal child (all but the last) is then split into four child nodes, at which point no node contains 3 or more points, so construction is complete.

less than or equal to the threshold value, that node will not be split and instead becomes a leaf of the quadtree. This process continues until there are no non-leaf non-parent nodes. Because of the division of territory, datasets with relatively uniform data distributions may be indexed by a quadtree with a height of approximately $O(\log_4(\frac{n}{z}))$, where z is the threshold value, although in extremely skewed data (which is not common in practice), the height of the quadtree might approach $O(n)$. Another major benefit is that all nodes may directly know the data points that they contain, yielding $O(z + h)$, access time for a given point in any node, where h denotes the height of the tree. See Figure 2.

While extremely skewed data is not common in geospatial datasets, there are more complex data structures, such as the k-d tree, which enforce balancing conditions so that the height of the k-d tree remains low. Dealing with skewed data is outside the scope of this paper, but for more information on the subject, see [6] and [14].

3.2 Grid DEM Construction Using Quadtrees

While our work is motivated towards developing techniques for the fast and efficient creation of spatial data structures for large data sets generally, our specific focus is on the use of quadtrees for the segmentation and indexing of lidar data for a given geographic region. Lidar, which stands for Light Detection And Ranging, is a remote sensing technology which uses light in the form of a pulsed laser to measure ranges from an aircraft to the Earth. Given a known position for the aircraft, lidar can generate relatively precise three-dimensional data on the position of the earth's surface at a large number of points. Lidar data consists of a 4-tuple containing of an x-coordinate, denoting (East-West), a y-coordinate (North-South), a z-coordinate (elevation), and intensity coordinate (strength of the signal). For the purposes of our experiments, we ignored the intensity coordinate. For more information of lidar, see [10]. Our quadtree implementations index the data points in R^3 based on their x and y coordinates only. After the data is segmented, the data can be interpolated to generate values for specific pixels for a DEM. Since data segmentation is an important part of DEM construction, its parallelization is necessary for fast processing of large data sets.

In their paper "From Point Cloud to Grid DEM: A Scalable Approach", Agarwal, Arge, and Danner propose a three part, sequential, scalable algorithm for fast construction of grid based digital elevations models (grid DEMs) on large datasets [2]. The major phases of their approach consisted of data segmentation using a quadtree, neighbor finding using said quadtree, and finally interpolation using any of a number of interpolation algorithms. The authors make special note that large quadtrees that do not entirely fit into RAM must be built in stages. Agarwal et al. demonstrate incremental and level-by-level construction of quadtrees, processing small pieces of a single level if the entire quadtree cannot be fit into RAM. Skipping to the third step, the author's description of the data interpolation phase emphasizes that there are a wide range of data interpolation algorithms which could be used, and that the three part algorithm is not data interpolation method specific. However, the key point is that rather than attempting to interpolate the spatial data across all data points, which would be an $O(n^3)$ task, and thus prohibitively expensive, the authors interpolated points within a given node of the quadtree. In order to prevent sharp deviations in the terrain height across quadtree node borders, the authors used the second step of their algorithm, neighbor finding, to find the leaf nodes adjacent to each leaf node, and to incorporate the points within adjacent leaf nodes in the interpolation. The final algorithm has an $O(n/z)$ component (for dealing with each leaf node), and an $O(z^3)$ component for interpolating all of the points within a node and its neighbors, for an overall runtime of $O(nz^2)$.

We focused entirely on the first phase, segmentation, and its parallelization. This was due to our observation that the parallelized construction of quadtrees could be useful not only to the segmentation of lidar data, but have been shown to serve other purposes. A more generalized version of our work could easily extend to quadtrees used for image compression or computer graphics generally. Additionally, GPUs have also been demonstrated to be efficient at solving the interpolation problem mentioned above. While this is not within the scope of this paper, see [2] and [15] for more information on the subject.

3.3 Existing Work on Quadtrees

Quadtrees are not a new data structure, and a significant amount of research has been done on them. Finkel and Bentley [5] provide additional background material on the quadtree data structure. They give a definition and provide algorithms for insertion and balanced insertion, searching and optimization techniques. The authors make the claim that the deletion of nodes and merging quadtrees together is a computationally difficult task. On the other hand, insertion is an $O(n \log(n))$ task and searching is $O(\log(n))$, where n is the number of nodes in the quadtree. These properties are useful because they allow for the quick construction and restructuring of quadtrees, making them the ideal candidates for multiple insertions.

Finkel and Bentley's work focuses on quadtrees using composite keys, which present additional problems which do not exist in the more naïve implementations for data indexing, segmentations, and compression. For example, removal of nodes will not occur for our application because a removal of a node would mean the complete loss of data about that segment of the dataset. Furthermore, merging two trees does not present a problem for the data segmentation phase of our algorithm because we seek to only create one quadtree to index the entire data set. Therefore, even though merging information on two datasets presents difficulties for quadtrees, it will not affect our application. Because of this, the quadtree is well suited for this application, since its primary weaknesses do not play a role in our algorithm.

3.4 GPU Construction of Data Structures

C. Lauter-bach et al.'s work, "Fast BVH Construction on GPUs", provides useful background for our work. In their paper, they present an algorithm for the creation of bounding volume hierarchies (BVHs) [4]. BVHs are important throughout many areas of computational geometry and computer graphics due to the efficiency they provide to algorithms which utilize them.

Because spatial indexing through the use of quadtrees represents a simplified two dimensional analogy for BVHs, understanding fast construction of BVHs represents an important step in making good choices when designing an algorithm for fast construction of quadtrees. In fact, this paper showcases several valuable lessons relevant to our work. First, they choose to implement their algorithm in CUDA because it allows for fine-grain parallelism and the execution of thousands of threads simultaneously. They also make heavy use of shared block memory on the GPU, which runs between 100-200 times faster than global device memory, which represents a potentially significant speed up.

Second, they implement a new algorithm, known as Linear Bounding Volume Hierarchy (LBVH) which reduces the problem of bounding volume hierarchies to one of sorting. The authors map the points to a space filling curve. They then sort hierarchically using the primitives' position relative to each other along the curve. If a parent node occupies the interval (0, X), then its children nodes would occupy some set of (0, a_1), (a_1, a_2), ... (a_{L-1}, X). In this way, each primitive may be sorted without destroying the ancestor's description of their data members, since sorting subintervals does not affect the relative ordering of the data on the larger curve.

The above represents an important concept for our work because we aimed to achieve parallel speedup by implementing an algorithm which reduces the problem of dataset segmentation into a sorting problem. Additionally, we hoped that our parallel implementation of quadtree construction cold achieve significant speed ups because sorting algorithms could make use of the fast, shared memory of thread-blocks.

4 Methods

For the purposes of parallelizing quadtree construction for the GPU, we followed the general idea from [4], described in more detail in [3], that the best method for building a quadtree on the GPU is to reduce it to a sorting problem. First, the initial data set was kept in a single unsorted array. At each step of our construction algorithm, we used a quadtree-sort algorithm on the dataset which assigned each new child node a specific interval of the array containing the dataset being used to build the quadtree. Using this method, individual nodes do not contain a list of all the points contained within them, but rather are represented as specific start and end in a quadtree-sorted array.

Therefore, for the quadtree-sort approach to quadtree construction, the quadtree data structure is comprised of two essential elements: first, the arrays of data points, and second, a two dimensional array of nodes, containing the nodes comprising the individual levels of the quadtrees. The nodes themselves are comprised of a two-dimensional bounding box of the geographic region which they represent, start and end indices for the array of data points, which represent the subset of the data inside the bounding box, and a pointer to their children nodes in the subsequent layer of the quadtree. See Figure 3 for a visual description of the data structure.

4.1 Building The Quadtree

To build the first level of the quadtree, we search the entire data set to find the maximum and minimum x and y coordinates of the data set and use these values to create a bounding box for the data. Then, as we build nodes for each sublevel of the tree, we create a new array of child nodes for the sublevel which consists of approximately (see section 4) four times as many nodes as existed in the previous level. Then, for each parent node, we subdivide that node's bounding box into four equally-sized bounding boxes, and assign them to that parent

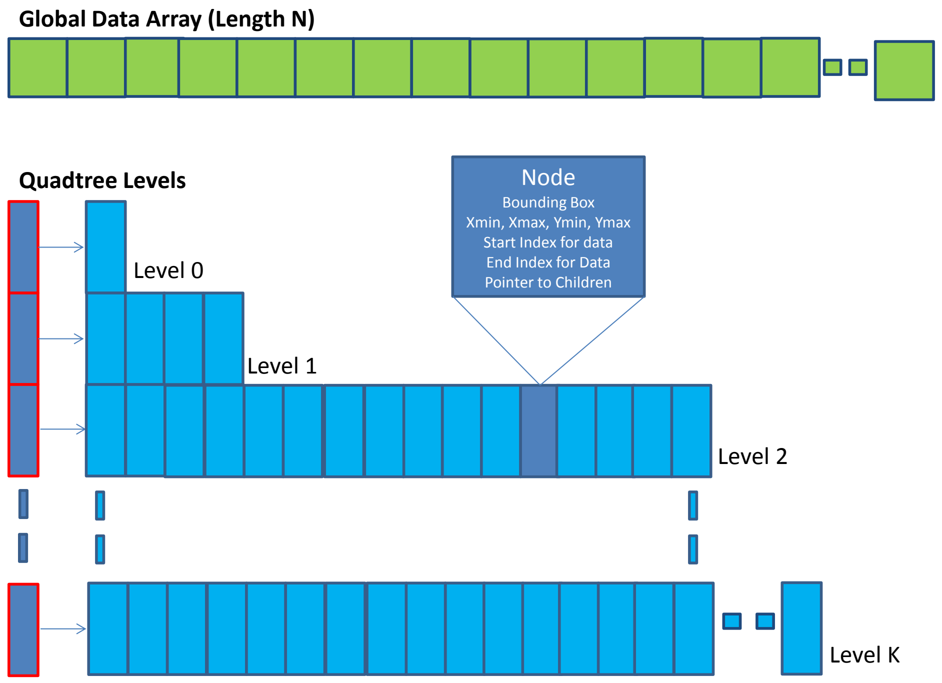

Figure 3: The quadtree data structure for the quadtree-sort method of quadtree construction is comprised of the original data-array, a two-dimensional array of nodes, with each 1-dimensional array comprising a level of the quadtree. Each node consists of a bounding box, start and indices representing their interval of the global data array, and a pointer to its four children.

node's children.

After assigning the child nodes' bounding boxes, we use quadtree-sort. This sort preforms similar to a modified bucket sort with two stages: first, each data point contained in the parent node's bounding box (the points in the data array between the parent's start and end points) is assigned one of the four buckets corresponding to the child node's bounding box to which the point belongs (see Figure 3). 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.

After sorting, we assigned start and end indices to each child node, based on the number of data points in 'their' bucket and 'lower' buckets, in the newly-sorted data array, with the start index of the child with the 'lowest' bucket being start index of the parent node, and the end index of the 'highest' bucket being the end index of the parent node. Because of this, and the fact that all of the data points initially contained in the indices of the parent node will be placed in one of the four buckets of the children nodes, the intervals of the children node represent subintervals of their parent node. More importantly, because all of the data points sorted into the different children buckets remain in the larger parent's intervals; higher levels of the quadtree are not disrupted by the creation of new children nodes and the sorting process. This process repeats until no parent node has more than the threshold value number of points.

4.2 Implementation Choices and Details

The algorithm described above for building a quadtree level-by-level can be easily parallelized on the GPU for relatively large datasets by simply assigning one GPU thread to every parent node, and using a number of blocks equivalent to the number of parent nodes divided by the maximum number of threads in a block to ensure coverage. This is because the sorting process involved in creating the children of a given parent node is functionally independent of the creation of other parent nodes, as any segment of the data array, and the points contained within it, which comprise a parent node will only be read and written to by a single thread. While this naïve solution has been shown to provide some computation speed gains [3], there are a number of inefficiencies and limitations to it. The rest of this section will be describing these problems and our attempts to solve or ameliorate them.

4.2.1 Block Level Parallelism

One of the greatest flaws with the solution presented above is that it only provides parallelism over the number of parent nodes being processed, as demonstrated by [3]. This means that construction of the first few levels of the quadtree is effectively sequential, in addition to the added costs of transferring data to and from the GPU. Because of this, the construction of the first few levels dominates the construction time of the quadtree. One potential solution used by [3] was to build the first few levels of the quadtree on the CPU, and then continue the building process on the GPU. While this did provide some speed gains, compared to a GPU only solution, the construction of the first few levels still dominated the entire construction time. Our solution was to abandon the notion of applying a single thread to each parent node, and instead assign an entire block of threads to each parent node. By using multiple threads, in our case 512, for processing each parent node at each stage, we could achieve a much higher degree of parallelism and therefore potential speed gains.

However, this approach presented a number of difficulties, mainly due to the fact that while processing various parent nodes simultaneously was effectively an embarrassingly parallel task, the actual quadtree sort algorithm presented above was not. Dealing with these difficulties required modifying the basic quadtree sort algorithm, changing it into a four step process. First, each thread processing a parent node would scan through a subset of the data points contained in the parent node's interval (using the thread id and blockDim variables) with the purpose of tagging each point with the bucket, and therefore the child node, in which it belonged, and keep a count of the total number of data points which it had tagged for each bucket in a shared array, which was kept thread safe by each thread writing to a specific portion based on its thread id. Second, after storing these counts in a separate variable, the total number of data points tagged for each bucket by each thread was accumulated into four shared variables, utilizing a parallel reduction. A parallel reduction involves initially using half the number of threads as entries in the array being reduced. These threads combine the data at their own thread id in the array with the entry at their thread id + the number of threads working on the reduction. This process is repeated, reducing the number of threads working by half each time, until the data is compressed into the number of entries desired.

Third, four threads, each assigned to one bucket/child, were used to calculate for each thread the number of points tagged by threads with lower thread id's for their assigned bucket, and to store the result in a shared array based on the thread id of the thread in question. Fourth, each thread repeats its scan of the data points, using the same method as above, and assigns them a new place in the data array in based on the bucket they have been tagged with, which adds a static offset of the total number of points tagged for buckets which represent children corresponding to earlier sub-intervals of the parent interval (calculated in step 2), a second static offset of the number of points tagged by threads with lower thread id's for the given point's bucket (calculated in step 3), and a third offset consisting of the number of points the thread in question has assigned to the given bucket/child, which the thread keeps as a thread-local variable and accumulates as it places points. It should be noted that this is not an in-place sort, and that the tagged points are kept in an separate array and are then copied back individually into the correct places in the main data array.

The algorithm presented above has some obvious deficiencies. The total amount of work which needs to be done is larger than a single threaded approach, as the second and third steps of the algorithm are simply overhead. In total, this means that there is are two $O(n)$

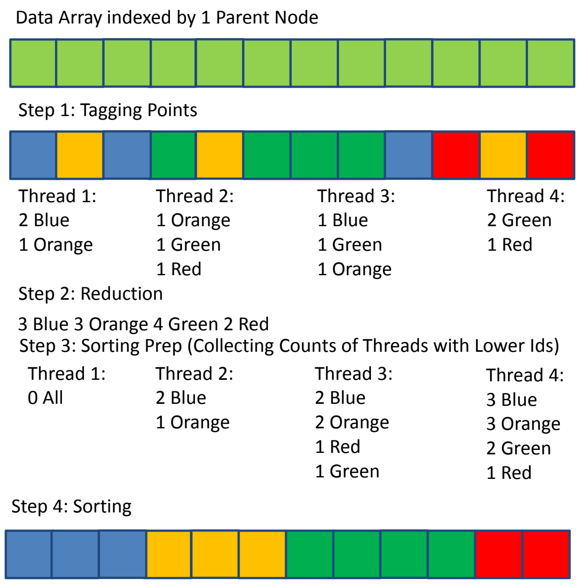
Data Array indexed by 1 Parent Node
Step 1: Tagging Points
Thread 1: 2 Blue 1 Orange
Thread 2: 1 Orange 1 Green 1 Red
Thread 3: 1 Blue 1 Green 1 Orange
Thread 4: 2 Green 1 Red
Step 2: Reduction
3 Blue 3 Orange 4 Green 2 Red
Step 3: Sorting Prep (Collecting Counts of Threads with Lower Ids)
Thread 1: 0 All
Thread 2: 2 Blue 1 Orange
Thread 3: 2 Blue 2 Orange 1 Red 1 Green
Thread 4: 3 Blue 3 Orange 2 Green 1 Red
Step 4: Sorting

Figure 4: This figure shows the four steps of our algorithm. The first step involves all the threads in a block (in this example 4) tagging the points in a particular parent node for residence in one of its four children Nodes. The second step preforms a reduction to collect all of this information together. The third step parcels out information on threads with lower thread ids so that the threads can effectively sort (write to the array) without overwriting each other. The fourth and final step is actually moving the data.

steps (the first and fourth steps), an $O(\log(t))$ step, where t is the number of threads (the second step) and 1 $O(t)$ step (the third step). While the overall amount of work is much larger, the span, or critical path, of the algorithm is significantly shorter. This is because parallelizing the first and fourth steps makes them $O(n/t)$ in length. In other words, the critical path length is $O(n/t + t)$ rather than $O(n)$ for a single-threaded approach. Because the purpose of fast quadtree construction is for large datasets, the benefits in reducing the computation time for the first and fourth steps should outweigh the additional overhead of the second and third steps, leading to an overall faster algorithm.

4.2.2 A Hybrid Approach

After considering the potential benefits of applying entire blocks of threads to processing individual parent nodes, we determined that the complete abandonment of the naïve approach used by [3] was not necessarily optimal. Our block level algorithm was designed to increase the processing speed of parents with large numbers of data points, which constituted the greatest portion of the computation time in [3]; however, it became apparent that if the threshold value for child creation was low enough, some parent nodes might have few enough data points contained within them so that the second and third steps of our algorithm could account for a larger proportion of the computation, and thus reduce speed gains. More importantly, presumably parent nodes which contained few data points would be at the lower levels of the quadtree, where there would be an exponentially large number of parent nodes. Processing these small parent nodes would not fully utilize an entire block's worth of threads efficiently, and the number of blocks that the GPU can run simultaneously is limited.

With the above factors in mind, we determined that a hybrid approach, in which our block level algorithm was used at the higher levels of the quadtree, and the individual thread based algorithm used by [3] was used at lower levels of the quadtree, could be significantly more efficient. Generating a mechanism to determine the most efficient time to switch proved difficult, due to the memory hierarchy of the GPU, specifically the independent memory spaces of blocks. Our solution was to keep an array with size equal to the number of blocks of binary flags, initialized to 0 before each level of quadtree construction, which would be set to 1 by a single thread in any block processing a parent node any of whose child had more than a certain number (based on the threshold value of the quadtree) of points contained within it. We reasoned that if any child node had a large number of data points contained within it, applying only a single thread to its computation in the next stage of construction would lead to significantly longer computation times. If, after any round of computation, all the flags in the array were set to 0, we would continue construction with the single-thread algorithm from [3].

4.2.3 Reducing Memory Requirements of Null Nodes

As mentioned earlier in this paper, one of the major problems with generating large data sets on the GPU is its limited memory space. Therefore, reducing memory requirements as much as possible is vital to allowing our algorithm to run effectively on the largest data sets possible. We focused on the wasteful creation of ‘null’ children nodes at lower levels of the quadtree. As mentioned previously, whenever a parent had fewer than the threshold value of points in its interval, rather than perform the sorting process, we simply created 4 children nodes with a non-existent interval and 0 points, ‘null’ children. This was done because of a limitation of the CUDA architecture [1], in which data structures on the GPU must be allocated by the CPU prior to running a kernel. In other words, the number of children was set before that level of the quadtree was constructed. This problem was compounded by the simple algorithm of generating four times the number of parent nodes as children nodes, as ‘null’ children would then create more null children nodes.

One potential solution to this problem was to copy the children node array off of the GPU after each level of quadtree construction, scan for the number of non-null children, allocate a new children array of this size, copy all non-null children to this new array, and then copy it back to the GPU. However, there were two major problems with this solution. First copying large chunks of memory from and to the GPU is a time-intensive process, which would mean increasing our algorithm’s computational overhead significantly. Second, since scanning the old children array and copy would be done on the CPU, due to the CUDA limitation mentioned in the last paragraph, we would effectively be performing a pair of O(n) operations sequentially. We concluded that together, these major costs to computation speed outweighed the gains we could achieve in expanding the scope of our algorithm.

Having run into these two major problems with our first idea, we set about coming up with a solution which would bypass them. Our solution involved allocating two integer arrays, A1 and A2. A1 consisted of an array sized to the number of blocks being used to run the program on the GPU. After completing our sorting algorithm on a parent node, one of the threads in the block checked how many of the children nodes created by this parent had more than the threshold value of points contained in their interval. This thread then incremented the position in the A1 array assigned to its block with this number. After completion of a level of construction, A1 would be copied off of the GPU and accumulated into a single value, which would be used to allocate the new children array on the GPU. The second integer array, A2, was sized to the number of children nodes being created at the current level of computation, and held a binary value, initialized at 0. After completion of our sorting algorithm a thread in the block assigned to a current parent (a different thread than handling A1, so as to achieve some parallel gains), checked each child node to see if it had more than the threshold value of points, and if so, set its position in A2 (which was the same as its position in the current children node array) to 1. After completion of a level of construction, a second GPU kernel is launched with the current children array, the new children array, and the A2 integer array as arguments.

Then, using a process similar to our block-level algorithm, individual threads scan through the A2 array, accumulate the number of 1’s in their segment of the array, then a single thread accumulates for each thread the number of 1's in threads with lower thread id's, and finally each thread uses this as an offset for copying the children nodes in positions with 1's from the current children array to the new children array. While the current children array would be stored as the current level of the quadtree, the new children array would be used to create the next generation of children, as only children in the new children array would have non-null children. Given that the number of nodes in a given level of the quadtree is exponential, a reduction of even a few nodes early on (or many nodes later) could lead to significantly lower memory requirements.

4.2.4 CPU-GPU Hybrid

While we disagree with [3] that a hybrid CPU-GPU approach to quadtree construction would represent a more computationally efficient solution than our block-level algorithm for non-trivially sized datasets, we agree that such a hybrid is necessary, due to GPU memory limitations. The hybrid approach expands the scope of our algorithm to datasets of a size larger than GPU memory. Our hybrid implementation is rather simple. First, we determine the number of points that can be allocated on the GPU with a fixed number of quadtree nodes. If the number is lower than the actual data set size, instead of computing the quadtree on the GPU, the CPU performs the sorting algorithm sequentially. The CPU keeps track of the maximum number of points in a given child node's interval at the start of each round. If that number is lower than the number determined earlier, the program shifts to computing on the GPU, with each of the current layer's children nodes standing in as the tree's root node in the algorithm described above. After one of the children node's segments of the tree is completely processed, the algorithm shifts to the next child node.

While the above ensures that the GPU has the memory space to handle all of the points contained in the data array it does not sure that it can handle these points and an arbitrary number of quadtree nodes, which can become quite large. Our solution was to perform a check before each level of quadtree construction to see the number of quadtree nodes that the GPU could handle in addition to the entire array of points under consideration. If it was lower than the number of nodes necessary to complete the processing of the next level, we split the level into pieces, performing our algorithms on fractions of the parent node array, and then copying these parent nodes and their children nodes back into memory sequentially. While this process has obvious downsides for computation speed, we determined that these problems would only become acute for cases in which our original algorithm would not function correctly, making the changes worthwhile.

5 Experimental Results

To serve as a baseline, we implemented a CPU-only sequential approach to our algorithm in C. For parallelization on the GPU, we utilized NVIDIA's CUDA with C-bindings and a number of independent kernels. The first was for bounding box assignment for tree-nodes.

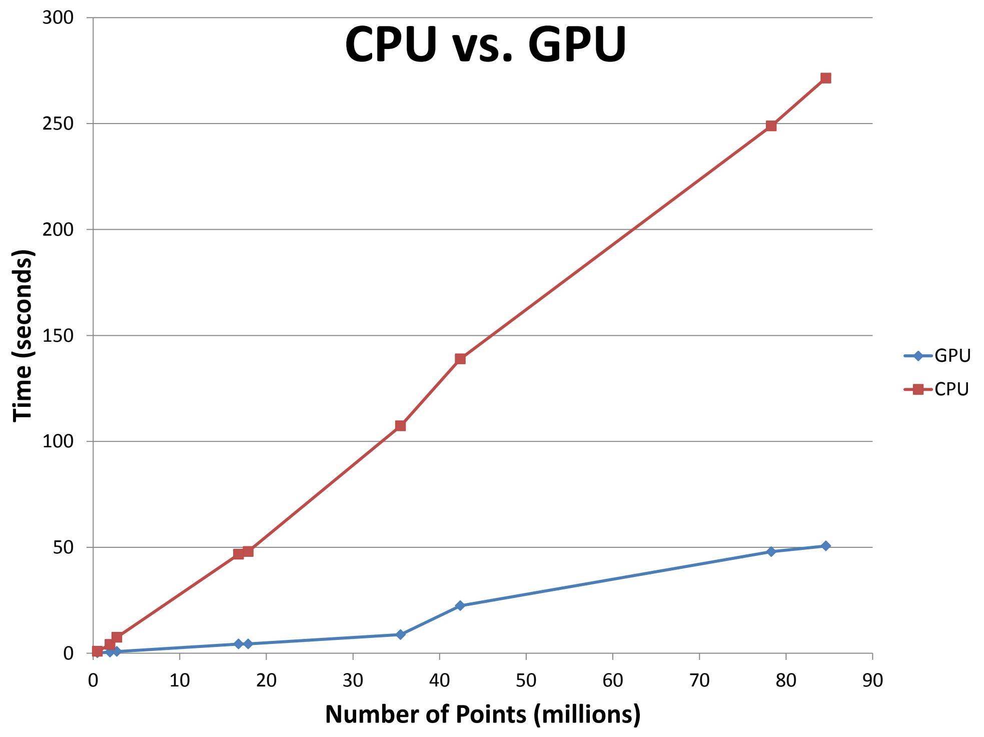

Figure 5: This graph compares the runtime of a sequential CPU approach to our full GPU-algorithm

The second was for the tree pruning elements of our algorithm, described in section 4.3.3. Finally, we implemented our full hybrid block/thread based algorithm, including a system which toggled between using the block and thread based implementation described above, as well as the naïve thread based algorithm, as described in section 4.3.1.

We ran our tests on an NVidia Quadro FX 3800 GPU, with 1000 blocks and 128 threads per block. We ran timed-trials using real world datasets of lidar points taken from the eastern portion of the state of Pennsylvania. These data sets ranged in size from several hundred thousands of points to over 80 million. Each timed experiment was run for ten trials on each dataset.

5.1 CPU vs. GPU

Our first set of tests compared our algorithm to the CPU-only sequential approach described above. and we had a set threshold value of 20 points.

As Figure 5 shows, the runtime of the GPU algorithm is significantly lower than the CPU version. As Figure 6 shows, the relative speed of our GPU algorithm is somewhere between 5× to 12× faster than the sequential CPU approach. It is important to the potentially disturbing trend in both Figures 5 and 6 that the relative speed of the GPU algorithm

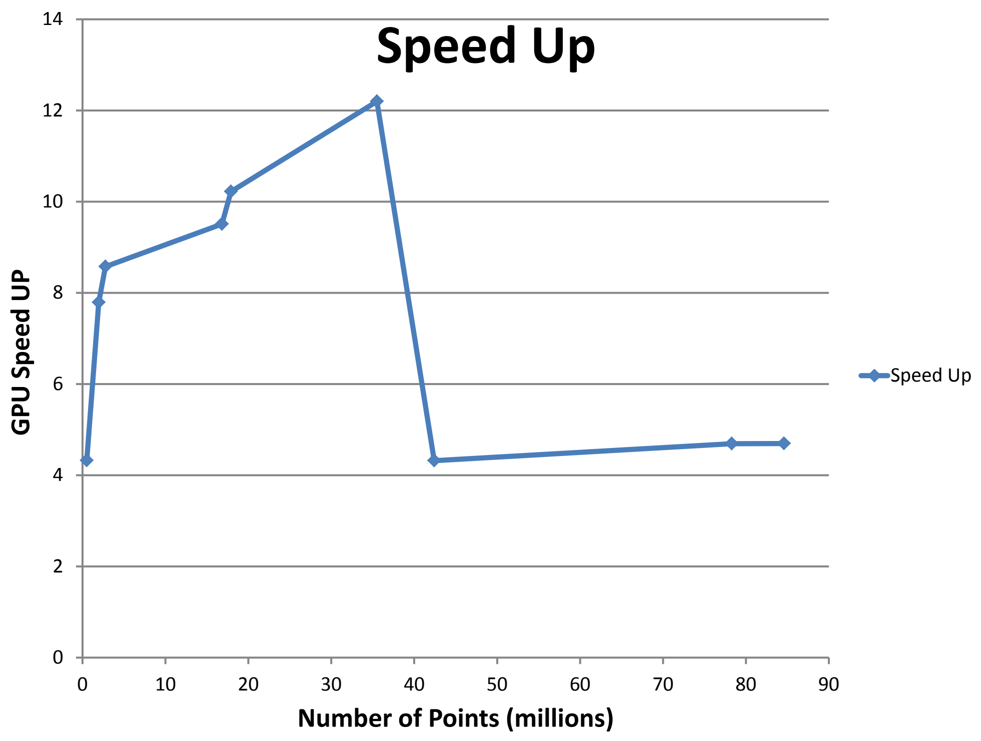

Figure 6: This graph shows the speed up factor of our full GPU-algorithm compared to a sequential CPU approach. The drop in relative speed for 40million+ points is due to point array overloading the GPU's memory

appears to be declining past 40 million points, suggesting its inability to scale to larger datasets. However, this relative slow down is due to the fact that past 40 million points, the GPU cannot contain the entire array of points in memory, and thus the top levels of the quadtree must be built on the CPU instead, which for our experiments used the same sequential CPU approach as the baseline. Removing the runtime of the levels created on the CPU from both our algorithm's runtime and the sequential approach yields the results seen in Figures 7 and 8. These figures show that without the memory constraints of the GPU, the relative runtime of our GPU algorithm to the baseline remains fairly high for large datasets.

5.2 Number of Threads

In addition to a simple comparison between our GPU-algorithm and the sequential CPU approach, we also attempted to demonstrate the value of using thread level parallelism in particular for the runtime of our algorithm. Figure 9 shows the runtime of our algorithm using between 4 and 128 threads per block. As can be seen in Figure 9, the runtime increase is not constant with the number of threads: using 128 threads per block does not make the algorithm twice as fast as using 64 threads, or 4 times as fast as using 32 threads. Having said that, the benefits of large scale thread paralleism are clear to see by the comparitive

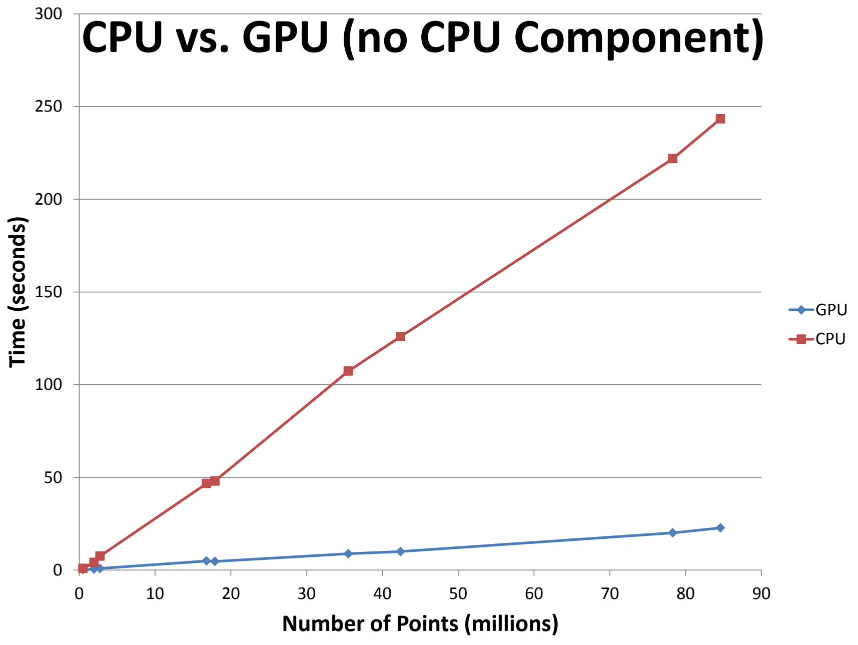

Figure 7: This graph compares the runtime of a sequential CPU approach to our full GPU-algorithm without the levels which are built on the CPU by our GPU-algorithm factoring into the runtime of either algorithm

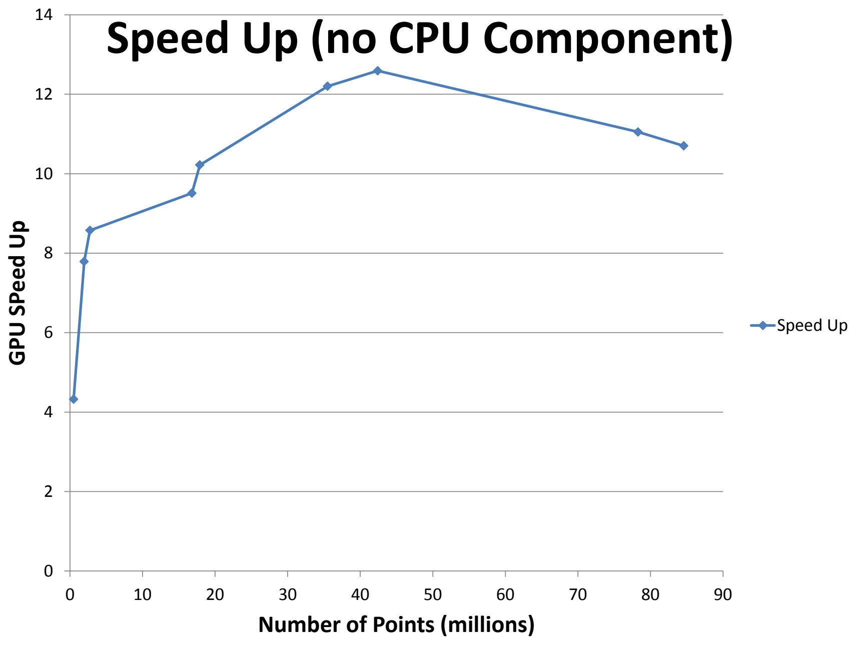

Figure 8: This graph shows the speed up factor of our full GPU-algorithm compared to a sequential CPU approach, but with the levels which are built on the CPU by our GPU-algorithm not factoring into the runtime of either algorithm

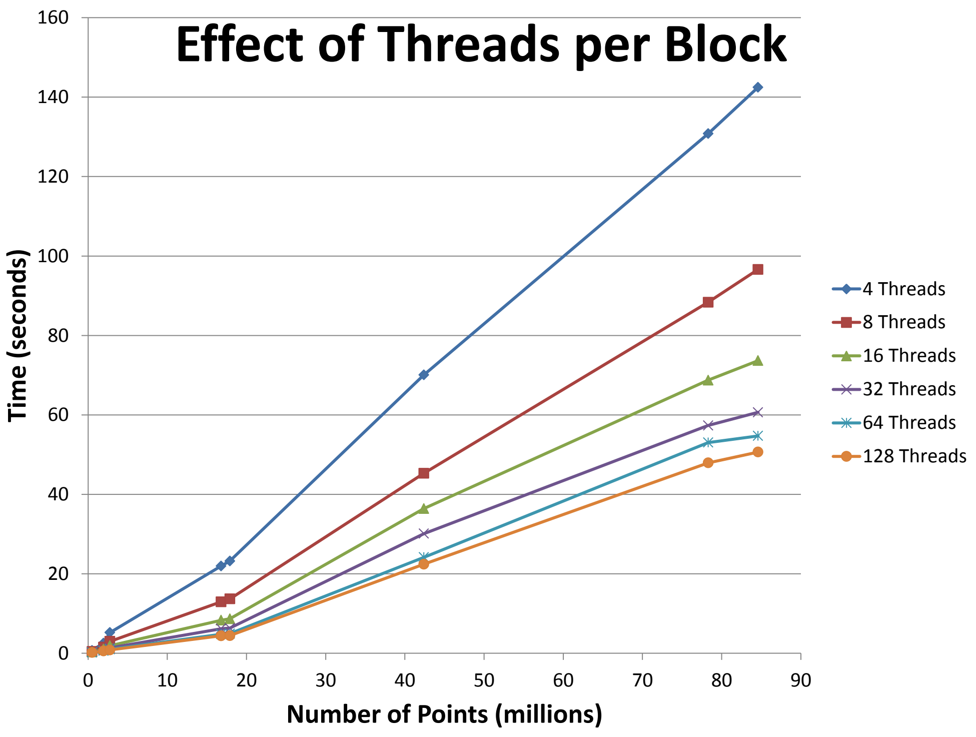

Figure 9: This graph shows the gains of thread level parallelism on the GPU by showing the speed of our algorithm when using differing numbers of threads per block over a range of input points sizes

speed of the runtimes of the algorithms using more threads.

5.3 Threshold Value

During our discussion of the fundamentals of quadtree, we noted that quadtree node creation continues until no node has more than a threshold value of points. For most of our testing purposes, we tested with this threshold value set to 20, for two reasons. First, for the sake of consistency between tests. Second, because 20 is a remarkably low threshold value when dealing with inputs sets of tens of millions of poins. Because lower threshold values means creating further levels of the quadtree, we tested at 20 in order to demonstrate that our algorithm could function under any reasonable threshold value for the size of the data sets we tested on.

However, it is also important to note that our algorithm can function for differing threshold values, and that higher threshold values do, in fact, lead to lower runtimes. The results of our experiments with theshold values can be seen in Figure 10. Using our regular test of $z = 20$ as a baseline, we also tested with z values of 200 and 2000. As Figure 10 showcases, higher threshold values did have a positive effect on runtime; however, its effects were not

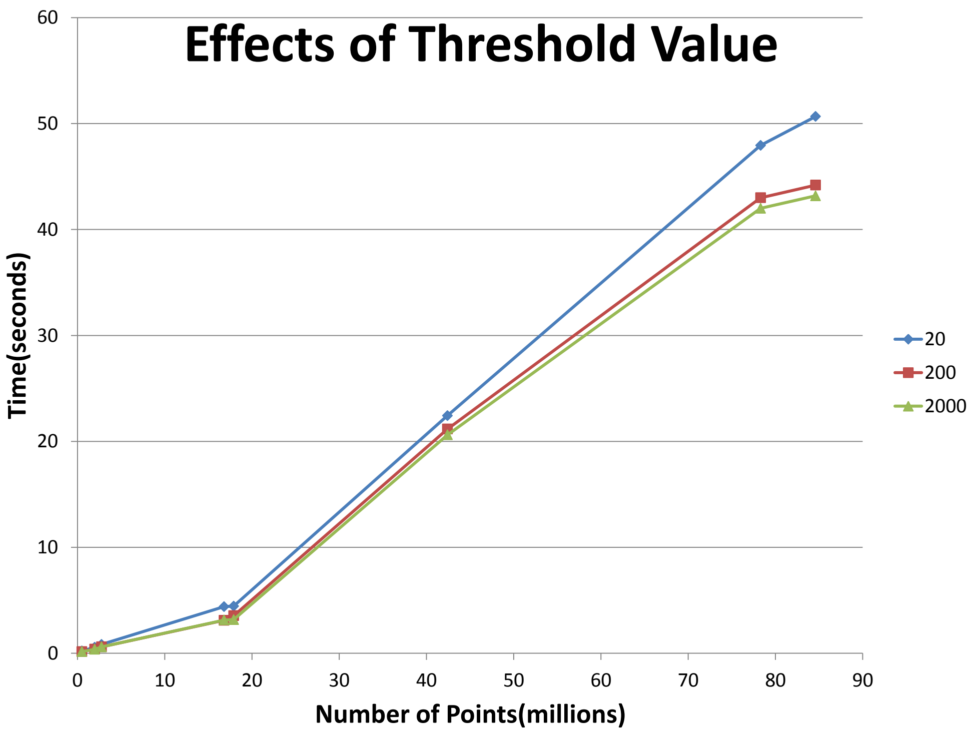

Figure 10: This graph shows effects of threshold value on the runtime of our algorithm, comparing z of 20, 200, and 2000

major.

6 Conclusions

Overall, we propose a hybrid CPU-GPU approach to the problem of quadtree parallelization which involves, depending on the size of the dataset in question, building the first few levels of the quadtree on the CPU to reduce memory constraints, then transferring the data to the GPU to compute the remaining levels of the tree, using both block and thread level parallelism to process individual parent nodes, additionally, we perform pre-processing on children nodes to determine if they have dropped below the threshold value, and remove them from splitting into children at the next level if they have.

The specific purpose of work was 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, and to build off the work done in [3] by fully utilizing both the block and thread level parallelism available to us on the GPU. By using block level parallelism for processing a single parent node, we altered the problem from being embarrassingly parallel at each level, since each parent node was considered separately, to a far more complex parallel problem, requiring extra computation in order to prevent race conditions and data races between GPU threads. The computation time gained from using our more complex algorithm demonstrates not only the viability for General Purpose GPU solutions to non-graphics and non-embarrassingly parallel problems, but that the GPU can allow more complex, non-embarrassingly parallel, solutions to problems to yield faster results, if they are more capable of utilizing the massively parallel nature of the GPU to ensure that its processors are doing useful work for a higher percentage of the time.

7 Future Work

There are a number of potential areas for future work relating to our quadtree construction algorithm. One relatively simple addition would be to parallelize the CPU component of our hybrid algorithm using one of the other parallel frameworks mentioned at the beginning of this paper. Given that the CPU component of our hybrid algorithm represents a significant portion of the overall runtime, this could yield a substantially reduced overall runtime. We did not perform this parallelization as our primary focus was on the gains that could be achieved on the GPU, with the CPU component mainly consisting of a method by which we could extend our solution past the memory constraints of the GPU.

A second potential area for future work would be to expand our hybrid CPU-GPU solution to a distributed CPU-GPU solution, which would allow us to run our algorithms on multiple machines, and GPU's simultaneously. Depending on the necessary level and extent of communication required between machines in order to maintain correctness, this could allow us to extend the runtime gains of the GPU element of our solution past the GPU's memory constraints simply by introducing another level of parallelism.

References

[1] CUDA Programming Guide Version 5.0

[2] P. K. Agarwal and L. Arge and A. Danner, From Point Cloud to Grid DEM: A Scalable Approach, Progress in Spatial Data Handling. 12th International Symposium on Spatial Data Handling, Springer-Verlag, 2006, 771–788, Andreas Riedl and Wolfgang Kainz and Gregory Elmes

[3] Kelly, Maria, and Alexander Breslow., "Quadtree Construction on the GPU: A Hybrid CPU-GPU Approach.", http://www.sccs. .edu /users/10/mkelly1/quadtrees.pdf

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

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

[6] Papadopoulos, A. N., and Y. Theodoridis, R-trees: Theory and Applications, Springer, 2005

[7] Lewis, Bil, and Daniel J. Berg, Threads primer: a guide to multithreaded programming, Prentice Hall Press, 1995.

[8] Chapman, Barbara, Gabriele Jost, and Ruud Van Der Pas, Using OpenMP: portable shared memory parallel programming, Vol. 10, MIT press, 2008.

[9] Zhang, Jianting, Simin You, and Le Gruenwald, "Indexing large-scale raster geospatial data using massively parallel GPGPU computing.", Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems, ACM, 2010

[10] Gatziolis, Demetrios, and Hans-Erik Andersen, "A guide to LiDAR data acquisition and processing for the forests of the Pacific Northwest.", (2008).

[11] Hanan Samet, The Quadtree and Related Hierarchical Data Structures., ACM Comput. Surv. 16, 2, (June 1984), 187-260, DOI=10.1145/356924.356930, http://doi.acm.org/10.1145/356924.356930

[12] Owens, John D., et al., "A Survey of General Purpose Computation on Graphics Hardware." Computer graphics forum. Vol. 26. No. 1., Blackwell Publishing Ltd, 2007

[13] Owens, John D., et al., "GPU computing.", Proceedings of the IEEE 96.5, (2008): 879-899

[14] Kakde, Hemant M., "Range Searching using Kd Tree.", (2005):, 1-12.

[15] Beutel, Alex, Thomas Mølhave, and Pankaj K. Agarwal., "Natural neighbor interpolation based grid DEM construction using a GPU.", Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems., ACM, 2010.

[16] Eppstein, David, Michael T. Goodrich, and Jonathan Z. Sun., "Skip quadtrees: Dynamic data structures for multidimensional point sets.", International Journal of Computational Geometry & Applications 18.01n02, (2008): 131-160.