A Project Report

On

# GPU Accelerated Range Aggregate Data Structures

BY

**Shikhar Bharadwaj**

**2014A7PS0113H**

Under the supervision of

**Prof Tathagata Ray**

**SUBMITTED IN PARTIAL FULLFILLMENT OF THE REQUIREMENTS OF**

**CS F366: Laboratory Project**

**BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI (RAJASTHAN)**

**HYDERABAD CAMPUS**

**(November 2017)**

# Acknowledgements

Foremost, I extend my deepest gratitude towards Prof. G. Sundar, the Director of BITS Pilani Hyderabad Campus for giving me the opportunity to receive education in his prestigious institute.

I am highly indebted to Prof. Tathagata Ray for his guidance, motivation, enthusiasm and immense support. I would be unaware of the gems of this field but for his motivation which generated in me an interest for Computational Geometry.

I am thankful to BITS infrastructure for providing me necessary resources.

Lastly, I would like to extend my thanks towards my family who were always there to help and give their best suggestions.

**-Shikhar Bharadwaj**

**Birla Institute of Technology and Science-Pilani,**

**Hyderabad Campus**

## Certificate

This is to certify that the project report entitled "**GPU Accelerated Range Aggregate Data Structures**" submitted by Mr. Shikhar Bharadwaj (ID No. 2014A7PS0113H) in partial fulfillment of the requirements of the course CS F366, Study Oriented Project Course, embodies the work done by him under my supervision and guidance.

**Date:27-11-17**

**(Prof. Tathagata Ray)**

BITS- Pilani, Hyderabad Campus

**Abstract**

Range Tree is the data structure used for “Point in a Box” queries and “Skyline” finding in Computational Geometry along with many other applications. CUDA is a framework provided for Parallel Programming by NVIDIA that runs on top of NVIDIA CUDA architecture GPUs. This project report explains implementation of two dimensional Range Tree on CUDA to use the power of GPU for accelerating construction and query times of Range Tree. The implementation uses linearized tree structure for Range Tree instead of using pointers. The secondary tree is built layer by layer from bottom to top and stored as arrays. Our implementation provides a speedup of 3X from the recursive code to implement the same structure.

# CONTENTS

## Previous Work

Fix et. al in [1] showed that instead of developing brute force algorithms for the GPU, sophisticated algorithms that use a hierarchical data structure can lead to an efficient way of solving the problem at hand. This is because the time penalty for synchronisation between threads is generally overcome by the logarithmic nature of such data structures. They showed this by implementing an accelerated B+ tree on CUDA.

Kim and Nam [2] proposed a Massively Parallel Three phase Scanning (MPTS) R-Tree traversal algorithm that converts recursive access of tree nodes to sequential access. They also showed that the braided parallel indexing improves system throughput when a large number of concurrent queries are submitted. Data parallel partitioned indexing methods on the other hand help in reducing individual query time.

Maramreddy and Kothapalli [3] proposed that hierarchical data structures can be efficiently constructed on modern parallel architectures using in built APIs, such as Thrust [4] for CUDA. They worked on Range Tree and proposed the flattened tree architecture for the tree that we have used in our method of construction.

Kelly and Breslow [5] proposed a level by level approach to Quad Tree construction. They worked in a top down manner to generate the quad tree by reducing the construction task to a sorting problem. Similar methodology has been adopted in our work to construct the Range tree. Foley and Sugerman [6] demonstrated two k-d tree traversal algorithms suitable for GPU implementation and integration into streaming raytracer.

# Range Tree and Range Querying

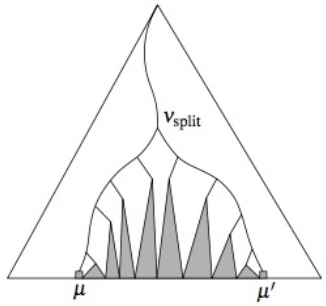
**Range Searching -** Efficient query of d- dimensional points lying inside an orthogonal query rectangle. Range searching finds its applications in database systems, mobile computing and geographical information systems. To perform the query efficiently we preprocess the given data points in the form of a data structure. The data structure studied is Range Tree.

**Range Tree** - The intuition for range tree data structure comes from the fact that a query in d dimensions can be broken down to d one dimensional queries where each next dimensional query works on the set of points filtered from the query before. Following definitions enable us to define the structure of Range trees-

Canonical subset of a node - Subset of points stored in leaves of the subtree rooted at a node v is the canonical subset of node v. For example, the canonical subset of root of tree is the whole point set.

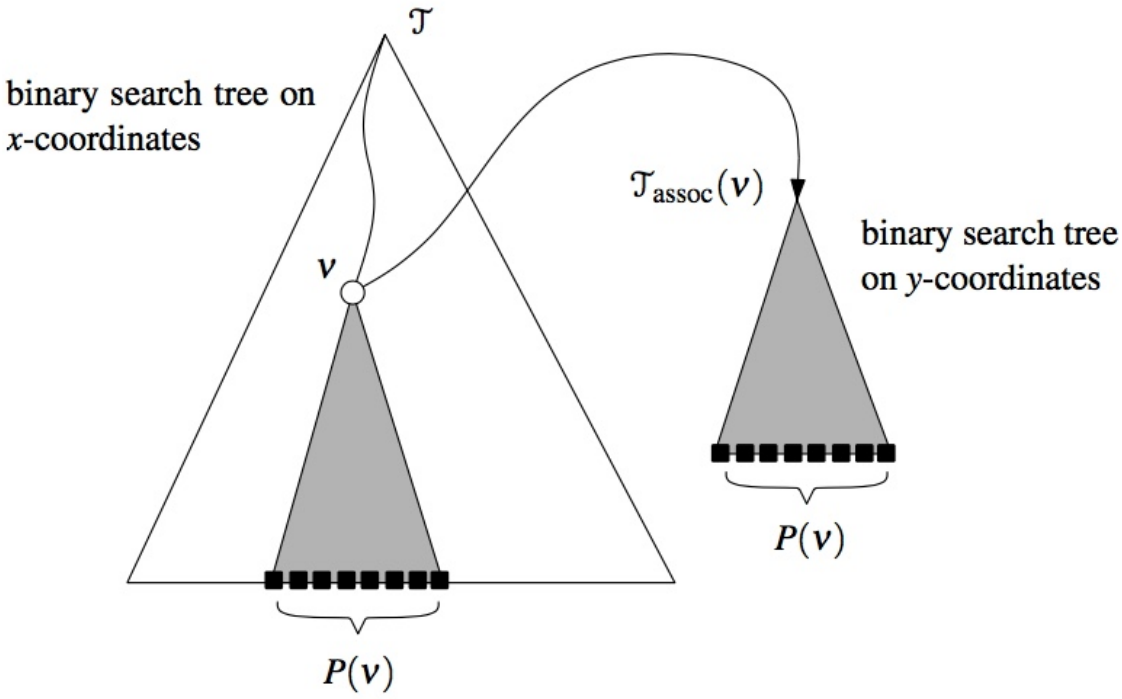
Canonical node of a subset - For the subset of points contained in the leaves of subtree rooted at a node v, its canonical node is v.

Range tree is a multi level hierarchical data structure where each level is a balanced binary search tree on one dimension of the dataset. The filtering of data points happen as we move from one level to another. Once we have searched the first dimension in the first level tree we get the filtered point set for next dimension search as follows-



• For any internal node v there is an associated binary search tree on next dimension composed on the canonical subset of v. Suppose the query is [x1,x2] [y1,y2] , we first search for the nodes x1 and x2 in the first level tree. In the figure on right the path for x1 and x2 splits at $v_{split}$. Let the path to x1 from this node take left turns then the corresponding nodes on right side for each turn forms a list V. By symmetry, the path to x2 gives a list W. The filtered subset for next dimensional search is formed by taking union of canonical subsets

of all canonical nodes v $\epsilon$ V , all w $\epsilon$ W, μ and μ' A d dimensional range tree requires $O(n \log^{d-1} n)$ construction time and space.



# CUDA

CUDA stands for Compute Unified Device Architecture. It is a programming model that allows programming for the GPU. More specifically, it is an extension of the traditional C language with some new constructs to allow GPU programming use "kernels."A cuda program spawns large number of threads on the GPU that access different portions of the input dataset. The parallelism comes from the fact that a GPU has many Streaming Multiprocessors (SMP) and each thread is run in parallel with different thread_id that dictates the portion of dataset it will use. The code run by each thread is same, namely the *kernel function*. Thus it is a Data Parallel mode of parallelism.The Cuda architecture provides for the synchronisation and data sharing between two threads using *thread blocks*. Synchronization is implemented through breakpoints in the kernel code using *__syncthreads()* .Thread block owns a small memory which is shared by all the threads in the block. Cuda also provides a large *global memory* also known as the *device memory*. The data to be computed upon first needs to be copied to device memory using *cudaMemcpyHostToDevice()* and *cudaMemcpyDeviceToHost()*. Cuda also defines a *warp* that consists a group of thread that compute in lockstep, i.e., at each point of kernel these threads execute the same instruction but with different data. Programmer can specify the number of blocks and the number of threads per block while invoking the kernel using *angular bracket syntax* <<<blocks, threads>>> . The threads are distributed across multiple SMP . The distribution and management of resources for the thread is taken care of by the cuda execution environment itself. In general, multiple threads are grouped into blocks and multiple blocks are grouped into a grid. Block (or grid) can be atmost three dimensionsional where along each dimension we have constituent threads(or blocks). Having multiple dimensions is helpful in running kernels on Image processing algorithms. For our purposes we used only one dimension. The total number of blocks in a grid and threads in a block are limited by the architecture of the GPU. For this project Quadro K620 GPU was used that could run 1024 threads per block and 65536 blocks per grid.

CUDA provides synchronisation primitives for synchronisation between various threads in a block. The __syncthreads() causes all threads in a block to wait until every other thread in the block has reached the statement. The use of __syncthreads() is tricky because using it inside a branching statement may lead to deadlock. Synchronisation is needed when multiple threads have to read and write in the same block of memory. As expected, use of synchronisation increases the latency but it is unavoidable at least if the current algorithm is used.

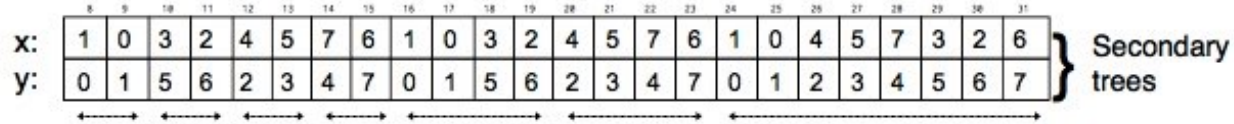
## Layered Tree Structure

Both the primary tree and secondary trees are implemented in a layered fashion to reduce the access latency of elements. Using pointers adds a level of indirection to accessing values which increases the latency and is impractical for implementation on CUDA. The primary tree is basically a list of points sorted by their x coordinate. Along similar lines, the secondary tree for a canonical node is a list of points in its canonical subset sorted by y coordinate. We maintain a single array for all the secondary trees that are appended one after the other.

For example let us consider,

P={ (0,1) , (1,0) , (2,6) , (3,5) , (4,2) , (5,3) , (6,7) , (7,4) }

The primary tree in our proposed structure for this point set will consist of (0,1) , (1,0) , (2,6) , (3,5) , (4,2) , (5,3) , (6,7) , (7,4) This is a list of points sorted in increasing x coordinate.The secondary trees will be :-




Each underlined block in the above diagram is a list of points in the canonical subset of internal nodes from bottom up (in level order) sorted on y coordinate. Each list is appended onto the previous list to give a single structure for the complete set of secondary trees. When construction is ordered from bottom to top it can be noted that the ith level's secondary trees can be created by merging the trees from (i+1)th level in groups of two.

The algorithm uses [9] to create the secondary tree and a parallel version of quicksort provided by CUDA. To use the full potential of GPU, parallel merge algorithm has been used that performs a binary search to divide the arrays to be merged in various threads. Then each thread works only with the part of array allocated to it.

# Algorithm

## BUILD-TREE

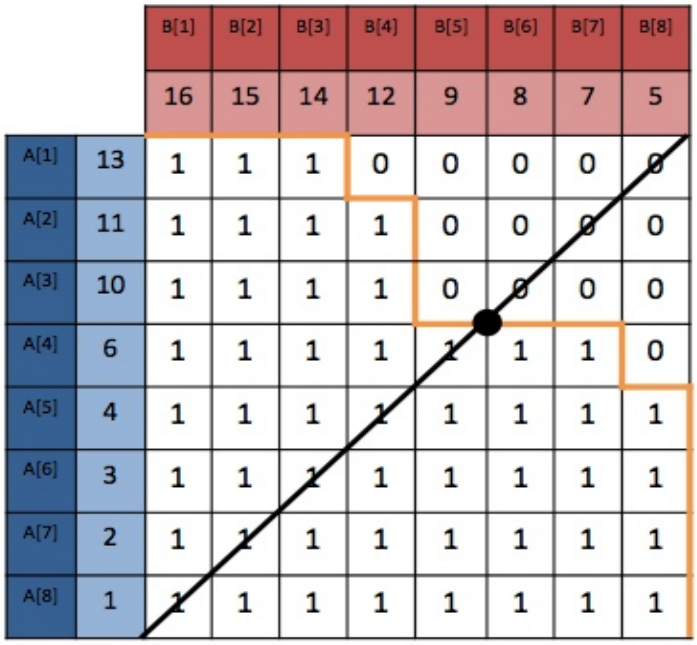
```
BuildTree(P):-
        Primary=Sort(P,key=P.x)
        Secondary=Primary
        buildSecondaryTree(Secondary,P.length,2,1)

buildSecondaryTree(Array,n,canonical_size,call):-
        if(canonical_size>=2*n):
                return
        else
                i=call*n
                while(i<(call*n+n)):
                        Array.append(merge_y(i,i+canonical_size/2,canonical_size/2))
                        i+=canonical_size
                buildSecondaryTree(Array,n,canonical_size*2,call+1)
```

P- list of points where P.x refers to x coordinate and P.y to the y coordinate of points.
Primary- Flattened representation of Primary Tree
Secondary- Flattened representation of Secondary Tree
Merge_y(x,y,z)- merge Array[x…x+z] and Array[y…y+z] to return a new array based on y coordinate

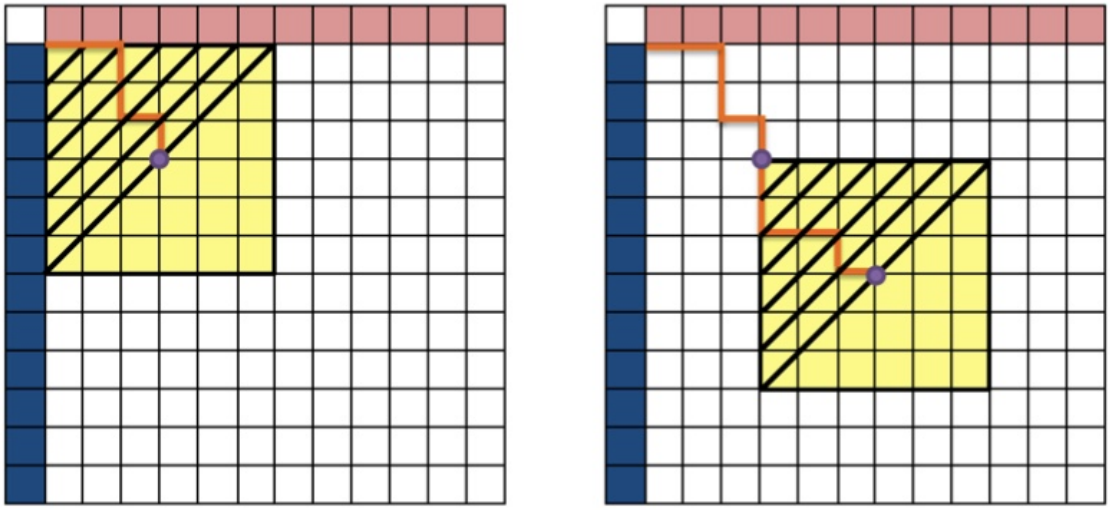
The while part of the code above is the main region that can be parallelised using GPU Merge Path technique. The code above makes n/2 + n/4 + n/8 +…+1 calls to merge_y procedure each of which can be run independently in a non blocking fashion. The only consideration is that before the next call to buildSeconaryTree via recursion the previous calls to merge_y must all be finished. This is done by using __syncthreads() primitive.

## GPU MERGE PATH

GPU Merge Path is an efficient parallel algorithm that merges two sorted arrays using multiple processors. The algorithm runs in two stages. First stage involves binary searching to divide the workload into multiple non overlapping segments. Pairs of such segment (one from each array) is then given to the processor for merging.




To understand how GPU Merge Path works, consider a matrix M(dimension m*n) formed from two arrays A and B with length m and n respectively where M(i,j)=A(i)<=A(j) (for arrays sorted in descending). The figure above shows such a matrix for the arrays appearing in red and blue as column and row headers. The line in green along the boundary of cells shows the order of selecting the next element to be put in the result array. When the line moves right at a cell , element from array B is selected and when it moves down element from A is selected. It can be observed that the line traces boundary between 1 and 0 in the matrix. This important observation leads us to the algorithm which basically does a binary search diagonally on the 1s and 0s to find this boundary point.Once we have a set of boundary points for all such diagonals, the arrays A and B can be segmented into non overlapping regions. These regions can be merged in parallel since they are independent of each other. For clarity consider the figure below-



Consider the first point in the fig on left. Let this be (x1,y1).This is a boundary point between ones and zeros along the diagonal line shown. Similarly the next point(x2,y2) is also a boundary point found using binary search. Now consider a rectangle with these two points as the top left and bottom right points resepectively. The rectangle is responsible for a portion of both A and B. More specifically A[x1..x2] and B[y1…y2].Also it can be inferred from the starting point that the position in final array where results are put will be x1+y1 to x2+y2-1. Thus we have successfully separated the problem into multiple disjoint array merge problems. These smaller problems can be run in parallel. This is the gist of GPU Merge technique that is used in the current implementation to merge arrays for creating Secondary Trees.

The only trouble that remains after understanding the algorithm is how to map the different subtasks to the programming platform CUDA provides(the threads and blocks). The details of this mapping are provided in the next section.

# Implementation

The algorithm was implemented on Quadro K620 GPU with 384 CUDA cores and 2GB on chip memory. The memory bandwidth of 29GB/s makes data transfer time insignificant. The serial version of the algorithm was implemented on Intel Xeon E5-4610 v4 with 16 cores and 32 GB RAM.

The algorithm consists of two stages. First is the construction of Primary Tree using sorting and next is constructing Secondary Trees using merging algorithm [9] described above.

**Constructing Primary Tree**

Primary tree is basically a sorted list of points according to x coordinate, as explained earlier. To sort the points a recursive version of quicksort is used. This algorithm is provided by the CUDA library. Each recursive call on one half of array is launched on a separate stream that makes the calls non blocking providing scope for parallelism. This technique is called dynamic parallelism and works only on CUDA architexture greater than 3.5. Faster algorithms for sorting are available but this was chosen because of the simplicity of the algorithm.

**Constructing Secondary Tree**

For each Secondary tree a separate block of threads is launched which takes care of one canonical node. Based on preliminary analysis it was found as a general rule that with increase in the number of threads in a block the execution time decreased. Keeping this in mind the current implementation is designed to launch same number of threads in a block as the number of elements in the array to be merged. If the canonical size of the current node goes above 1024 the program launches its maximum poissible 1024 threads. The threads first do a binary search each to find the portion of array they work on. Then they write their results in a globally shared array and wait for other threads to finish. For the ith thread its portion of data is uniquely determined by i and (i+1)th thread's results(Ignoring the corner cases). Once all threads have written into the array the second phase of merging the results begins. This is just a simple scan from beginning to end to find the next element to put in the result array.

Also prior to and after the Primary tree construction the points are transferred to and from CUDA via cudaMemcopy function calls. The data transfer time to and from the device is insignificant compared to the execution time of the algorithm.

## Results for GPU Merge

The bottleneck in the implementation is construction of primary tree. But the secondary tree construction gives a speedup of 3X as can be seen in the graph below.

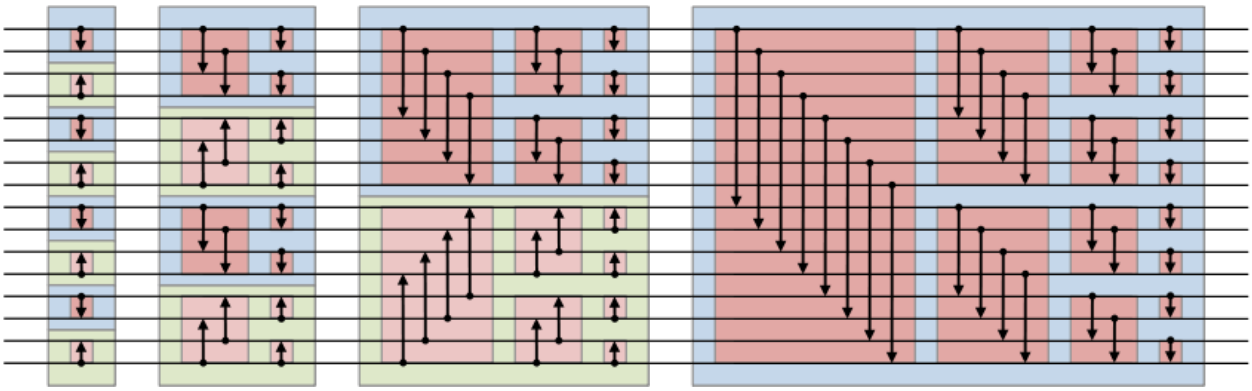

The graph above shows the execution time of buildSecondaryTree using the architectures specified in the previous section. The current implementation breaks at 2^16 elements.This is due to the number of blocks launched exceeding the device capacity. One solution to the problem is allocating more points to a thread. This over allocation takes a toll on the total time because each thread does sequential computation for each individual point.

## Sorting on the GPU-Bitonic Sorting

Bitonic sorting is a parallel sorting algorithm that converts a random sequence to a bitonic sequence and then sorts the bitonic sequence. A bitonic sequence is one which first increases then decreases or vice versa. More formally, as defined in [7]:

***Bitonic Sequence:*** A sequence of values A0,A1,A2….An-1 with the property that (1) there exists an index i, 0<=i<=n-1, such that A0 through Ai is monotonically increasing and Ai through An-1 is monotonically decreasing, or (2) there exists a cyclic shift of indices so that the first condition is satisfied.

On a Bitonic Sequence we do Bitonic Split to create two sequences, of half the original size, such that all the elements of first sequence are smaller than the second sequence. This process of Bitonic splitting is continued till the size of sequence obtained is greater than one. Doing so we obtain a sorted sequence. This procedure is named *Bitonic Merge* and its implementation is based on a comparator network called bitonic merging network. Here is an image[1] for the network.



In the figure above, the bitonic network sorts a sequence of size $2^n$. The figure shows wires connected by vertical connectors. The connectors act as comparator and swapper, ie, for every connector the smaller value comes at the tail of the arrow and greater at head of the arrow. The direction of the arrow depends on the half in which connector is located.

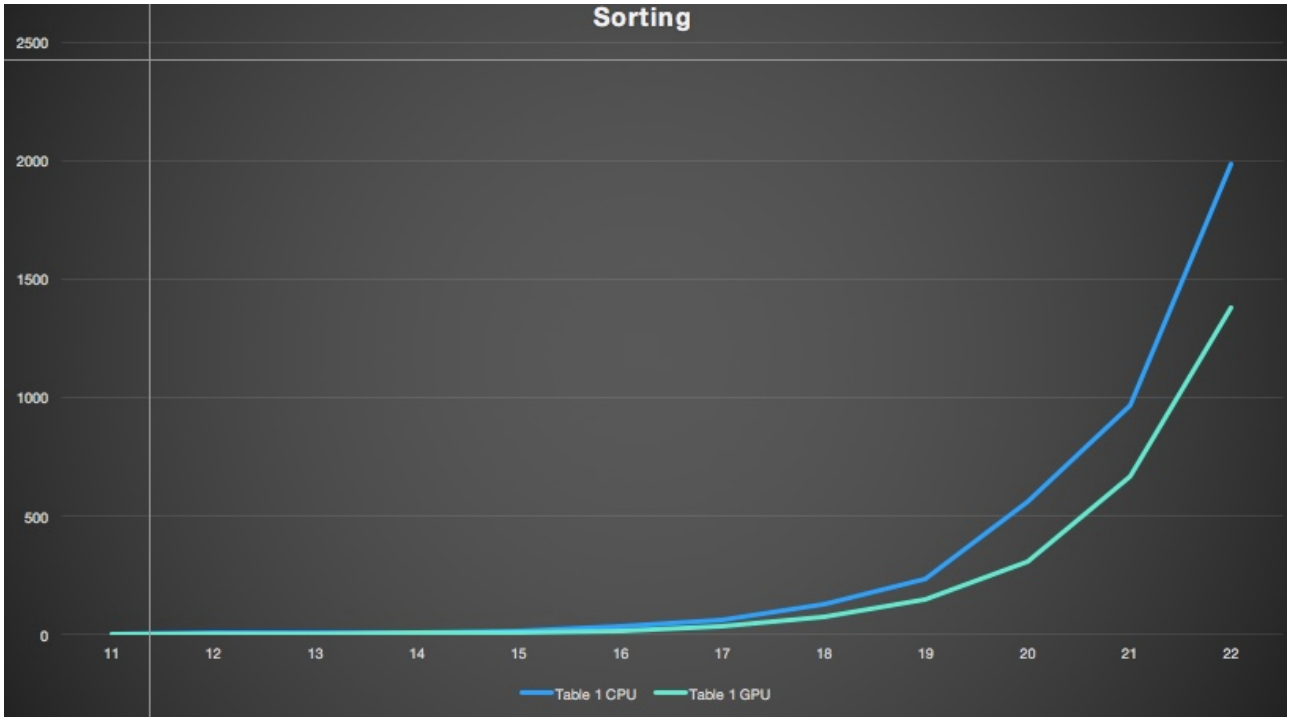
Batcher [8] showed that Bitonic Sorting can sort in O(log ^2 n) time using O(n) processors. Much work has been done in efficient implementations of Bitonic sort. We however use a very basic version of bitonic sorting that works with linear number of processors. The algorithm first converts a random sequence into bitonic sequence and then sorts the bitonic sequence as discussed above. The benefit of using comparator networks is that while sorting we don't have to pay attention to the actual value of data items that are to be sorted. That is , doing a fixed number of comparisons on fixed element indices as defined by the sorting network we can obtain the sorted sequence for any given distribution of data. This is particularly useful in a parallel setting where the communication between processors leads to an additional cost.

In the algorithm for construction of Range Tree, Bitonic Sort is employed to construct the primary tree. The primary tree is a sorted sequence of points according to their x coordinates.

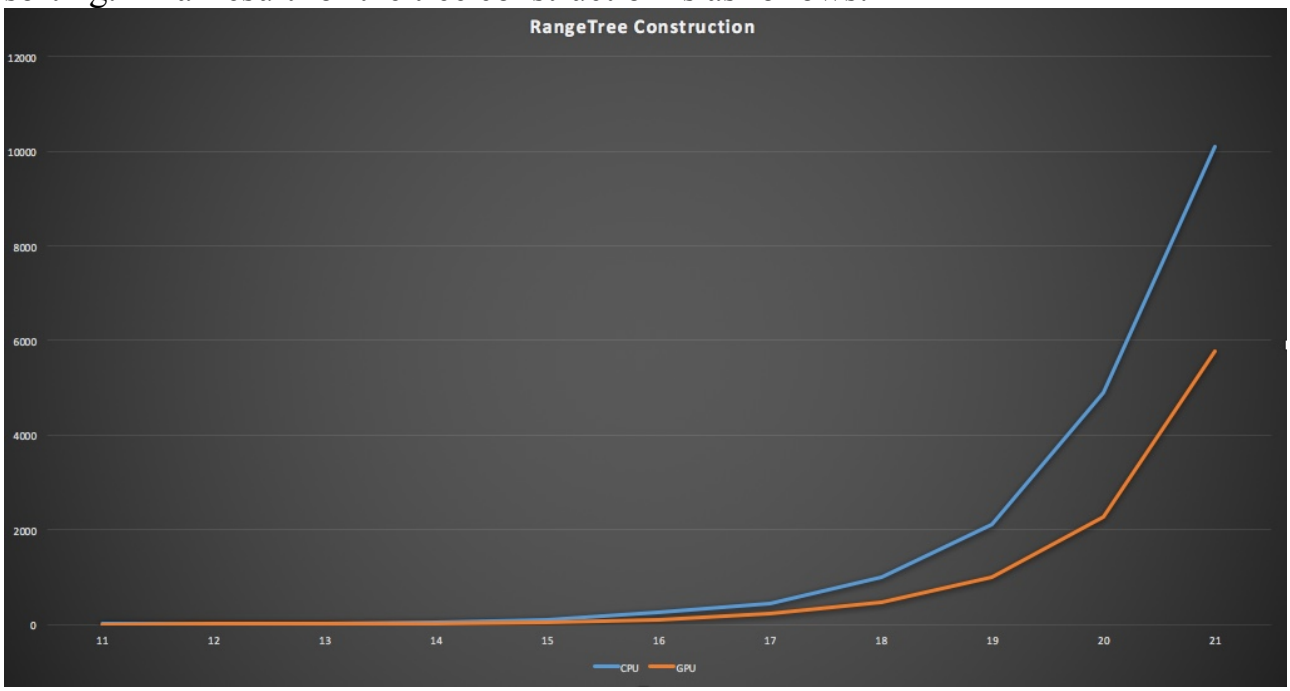
For the purpose of implementation every thread in a block was allocated exactly one value in the unsorted array. From initial experimentations it was found that the device allowed for optimal usage at 512 threads per block. Also the maximum number of blocks that can be launched is limited. Each stage of the algorithm is run in parallel (for all data items). Thus a total of (logN)^2 stages are run. These stages themselves are run sequentially, thereby lending the algorithm the runtime as stated above.

## Results and Conclusion

The graph below shows the comparison between primary tree construction using recursive sequential code and CUDA on a point set generated from a random distribution.




So, the Bitonic Sort network gave a speed up of 1.4-1.8 compared to sequential sorting. Final result for the tree construction is as follows:



This shows that GPU construction of Range Trees is valuable for use cases that have high frequency of Range Tree construction in a second. In such cases, GPU is a valuable tool to speed up the overall process by cutting down on each individual construction. This may be the case with video processing algorithms that have to process many frames per second and the location of points change in every frame.

# References


1. Jordan Fix, Andrew Wilkes, Kevin Skadron Accelerating Braided B+ Tree Searches on a GPU with CUDA In Proc. ISCA Workshops, 2011.
2. Jinwoong Kim, Sul-Gi Kim, Beomseok Nam Parallel multi-dimensional range query processing with R-trees on GPU J. Par. Dist. Comp., 73(8), 2013, 1195-1207.
3. Manoj Kumar Maramreddy and Kishore Kothapalli "GPU Accelerated Range Trees with Applications." Proceedings of Euro-Par 2014, pages 740-751, Porto, Portugal, 2014.
4. Jared Hoberock, Nathan Bell (2015) Thrust (Version 1.8.1) [Source Code]. https://thrust.github.io
5. Kelly, Maria and Alexander Breslow. "Quadtree Construction on the GPU: A Hybrid CPU-GPU Approach." (2010)
6. Foley, T., and Sugerman, J. Kd-tree acceleration structures for a gnu raytracer In Proc. Graphics hardware, pp: 15–22, 2005.
7. V. Kumar, A. Grama, A. Gupta, and G. Karypis. Introduction to Parallel Computing. Benjamin Cummings, 1994.
8. K. Batcher. Sorting Networks and their Applications. In Pro- ceedings of the AFIPS Spring Joint Computing Conference, volume 32, 1968.
9. Green et. al. "GPU Merge Path- A GPU Merging Algorithm."