Characterization of CUDA and KD-Tree K-Query Point Nearest Neighbor for Static and Dynamic Data Sets

Brian Bowden
bbowden1@vt.edu

Jon Hellman
jonatho7@vt.edu

1. Introduction
Nearest neighbor (NN) search is a common Computer Science problem, but its optimization on General Purpose Graphics Processing Units (GPGPUs) is far from being complete. Finding a solution to the nearest neighbor problem in 2-D and 3-D dimensions would be helpful in applications like computer graphics, computer vision, and Geographic Information Systems. A nearest neighbor application could be helpful in situations such as finding the closest distribution center for mail, the closest point on a map, or pattern recognition. For higher dimensions, finding the nearest neighbor could be used in order to do a “similarity search in databases (text, images, etc), find pairs of similar objects (copyright violation detection), and use subroutines for clustering” [7].

Nearest neighbor search has a few different variations. Some variations of the problem are to find one point’s (known as the query point) nearest neighbor, to find a query point’s k nearest neighbors, find k query points nearest neighbors, to find every point’s nearest neighbor, or to find every point’s k nearest neighbors. In our project and discussion, we will focus on k-query point’s nearest neighbor, which we will call “K Query Points.”

2. Problem Statement
Nearest neighbor search is a problem in which:

Given a set of n points S, in some d-dimensional space X and a distance (or dissimilarity) measure M, our task is to preprocess the points in S in such a way that, given a query point q, which is a member of X, we can quickly find the point in S which is nearest (or most similar) to q. [4]

A simple approach to solving the nearest neighbor problem on a single core CPU could be done by comparing all of the data points to the query point. This brute force approach takes linear time to compute. The brute force approach has the downside that it compares points that are far away from the query point instead of only focusing on the closer points. For a single query point, we have a running time of O(dN), which means it can take a long time if the number of dimensions and/or points are large. For the K query point variation, the running time becomes O(k*dN) for the naïve implementation, which is fairly inefficient if any of the variables is large. For this project, the number of dimensions was kept constant at d = 2, so the running time for k query points would be O(kN). The parallel implementation could decrease the execution by a factor of c, where c is the number of cores being utilized. This means the k query points would be O(kN / c).

Using a KD-Tree can speed this up for smaller dimensions by reducing the search space to only a fraction of the total search space, but there is preprocessing time plus the time to prune and search the tree. For larger dimensions, it becomes more difficult to partition the search space and KD-Trees quickly become inefficient [2]. However, since we are keeping d constant, then KD-Trees should retain their efficiency. KD-Trees have a O(nlogn) preprocessing time to create the tree and O(logn) per query point to search the tree. This means for k query points, it would be O(nlogn + klogn) to find the nearest neighbor for k query points. Creating the tree is often the longest part since the queries can done extremely quickly once the tree is made. If the data set is static, then the tree only needs to be created once and updated infrequently. If the data set is dynamic, then the tree has to be created multiple times or updated extremely frequently, which significantly reduces the gains from the fast queries.

Parallelizing the query search can be done on the CPU if it has multiple cores using a programming language such as OpenMP or pthreads. However, when using a GPU, the programmer has hundreds of cores at his or her disposal and needs to program in such a way to make efficient use of these cores. Since GPU cores are much simpler than CPU cores, the GPU algorithm needs to be designed with as few branches as possible and the programmer must try to get as much parallelism as possible by focusing on maximizing throughput. This means that trees, which are branch dependent, are less suited for the GPU and makes trees poor choices to use on the GPU unless the programmer adds several levels of complexity to the implementation.

3. Related Work
Although recent, nearest neighbor search is not totally new to GPUs. A number of different approaches have been taken in order to solve this problem. An approach taken by Bustos [5] was to “map vectors into texture data, without restrictions on the dimensionality of the data.” Their approach involved using texture reduction, and a linear scan through the elements to return the nearest neighbor. Their results were a significant speedup over their CPU.
implementation, with a great ability to analyze high order dimensions (16 < d < 256) and large database sizes (7 million objects) [5]. Their study did not test dimensions of size less than 16. A reason for this is that a linear scan would not be the optimal algorithm to use for lower dimensions.

A lot of the analysis on which data structure and which algorithm to use for nearest neighbor search has already been performed. Kibriya in *Fast Algorithms for Nearest Neighbor Search*, has performed a highly extensive analysis on the different data structures and algorithms used in nearest neighbor search with the following results:

“In terms of n, ideal solutions exist with logarithmic query time, and near linear preprocessing and linear space requirements, for d <= 2. For d = 1, a simple binary search on a sorted array gives O(logn) query time in the worst case, while requiring only O(n) space and O(nlogn) preprocessing time (which equates to time required for sorting the array in this case). For d = 2, O(logn) query time in the worst case, with linear space and near linear preprocessing time, is possible using methods based on Voronoi diagrams [9]. However, for d > 2, no known solution exists that can guarantee a sublinear query time while still keeping the space complexity linear and the preprocessing time near linear.” [4]

Kibriya’s results would support that nearest neighbor search using a linear scan would not be optimal for dimensions lower than 16. A similar approach was taken by Garcia [1], who presented a naive brute force k-nearest neighbor search algorithm on the GPU. His results show a 64x and a 189x speedup on CUDA and CUBLAS implementations respectively, on synthetic data over the nearest neighbor search computed on the CPU using the ANN C++ library. However, these speedups were only achieved when the number of dimensions was greater than 16, with greater speedups being achieved with a greater number of data elements.

For lower dimensions, an approach by [6] proved effective. Zhou used a breadth first search kd-tree and a voxel volume split heuristic to achieve a search time that was 5x – 10x faster than the CPU implementation. While most approaches involve a preprocessing step of building the kd-tree, or other data structure, on the CPU, Zhou was able to build the kd-tree in real time with a 9x – 13x speedup.

At a larger scale, in [8], the authors were able to take advantage of multiple graphics processors to solve this problem. Although using multiple graphics processors is outside the scope of this project, there are a few important points brought up by this article. Kato mentions that their algorithm is effective because “1) there is no synchronization between GPUs until the very end and 2) the workload is well balanced” [8]. Applying the first lesson to maximize performance on one GPU, synchronizing between threads within a block, or even between blocks within a grid will ultimately slow down performance. These actions should be avoided or performed only if absolutely necessary. Additionally, the work between threads and between blocks should be well balanced so that the entire nearest neighbor algorithm can finish as quickly as possible.

4. Implementation

For the implementation of the static data set, we only compared the execution time for the CPU KD-Tree query time against the GPU KD Tree query time. For the dynamic data set, we compared a GPU Brute-Force implementation against a CPU KD-Tree including the time to build the tree to compare how the two approaches would work. We also implemented a CPU Brute-Force as a baseline in order to see how the other approaches would compare. These tests were run on the Fermi machine, which uses a GTX 580 Nvidia GPU and an Intel i7 3.20 GHz CPU.

For the implementation of the GPU KD Tree kernel, an array representation of a kd-tree was used. This facilitated the quick calculation of jumping to a child or to a parent. The formula 2k + 1 was used to jump to a point’s left child, where k is the current point’s index which is stored in a register. The formula 2k + 2 was used to jump to a point’s right child. To jump to a parent from a left child or right child, (k-1) / 2 and (k-2) / 2, respectively, was used.

Using an array representation rather than a struct with pointers gives a few advantages. For each data point of a struct, one integer value and an additional two pointer addresses would need to be stored in global memory, meaning that 3x as much data would be needed. For an array representation, only the integer values would need to be stored.

For the creation of our kd-tree, we split the data based on x-median first, y-median second, and alternated splits until all of the data was added into the array. Padding values were given an integer max value. The number of nodes in the array was equal to 2^d – 1, where d is the depth of the tree. Using a struct with pointers would use less space if the tree’s depth was very large and therefore had a lot of padding values. The kd-tree used in this implementation, however, was balanced, and therefore had a lower depth and a lower number of padding values, saving space.

The GPU KD Tree search algorithm works as follows:
1. Perform a Depth-First-Search to find the deepest box that the query point is in. As the tree is traversed, the distance from the query point to the current point is calculated. The minimum distance is updated based on the current distance and the current index is saved.
2. When the deepest box is found, the minimum distance is used as the current radius. A pre-order traversal is then
performed on the rest of the data set, beginning right at the point where the depth-first-search finished. A point is only further traversed if the distance from the query point to the bounding box is less than the current radius. Points are pushed and popped from the stack as necessary.

3. When the last point has been popped off of the stack, the algorithm has finished, and the current index is equal to the nearest neighbor index, and the current radius is equal to the nearest neighbor distance.

Further explanation of the details of the algorithm is necessary to understand how the algorithm was implemented. Each query point was performed by one thread, although multiple query points could be performed by one thread if desired. Each thread, for the depth-first-search and pre-order traversal, stored information in the thread’s personal stack. This personal stack stored (depth of tree x number of parameters x 4bytes/int) bytes of information. The four parameters that were stored were: leftVisited, rightVisited, bounding box stored value, and bounding box stored location. The actual indexes of the points on the stack were never stored in this implementation. They didn’t need to be stored because the simple jump calculations could be performed in order to traverse the array.

One common method used in a traditional depth-first-search algorithm is to have an array of N values which store whether a point has been visited or not. When the point is visited, the Boolean or integer value is updated, and then the next point is traversed. However, this strategy requires having N values in order to keep track of which point has been visited. Furthermore, if thousands of threads are running, each thread would need to have its own array of N values for the thread’s traversal. This implementation takes a different approach. In each level of the stack, (where the level represents the current point in the stack), an integer is used to store whether this current point’s left or right child has been visited. In total, only “2 x depth of stack” values maximum would need to be stored. This represents a huge storage savings and a highly parallelizable solution for depth-first-search or pre-order-traversal for large numbers of threads on the GPU.

The next two parameters, bounding box stored value and bounding box stored location, will now be discussed. Four bounding box values are stored in registers, one each for bottom, top, left, and right. These values are initially equal to the minimum and maximum x and y values of the data points. When a new point is pushed onto the stack, the bounding box values are updated. The following changes are made, depending on whether the current depth is on an X-Split or Y-Split, and whether a push to the left or push to the right occurred:

push to left child:
  - on X-Split: bounding box top = data point Y
  - on Y-Split: bounding box right = data point X
push to right child:
  - on X-Split: bounding box bottom = data point Y
  - on Y-Split: bounding box left = data point Y

However, whenever a push occurs, the current bounding box values are overwritten. Since the bounding box values will need to be retrieved when a pop occurs, the bounding box value and location (bottom, top, left, or right) is stored in the stack on a push.

This strategy removes the need for an array of size 4N, which stores the four bounding box values of each point. The bounding box is simply calculated on-the-fly at each point instead.

Further optimization of the GPU KD Tree implementation would include making use of shared memory or registers to store each thread’s stack information. Each thread’s stack was stored in global memory in this implementation. For a depth of 24, which could represent about 224 or 16.7million data points, a thread stack size of 96 (24 x 4) integers would be needed. For an Nvidia Kepler card, which supports up to 255 registers per thread, the thread’s entire stack could be stored in registers, which could lead to a very significant performance increase. This increase would be the result of registers having a bandwidth of 8TB/s, while global memory only has a bandwidth of about 177GB/s. For Tesla or Fermi cards, which support lower numbers of registers per thread, further optimization would include storing at least part of the stack in shared memory. The current implementation, however, stores data for each threads

4.1 Static Data Set
For the static data set, we would build the tree on the CPU using an implementation done by [10]. Once the tree was built, we would time how long it takes the CPU to calculate the k query points. Since we are assuming the data set is static, then we can assume that the nodes in the tree won’t need to be changed and we can simply send queries to the tree.

For the GPU implementation of the k query search, we first converted the kd-tree to an array representation because traversing a tree is not well suited for the GPU due to the amount of divergent branching that would occur. A Breadth-First Search was used to traverse the tree and map each node in the tree to an index in the array. Trees can be represented in arrays since if a parent is at index i, then its two children are at indices 2i and 2i + 1 respectively. A downside to this approach is that if the kd-tree is not full on every level, then there ends up being wasted space due to padding that is needed. Once we have the array representation, then we transfer it to the GPU and perform the queries on the GPU and transfer the results back to the CPU. MAYBE MORE INFO HERE

4.2 Dynamic Data Set
For the dynamic data set, we once again build the tree on the CPU. When we time the CPU query time, we also
include the time it takes to build the tree since we are assuming that the tree requires a different data set every time or that there are nodes constantly being added and deleted from the tree. The time taken for the CPU KD-Tree to compute the k queries is then compared to the time it takes for the GPU to transfer the array over, perform the search, and then transfer the results back. The GPU implementation was done using a parallelized brute force approach where the GPU tries to calculate as many queries as possible during each kernel to reduce the number of kernel launches needed.

5. Results

5.1 Static Data Results

<table>
<thead>
<tr>
<th># Query Points</th>
<th>GPU Query Time (s)</th>
<th>Build Tree (s)</th>
<th>CPU Query Time (s)</th>
<th>Total Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>3.598</td>
<td>7.707</td>
<td>0.0223</td>
<td>7.729</td>
</tr>
<tr>
<td>10000</td>
<td>6.904</td>
<td>7.865</td>
<td>0.0454</td>
<td>7.910</td>
</tr>
<tr>
<td>20000</td>
<td>13.742</td>
<td>7.738</td>
<td>0.0874</td>
<td>7.825</td>
</tr>
</tbody>
</table>

For these tests, we varied the number of query points and the number of nodes in the tree. Shown in both the graphs, the CPU KD-Tree performs in about the same amount of time for both 4 million and 8 million points. However, the GPU implementation performs significantly better for 4 million points, but performs slightly worse for 8 million. What’s interesting to note is that the 8 million points show that both implementations increase in computation time, so the GPU implementation only performs worse by a constant factor in all cases. Since the current implementation of the GPU KD-Tree is the un-optimized version, then with further optimizations, the GPU version could possible perform better for 8 million as well.

A general trend that was observed was that the GPU would tend to perform better when n was smaller, which is likely because of the time taken to not only transfer the values, but because of the amount of work each thread has to do increases significantly. The CPU KD-Tree query time scales extremely slowly for n in comparison since each query is only O(logn). The GPU KD-Tree query time is also O(logn), however, the GPU KD-Tree query time experiences the overhead of loading elements to and from the stack, which is seen at higher values of n.

5.2 Dynamic Data Results

For the dynamic data set, we noted that the closer k was to n, the complexity would approach O(n^2 / c) for the brute force and O(2nlogn) for the KD-Tree query time. Based on this observation, we estimated that the GPU would outperform the KD-Tree if n was significantly larger than k and the KD-Tree would perform better if the values were close together. The results we found confirmed our theory as we noted that the larger k was, the larger n had to be before it began to outperform the KD-Tree time. As we can see in Figure 3, doubling the number of query points roughly doubles the time for the GPU, but the time for the KD-Tree roughly remains the same since only the query time doubles. The query time makes up an extremely small portion of the overall KD-Tree time, so increasing the number of query points only leads to an extremely small increase in the computation time.

However, as we can also see in the table, doubling the number of points only doubles the execution time for the GPU, but more than doubles the time to build the tree. This indicates that the GPU brute force scales much better to the number of points, but the KD-Tree scales much better with the number of query points.

To see what values of n and k the GPU performs better than the CPU KD-Tree and vice-versa, n was scaled from 1 million to 20 million points while the number of query points used were 1, 500, 5000, 10000, 15000, and 20000.
As we can see in the table, the higher the number of points, the more likely the GPU Brute Force is to be the best algorithm choice. However, we can also see that the KD-Tree scales more strongly with the number of query points, and doubling the number of query points from 10000 to 20000 resulted in the KD-Tree being the best algorithm choice for n less than 1 million to being the best choice for n less than 20 million.

6. Conclusion
For the static data set, we found that the GPU KD-Tree query implementation scales very well with the number of query points, but the CPU KD-Tree scales much better with the number of nodes in the tree. These results are based on the un-optimized GPU implementation however, so an optimized implementation could perform much better at higher values of n to the point where it is always the better choice.

As for the dynamic data set, we found that the KD-Tree scales extremely well with the number of query points, but doesn’t scale as well with the number of points due to the time it takes to build the tree. The GPU implementation works well with large values of k and n, but increasing the number of query points impacts performance more significantly than the number of points. From this, the GPU implementation should be used when k is not too large for any value of n and the KD-Tree should be used when k grows to be extremely large for a dynamic data set. The impact of building the tree can be significantly reduced if parts of the tree can be reused or if the tree can be stored and read from memory instead of rebuilding it. Parallelizing the building of the tree might also be the best means of decreasing the runtime of the CPU KD-Tree. The GPU implementation can also be further optimized and adding additional GPUs could further help parallelization of the brute force approach.

7. References

8. Team Member Contributions
For this project, Brian worked on the CPU side by working on the KD-Tree and the query search as well as the Breadth-First Search that is used to convert the tree to an array. Jon worked on the GPU side and worked on the GPU Brute-Force as well as the GPU KD-Tree query. Both partners contributed to the report and the presentation equally.