Massively Parallel KD-tree Construction and Nearest Neighbor Search Algorithms

Linjia Hu, and Saeid Nooshabadi
Department of Computer Science
Michigan Tech, Houghton, MI, USA
Email:{linjiah,saeid}@mtu.edu

Majid Ahmadi
Department of Electrical and Computer Engineering
University of Windsor, Windsor, ON, Canada
Email:ahmadi@uwindsor.ca

***Abstract*—This paper presents parallel algorithms for the construction of k dimensional tree (KD-tree) and nearest neighbor search (NNS) on massively parallel architecture (MPA) of graphics processing unit (GPU). Unlike previous parallel algorithms for KD-tree, for the first time, our parallel algorithms integrate high dimensional KD-tree construction and NNS on an MPA platform. The proposed massively parallel algorithms are of comparable quality as traditional sequential counterparts on CPU, while achieve high speedup performance on both low and high dimensional KD-tree. Low dimensional KD-tree construction and NNS algorithms, presented in this paper, outperform their serial CPU counterparts by a factor of up to 24 and 218, respectively. For high dimensional KD-tree, the speedup factors are even higher, raising to 30 and 242, respectively. Our implementations will potentially benefit real time three-dimensional (3D) image registration and high dimensional descriptor matching.**

***Index Terms*—KD-tree, Parallel algorithm, GPU, CUDA, descriptor matchin, pattern recognition.**

I. Introduction

The nearest neighbor search (NNS) problem is an import research topic in image progressing, computer vision and computer graphics. The purpose of NNS is to search for the k closest points in a target set S for any point in query set Q. A brute force NNS directly compares the query point to all n points in the target set resulting in a high time complexity of $\mathrm{O}\big(n^2\big)$. However, KD-tree, a generalized binary tree [1], can decrease the time complexity using spatial data structures. For well distributed point sets, an efficient NNS algorithm in [2] improves the search complexity to $\mathrm{O}\big(log(n)\big)$ by using a balanced KD-tree. In recent years, massively parallel architecture (MPA) of graphical processing unit (GPU) has been employed to accelerate KD-tree construction or NNS in [3] [4] [5]. The work in [3] implemented a brute force NNS on GPU with speedup factor between 1 to 100 comparing to the equivalent MATLAB implementation. The work in [4] built a KD-tree on GPU with a splitting metric that combines empty space splitting and median splitting. The speedup is between 9 and 13 with respect to serial counterpart. The work in [5] developed an all-NNS on GPU based on KD-tree to solve the three-dimensional (3D) registration problem. The KD-tree was built on CPU and then transferred to GPU before performing NNS. The registration speedup compared the counterpart on CPU can reach 88. In all these previous works, the implementations are limited to three dimensional KD-tree, and not scalable to higher dimensional descriptor matching. Also, these works only focus on individual KD-tree construction or NNS and give no consideration to their joint integration. In this work, we have developed massively parallel algorithms for KD-tree construction and NNS at almost each stage on GPU with high speedup. Moreover, our implementations are also scalable to high dimensions. These work can significantly accelerate 3D image registration with low dimensional KD-tree and 3D descriptor matching with high dimensional KD-tree [5] [6] [7].

II. KD-tree Construction on GPU

To facilitate the development, we have used the parallel algorithms and data structures in the *Thrust* library [8]. Its use allows for implementation of high performance parallel applications with minimal programming effort through a high-level interface that is fully interoperable with CUDA. We refer the serial KD-tree construction and NNS algorithms in PCL [9] to design and validate our parallel algorithms.

Our KD-tree construction algorithm employs breadth first search (BFS) to fully exploit the fine-grained parallelism of GPU and its streaming multiprocessor architecture in all stages of construction. The quality of the proposed technique for building the KD-tree is comparable to those constructed on CPU. In our implementation at each BFS step, every node with the same distance from the tree root spawns a new CUDA thread, with number of threads doubling from the preceding step. The details of our implementation is shown in Algorithm 1. With data staged properly on the GPU memory, point splits on different nodes can begin. The split operation on the GPU is implemented using the parallel reduction kernels. One thread is responsible for one node split. The number of nodes and the threads involved in the split doubles at each iteration. There are two major steps in each split iteration. The first step as shown in Algorithm 2 concentrates on computing the indices of the parent and and children of the splitting node, as well as the split value and dimension. The second major step as show in Algorithm 2 focuses on the re-distribution of points after all the splitting related information for children and parent are gathered in the data structure. We launch n threads to process

n groups of k-dimensional points in the descending order of dimensional values.

Algorithm 1 Construct KD-tree on GPU

```
Input:  $k$ dimensional points (descriptor)
Output: KD-tree on GPU
procedure KD-TREE-CONSTRUCTION-GPU
    $n \leftarrow$ number of points;
    $m \leftarrow n$/number of points in one leaf;
    allocate child/parent/split/bounding box indices for $m$ node on GPU;
    for $i = 0$ to $n$ do
        allocate temporary indices for point $i$, its owner and backup points for it;
    end for
    for $i = 0$ to $k$ do
        allocate indices for all points/corresponding owner/left and right masks on $i$ dimension;
    end for
    assign indices for $n$ points from 0 to $n - 1$;
    for $i = 0$ to $k$ do
        sort all the $n$ value at dimension $i$;
    end for
    compute bounding box for root node according to the minimal and maximal values at each dimension;
    while true do
        Node-Split-Points-Movement;
    end while
end procedure
```

Algorithm 2 Split Node and Move Points to Children

```
Input:  indices of points in $k$ dimensions
Output: parent node, child nodes, split information
procedure NODE-SPLIT-POINTS-MOVEMENT
    if ($enough\_space$ and $split\_enable$) then
        compute left/right node indices;
        update split information for left/right nodes;
        update parent and children indices;
    end if
    if (no more node to be split) then
        break;
    end if
    [GPU kernel] launch $n$ threads kernel on GPU to compute split
        information for left/right children. Each thread process $k$
        dimension split information for one or more than one point
        with the sorted indices. Finally, each point at each dimension
        has split information at left/right children of current node;
    synchronization;
    relocate points in left/right children to make points in a node continuous;
    compute bounding box for left/right children;
end procedure
```

To better understand the KD-tree construction on GPU, we demonstrate the procedure through an example with ten points in two dimensions (x and y). The layout of the 10 points is shown in Figure 1.(a). The first split is performed along x dimension and the split value is the median of maximum and minimum value of those points on the x dimension. The output is shown in Figure 1.(b)

As show in Figure 2, points are sequenced from 0 to 10. At the start as shown in Table a, we assign three sequences for the points. In Table b, values of points are sorted in ascending order in x and y dimensions. After sorting the point values and updating sequences in two dimensions, the bounding box

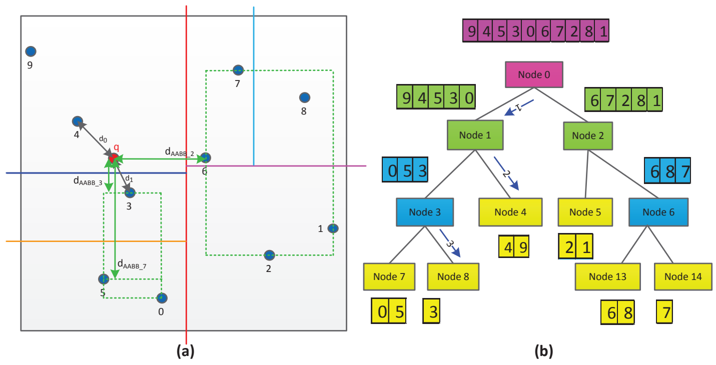

Fig. 1: Two-dimensional KD-tree partitions and NNS

is formed through the minimum and maximum values in two dimensions from Table b.

From the bounding box in Table c, we compute split value in the x dimension from $(p[9].x + p[1].x)/2$. In Table d, the split dimension and value are updated. The information for the left and right children of the current node (root) are renewed to 1 and 2. The parent information is updated as 0. In Table e, we assign one thread to a column from Table b, containing one point from the ordered lists in two dimensions. Each thread checks the points belonging to either left or right child in each of the two dimensions.

The owners and left and right marks in each dimension, are updated and stored in Table f. For example, the first element in owner array in x dimension is recorded as 1, which indicates point 9 sorted in the x dimension belongs to the node 1. Similarly, the first element in $left\_right\_mark(x)$ is marked as 0, a correspondence to the left node (node 1). By the same token looking at the sorted list in the y dimension, point 2 belongs to node 2, and its corresponding $left\_right\_mark(y)$ is set to 1. Tables g and h show the sequencing after the split.

The first and second halves of $new\ index(x)$ are the sub-sequences in x dimension for the left and right children after the splitting. $left\ counter$ is the exclusive scan result from $left\_right\_mark\ (x)$. The last element in $left\ counter$ correspond to the number of points in the left child. Table i lists the indices of new owners after split. The computation of these indices is similar to that of the new sub-sequencing in the x dimension.

Tables j and k demonstrate the computation of the new sequences in the y dimension. Tables l to n present the computation of the bounding boxes for the left and right children of current node. From the $new\_onwer\ (x)$ listed in Table l, we can count the number of nodes involved in split through a scan operation. Based on these results we launch $(number\ of\ lables)$ threads to compute the bounding boxes of left and right children. Since the nodes are already sorted in each dimension, we can calculate the bounding box through the first and last elements in each dimension for the left and right children. The procedures in Tables a to n correspond to the first iteration of split. If any more nodes are left, further

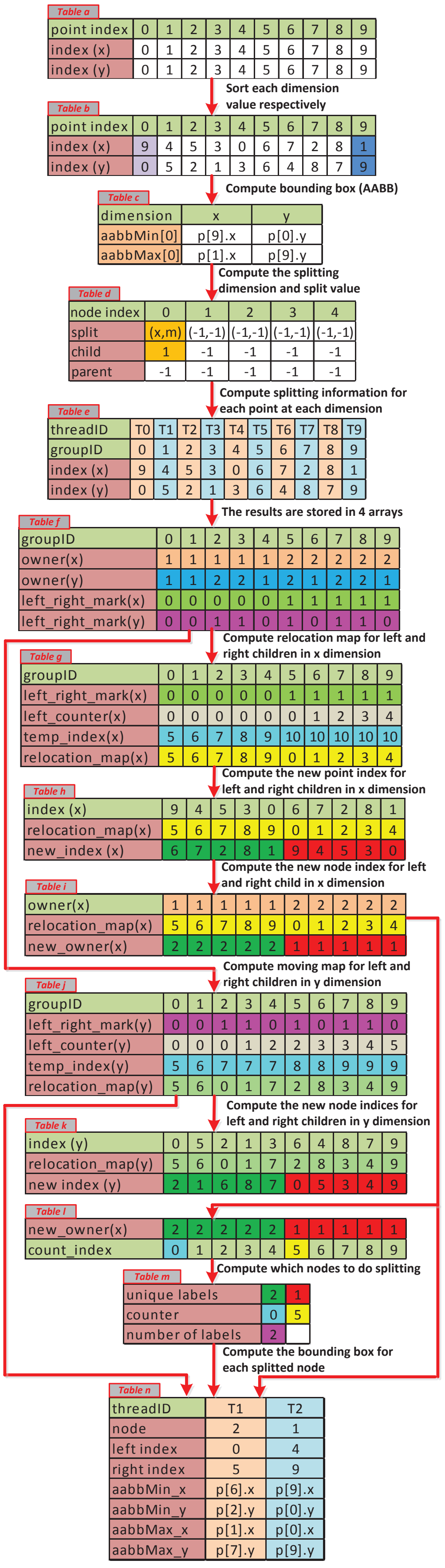
Sort each dimension value respectively
Compute bounding box (AABB)
Compute the splitting dimension and split value
Compute splitting information for each point at each dimension
The results are stored in 4 arrays
Compute relocation map for left and right children in x dimension
Compute the new point index for left and right children in x dimension
Compute the new node index for left and right child in x dimension
Compute moving map for left and right children in y dimension
Compute the new node indices for left and right children in y dimension
Compute which nodes to do splitting
Compute the bounding box for each splitted node

Table a

point index	0	1	2	3	4	5	6	7	8	9
index (x)	0	1	2	3	4	5	6	7	8	9
index (y)	0	1	2	3	4	5	6	7	8	9

Table b

point index	0	1	2	3	4	5	6	7	8	9
index (x)	9	4	5	3	0	6	7	2	8	1
index (y)	0	5	2	1	3	6	4	8	7	9

Table c

dimension	x	y
aabbMin[0]	p[9].x	p[0].y
aabbMax[0]	p[1].x	p[9].y

Table d

node index	0	1	2	3	4
split	(x,m)	(-1,-1)	(-1,-1)	(-1,-1)	(-1,-1)
child	1	-1	-1	-1	-1
parent	-1	-1	-1	-1	-1

Table e

threadID	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9
groupID	0	1	2	3	4	5	6	7	8	9
index (x)	9	4	5	3	0	6	7	2	8	1
index (y)	0	5	2	1	3	6	4	8	7	9

Table f

groupID	0	1	2	3	4	5	6	7	8	9
owner(x)	1	1	1	1	1	2	2	2	2	2
owner(y)	1	1	2	2	1	2	1	2	2	1
left_right_mark(x)	0	0	0	0	0	1	1	1	1	1
left_right_mark(y)	0	0	1	1	0	1	0	1	1	0

Table g

groupID	0	1	2	3	4	5	6	7	8	9
left_right_mark(x)	0	0	0	0	0	1	1	1	1	1
left_counter(x)	0	0	0	0	0	0	1	2	3	4
temp_index(x)	5	6	7	8	9	10	10	10	10	10
relocation_map(x)	5	6	7	8	9	0	1	2	3	4

Table h

index (x)	9	4	5	3	0	6	7	2	8	1
relocation_map(x)	5	6	7	8	9	0	1	2	3	4
new_index (x)	6	7	2	8	1	9	4	5	3	0

Table i

owner(x)	1	1	1	1	1	2	2	2	2	2
relocation_map(x)	5	6	7	8	9	0	1	2	3	4
new_owner(x)	2	2	2	2	2	1	1	1	1	1

Table j

groupID	0	1	2	3	4	5	6	7	8	9
left_right_mark(y)	0	0	1	1	0	1	0	1	1	0
left_counter(y)	0	0	0	1	2	2	3	3	4	5
temp_index(y)	5	6	7	7	7	8	8	9	9	9
relocation_map(y)	5	6	0	1	7	2	8	3	4	9

Table k

index (y)	0	5	2	1	3	6	4	8	7	9
relocation_map(y)	5	6	0	1	7	2	8	3	4	9
new index (y)	2	1	6	8	7	0	5	3	4	9

Table l

new_owner(x)	2	2	2	2	2	1	1	1	1	1
count_index	0	1	2	3	4	5	6	7	8	9

Table m

unique labels	2	1
counter	0	5
number of labels	2	

Table n

threadID	T1	T2
node	2	1
left index	0	4
right index	5	9
aabbMin_x	p[6].x	p[9].x
aabbMin_y	p[2].y	p[0].y
aabbMax_x	p[1].x	p[0].x
aabbMax_y	p[7].y	p[9].y

Fig. 2: Array based KD-tree construction on GPU

iterations will performed. In this example, the splits stop on the third iteration.

III. NNS on GPU

We search for the nearest neighbor using the depth first search (DFS) by traversing down the tree until reaching the leaf node of interest. Next we backtrack the tree to search for the other possible better candidate points in the neighboring nodes. Since recursion is not possible in the CUDA programming environment, the traditional KD-tree DFS search cannot work directly. Instead, we perform the search iteratively, using the priority queues. During the KD-tree construction, the minimum and maximum values at each dimension are used to determine the axes aligned bounding box (AABB). In each dimension, if the value of the query point is less than minimum value or greater than the maximum value, there exists an orthogonal distance between query point and the AABB at that dimension.

The square distance from query point to the AABB is accumulation of the square of orthogonal distance at each dimension. Since NNS for the points in the query set Q are independent of each other, we can associate each search to a thread on GPU. The detail of the implementation is shown in Algorithm 3. Backtracking start node is extracted from priority queue Q_p. This node has the minimum distance from the query point to its bounding box. A new entry is inserted into Q_p at with each branch during the DFS descending towards the leaf node. When search arrives at leaf node, the best nearest neighbor point candidate is computed. The entry with current node is removed and the entries with distance great than the best smallest distance are also removed from Q_p. The search algorithm terminates when the priority queue is empty or the backtracking reaches the root node. As shown in Figure 1.(b), first, the search traverses down to leaf node 4. At each branch in the internal nodes, the indices of traversed nodes and the distances are inserted to priority queue. At node 4, we compute the distance between the red query point and each point in node 4. The best candidate is point 4, and the smallest distance, denoted as d_0, is between red point and point 4. The entries with distance less than d_0 are removed. As mentioned before, point 4 may not be the nearest neighbor of red point. So, we extract best backtrack node from the priority queue. The first backtrack node is node 3 with the highest priority. Then, the search descends to leaf node 8, where point 3 with d_1 is found to be the best. Since $d_1 < d_0$, the smallest distance is updated to d_1 and the best candidate is updated as node 3. Both the distances of the entries in the priority queue are great than d_1, so they will be removed. Finally, the priority queue is empty, and the search terminates.

IV. Experiment Results

We validated our GPU accelerated KD-tree construction and NNS algorithms through extensive experimentation on a platform with Intel $i7-960$ CPU and GeForce $GTX660$ GPU.

Algorithm 3 NNS on KD-tree

```
Input:   parent, child, splits, aabbMin, aabbMax, query,
Output: elements, result, distance
procedure SEARCH-NN-POINTS
    allocate global memory for $m$ query points on GPU;
    copy the points from CPU to GPU;
    launch $m$ threads for $m$ query points to run the below
     kernel on GPU;
    [NNS Kernel on GPU]
    $backtrack \leftarrow false$;
    $lastnode \leftarrow -1$;
    $currentnode \leftarrow 0$;
    create a array based priority queue $Q_p$;
    while $true$ do
        if $currentnode == -1 \parallel sizeof(Q_p) \leq 40$ then
            $break$;
        end if
        $split \leftarrow splits[currentnode]$;
        $delta \leftarrow query[i].[split\_dim\_val] - split.split\_val$;
        if $delta < 0$ then
            $otherchild$++;
        else
            $bestchild$++;
        end if
        if (!$backtrack$) then
            if $leftchild == -1$ then
                compute the distances and update $restults[i]$ and $Q_p$;
                $backtrack \leftarrow true$;
                $lastnode \leftarrow current$;
                $current \leftarrow$ node index of $dequeue(Q_p)$;
            else
                $lastnode \leftarrow currentnode$;
                $current \leftarrow bestchild$;
                compute the distance update $Q_p$;
            end if
        else
            $mindist \leftarrow$ compute the distance;
            if $(lastnode == bestchild)$ and
              $(mindist \leq restults[i].worstdist)$ then
                $lastnode \leftarrow currentnode$;
                $currentnode \leftarrow otherchild$;
                $backtrack \leftarrow false$;
            else
                $lastnode \leftarrow currentnode$;
                $currentnode \leftarrow$ node index of $dequeue(Q_p)$;
            end if
        end if
    end while
end procedure
```

The points in the data sets are synthetic randomized data and the query sets are the same as the data set for KD-tree construction. We randomly generate target points sets S with n points in a sequence of six tests, where n = 3200, 6400, 12800, 25600, 51200 and 102400, respectively, for both low and high dimensional KD-tree. The number of nearest neighbors (k) of each queried point is fixed as 4. In low and high dimensional KD-tree experiment, we set the number of dimensions d to 3, and 512 respectively.

Table I present the low dimensional KD-tree construction and NNS runtimes and speedups. Listed in the table are runtimes for serial KD-tree construction (T_{ccnst}) and NNS (T_{cnns}) on CPU, and our parallel KD-tree construction (T_{gcnst}) and NNS (T_{gnns}) on GPU for different **point** set sizes. S_{cnst} and S_{nns} are the speedup factors for our parallel construction and search algorithms on GPU with respect to the serial counterparts on CPU. From this table, we can see the speedup of KD-tree construction algorithm on GPU is up to 24.2, while the speedup of NNS on KD-tree can reach up to 218. Table II shows the high dimensional KD-tree construction and search speedup. **The maximum construction speedup can reach** 30.5 **while that of search can raise to** 242.1**.**

TABLE I: Low dimensional KD-tree runtime and speedup

number of point	CPU		GPU		Speedup	
	T_{ccnst}	T_{cnns}	T_{gcnst}	T_{gnns}	S_{cnst}	S_{nns}
3200	12.6	140.9	4.6	1.4	2.7	57.7
6400	25.4	292.2	7.2	2.5	3.5	118.3
12800	53.6	616.7	8.2	4.7	6.5	130.3
25600	111.3	1291.7	8.8	8.0	12.7	161.7
51200	232.9	2894.2	12.4	15.3	18.8	188.8
102400	488.8	6615.4	20.2	30.3	24.2	218.0

TABLE II: High dimensional KD-tree runtime and speedup

number of point	CPU		GPU		Speedup	
	T_{ccnst}	T_{cnns}	T_{gcnst}	T_{gnns}	S_{cnst}	S_{nns}
3200	818.2	428.7	247.9	3.9	3.3	109.9
6400	1640.7	893.8	315.5	6.7	5.2	133.4
12800	3286.9	1864.8	405.8	13.9	8.1	134.2
25600	6595.8	3902.5	519.4	20.8	12.7	187.6
51200	13225.2	8221.7	760.1	41.1	17.4	200.0
102400	26727.6	17621.8	876.3	72.8	30.5	242.1

V. Conclusion

This paper presented the design of parallel KD-tree construction and NNS on GPU for low and high dimensional spaces. The proposed algorithms are of comparable quality as traditional sequential counterparts on CPU, while achieve high speedup performance. Our implementations will benefit real time 3D image registration using low dimensional KD-tree and descriptors matching employing high dimensional KD-tree.

References

[1] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, September 1975.

[2] T. Rozen, K. Boryczko, and W. Alda, "GPU bucket sort algorithm with applications to nearest-neighbor search," in *International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, Plzen, Czech Republic, Febuary 2008.

[3] V Garcia, E. Debreuve, and M. Barlaud, "Fast K nearest neighbor search using GPU," in *Computer Vision and Pattern Recognition Workshops (CVPRW)*, Anchorage, AK, June 2008, pp. 1–6.

[4] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time kd-tree construction on graphics hardware," *ACM Transactions on Graphics*, vol. 27, no. 5, pp. 126:1–126:11, December 2008.

[5] D. Qiu, S. May, and A. Nuchter, "GPU-accelerated nearest neighbor search for 3D registration," in *International Conference on Computer Vision Systems (ICCVS)*, Liege, Belgium, October 2009, pp. 194–203.

[6] D.G. Lowe, "Distinctive image features from scale-invariant keypoint," *Discrete and Computational Geometry*, vol. 60, no. 2, pp. 91–110, 2004.

[7] F. Tombari, S. Salti, and L. Di Stefano, "Unique signatures of histograms for local surface description," in *European Conference on Computer Vision (ECCV)*, Hersonissos, Greece, September 2010.

[8] NVIDIA, "An introduction to thrust," https://developer.nvidia.com/Thrus, Accessed: 2015-01-23.

[9] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 2011.