Semi-Convex Hull Tree: Fast Nearest Neighbor Queries for Large Scale Data on GPUs

Conference Paper · November 2018

DOI: 10.1109/ICDM.2018.00110

Yewang Chen
National Huaqiao University
25 PUBLICATIONS **166** CITATIONS

Nizar Bouguila
Concordia University Montreal
325 PUBLICATIONS **3,394** CITATIONS

Bineng Zhong
National Huaqiao University
115 PUBLICATIONS **1,274** CITATIONS

Fei Wu
Shanghai Maritime University
194 PUBLICATIONS **2,406** CITATIONS

Semi-Convex Hull Tree: Fast Nearest Neighbor Queries for Large Scale Data on GPUs

Yewang Chen[1], Lida Zhou[1], Nizar Bouguila[2], Bineng Zhong[1],
Fei Wu[3], Zhen Lei[4], Cheng Wang[1] Jixiang Du[1], Hailin Li[5]
[1]College of Computer Science and Technology Huaqiao University, China, 361021, ywchen@hqu.edu.cn
[2]Engineering and Computer Science, Concordia University,Canada, H3G 2W1. nizar.bouguila@concordia.ca
[3]College of Computer Science and Technology, Zhejiang University, China, 310058, wufei@cs.zju.edu.cn
[4]CBSR&NLPR, Institute of Automation, Chinese Academy of Sciences, Beijing, 100190,China, zhen.lei@nlpr.ia.ac.cn
[5]College of Business Administration, Huaqiao University, Quanzhou 362021, China

***Abstract*—A fast exact nearest neighbor search algorithm over large scale data is proposed based on semi-convex hull tree, where each node represents a semi-convex hull, which is made of a set of hyper planes. When performing the task of nearest neighbor queries, unnecessary distance computations can be greatly reduced by quadratic programming. GPUs are also used to accelerate the query process. Experiments conducted on both Intel(R) HD Graphics 4400 and Nvidia Geforce GTX1050 TI, as well as theoretical analysis show that the proposed algorithm yields significant improvements and outperforms current k-d tree based nearest neighbor query algorithms and others.**

I. Introduction

Nearest neighbors (NN) query is a fundamental problem in computational geometry [13] and machine learning [15]. It plays an important role in various applications, including data mining, document retrieval and data compression etc, in which searching for the most similar matches is the most computationally expensive part. For example, the key of DBSCAN [7] is to retrieve core points whose first $MinPts$ nearest neighbors are all within a specified ϵ-neighborhood, which is obviously a NN problem, and DCore [6] is similar.

There are some techniques used to boost neighbor query, such as partition trees, graph methods[14] and hashing techniques (e.g., ANN [18], LSH [1], FLANN [11]) etc. Among them, partition trees are popular techniques that are used to recursively split the search space into subspaces, where K-d tree [3] is the most famous partition tree and widely used in many applications. It has various variants, such as buffer k-d trees [9], optimized k-d trees [16] and FRS[5]. However, it is not suitable to deal with high-dimensional data, since a general rule $n >> 2^d$ should be satisfied [8] in k-d tree, where d is the dimensionality and n is the cardinality. In addition to k-d tree, there are various other partitioning trees, such as cover trees [4] , vocabulary tree [12] and ball-tree [10].

In this paper, a fast parallel algorithm for nearest neighbor search over large scale data is proposed based on semi-convex hull tree, and we mainly focus on comparing our algorithm with buffer k-d tree and cover tree. Before starting, notations used in this paper are listed as following:

Notations: Data points set is noted as $P \subset \mathbb{R}^d$, where d is the dimension; n is the cardinality; p_i is the i^{th} point; $dist(q, p)$ is the distance between point p and q; $node_i$ is the i^{th} node, and $node_0$ is the root. In fact, a node represents a convex set H, and we use H_i as the convex set of $node_i$.

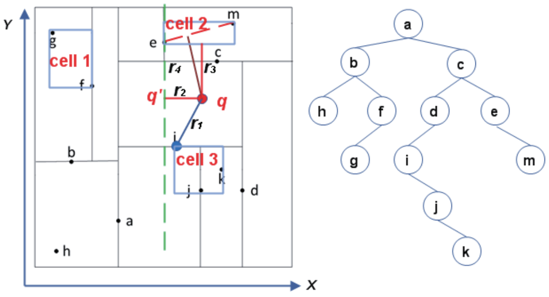
II. The drawbacks of k-d tree and our ideas

Fig. 1 shows an example of the subdivision and structure of a k-d tree. Each point is a node in k-d tree, and there exists a minimum hyper-rectangle, which is called a cell, that covers the point and all its descendants. For example, as shown in Fig. 1, *cell* 1 is the cell of node f, which is a hyper-rectangle that covers f and g. The cell of node i is *cell* 3 that covers i, j and k.

During the task of NN search based on k-d tree, we have to compare the current nearest distance with that of between query point and splitting hyper plane. For example, q (the red point) is a query point in Fig. 1, suppose point i is current nearest neighbor, and $dist(q, i) = r_1$, in order to check whether there are closer neighbor points within node e, two steps will be performed: (1) get the approximate min distance r_2 from q to the cell of e, in fact $r_2 = d_{q,q'}$, which is the distance from q to the splitting hyper plane (the green dash line) of node e, where q' is the projection of q on the hyper plane. (2) Check e and its sub nodes if $r_1 <= r_2$.

However, we notice that each splitting hyper plane is axis-parallel, and the approximate distance is simplified as the distance from query point q to the splitting axis, which makes it always far less than the distance from q to the real nearest neighbor, especially in high dimension.

Our ideas: (1) As mentioned above, the disadvantage of k-d tree is mainly caused by the axis-parallel hyper plane. Our first is to divide data set into hierarchical nodes, each of which is a convex set H, made by a set of non-axis-parallel hyper planes (essentially a set of linear inequations). Since the distance metric is usually convex, such as $\|\cdot\|_2, \|\cdot\|_1$ and $\|\cdot\|_\infty$ etc (in this paper, we only use $\|\cdot\|_2$), and each convex set H is made of a set of linear inequations. Then, quadratic programming is applicable to retrieve the minimum distance from a query point q to H, which is used as the lower bound from q to all points in the node. Since the aim is to filter unnecessary computations, it is obvious that the smaller the convex set of each node, the better lower bound we may obtain. Although, convex hull is the minimum convex



(a) subdivision (b) structure

Fig. 1. The subdivision and structure of a k-d tree. The 3 rectangles are the cell of node f, e and i, respectively. The green dash line is the splitting hyper plane of node e. The red dash line segment is the convex hull that covers point e and m. Point q is a query point, r_1 is the distance from q to point i.

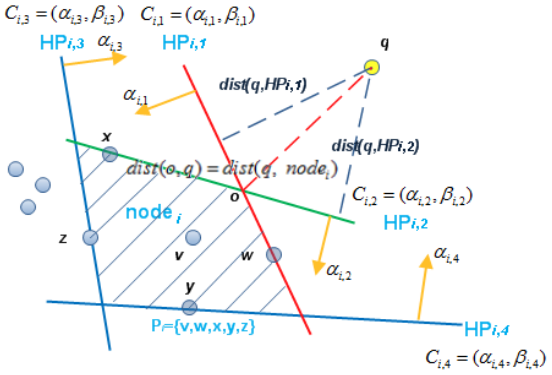

$C_{i,3}=(\alpha_{i,3},\beta_{i,3})$
$C_{i,1}=(\alpha_{i,1},\beta_{i,1})$
HPi,3
$\alpha_{i,3}$
HPi,1
q
dist(q,HPi,1)
$\alpha_{i,1}$
x
dist(q,HPi,2)
dist(o,q) = dist(q, node_i)
o
$C_{i,2}=(\alpha_{i,2},\beta_{i,2})$
HPi,2
node i
z
v
w
$\alpha_{i,2}$
$\alpha_{i,4}$
y
Pi={v,w,x,y,z}
HPi,4
$C_{i,4}=(\alpha_{i,4},\beta_{i,4})$

Fig. 2. $Node_i$ is made of 4 constraints $(C_{i,1},...,C_{i,4})$, and $HP_{i,1},...,HP_{i,4}$ are the four constraint hyper planes; $P_i=\{v,w,x,y,z\}$ is the data point set within $node_i$; the shadow region is one semi-convex hull of $Node_i$; $dist(q,HP_{i,1})$ and $dist(q,HP_{i,2})$ are the distance from q to $HP_{i,1}$ and $HP_{i,2}$, respectively; $dist(o,node_i)$ is the distance from q to $node_i$.

set that covers all points within a node, it is nontrivial to build a convex hull [2]. Therefore, semi-convex hull is optional for the proposed method. (2) The second is to use GPUs to accelerate distance computations which can not be filtered, because all distance computations are independent.

III. The Semi-Convex Hull Tree

Definition 1. *Linear Constraint: $C_{i,j}=(\alpha_{i,j},\beta_{i,j})$ is the j^{th} linear constraint on $node_i$ where $\|\alpha_{i,j}\|=1$ s.t. $\forall x\in H_i$,$\alpha'_{i,j}*x\geq\beta_{i,j}$.*

Obviously linear constraint is in fact a half space. Suppose $node_i$ has linear constraints $(C_{i,1},C_{i,2},...)$, then $node_i$ represents a convex set, e.g., the shadow region in Fig. 2 is the convex set H_i of $node_i$, which is actually a **semi-convex hull**.

We say a point p is within $node_i$ or $p\in H_i$ if $\forall j$ $s.t.$ $\alpha'_{i,j}*p\geq\beta_{i,j}$; we also note $P_i=\{p|p\in P\bigwedge p\in H_i\}$ is the data point set within $node_i$, obviously $P_i\subset H_i$. In Fig. 2, $P_i=\{v,w,x,y,z\}$ is also within $node_i$.

Definition 2. *Constraint Hyper Plane: $\{x|\alpha'_{i,j}*x=\beta_{i,j}\}$ is the Constraint Hyper Plane of $C_{i,j}$, and noted as $HP_{i,j}$.*

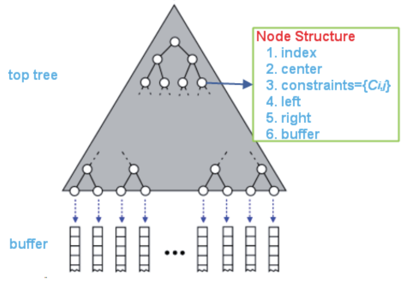


Fig. 3. Semi-convex hull tree has two main parts: top tree and buffers (this figure is revised from Figure. 1 in [9]). Each node has three kinds of basic information: hierarchical location (index, left and right), constraints and node center. Leaf node has additional buffer which is a set of data points.

In Fig. 2, $HP_{i,1},HP_{i,2},HP_{i,3}$ and $HP_{i,4}$ are the four constraint hyper planes of $C_{i,1},C_{i,2},C_{i,3}$ and $C_{i,4}$, respectively. Then, we also note $dist(HP_{i,j},q)=|\alpha'_{i,j}*x-\beta_{i,j}|$ as the distance from q to $HP_{i,j}$.

Definition 3. *Active Constraint: we say $C_{i,j}$ is active w.r.t q and $node_i$ if $\alpha'_{i,j}*q<\beta_{i,j}$.*

In Fig. 2, $C_{i,1}$ and $C_{i,2}$ are both active constraints w.r.t q and $node_i$. According to this definition, $dist(HP_{i,j},q)=\beta_{i,j}-\alpha'_{i,j}*x$ if $C_{i,j}$ is active w.r.t q and $node_i$, otherwise, $dist(HP_{i,j},q)=\alpha'_{i,j}*x-\beta_{i,j}$. Hence, obviously, for a query point q, there is at least one active constraint w.r.t q and $node_i$, if q is not within $node_i$,.

Definition 4. *Distance from q to a node: we note $dist(node_i,q)$ or $dist(q,node_i)=\min_x(dist(x,q))$ s.t. $x\in H_i$ as the distance from q to $node_i$.*

In Fig. 2, $dist(q,node_i)=dist(o,q)$ is the distance from q to $node_i$, where $o=arg\min_x(dist(x,q))$ s.t. $x\in H_i$.

Data structure: As shown in Fig. 3 semi-convex hull tree has similar structure as buffer k-d tree, i.e., it has top tree and buffers, buffers are a set of data points saved in leaf nodes. Each node has some information, such as hierarchical location (index, left and right), constraints and node center.

Building semi-convex hull tree: As mentioned above, we use non-axis-parallel hyper planes to divide data points within a node by two main steps:

(1) Determine a new hyper plane: Suppose Fig. 4 shows the points distribution of P_i, we divide it as following:

(a) Select yellow point o randomly, find the farthest point p from o, and then find the farthest point q from p, s.t. $p,q\in P_i$.

(b) Determine the hyper plane (red dash line) whose norm vector is $(p-q)/\|(p-q)\|_2$ and passes $(p+q)/2$. Suppose the hyper plane equation is $\frac{(p-q)'*x}{\|(p-q)\|_2}+b=0$, then $\frac{(p-q)'}{\|(p-q)\|_2}*\frac{(p-q)}{2}+b=0$, and then $b=-\frac{\|p-q\|_2}{2}$, i.e., hyper plane $\{x|\frac{(p-q)'*x}{\|(p-q)\|_2}-\frac{\|p-q\|_2}{2}=0\}$ divides P_i into two parts, i.e., P_j and P_k within $node_j$ and $node_k$, respectively.

(c) Find the nearest point $s\in P_j$ to the red hyper plane (red dashed line), and find the nearest point $t\in P_k$ to the red

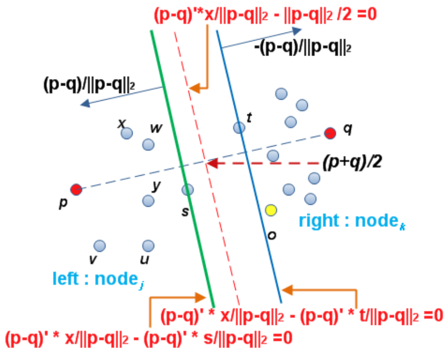

(p-q)'*x/||p-q||2 - ||p-q||2 /2 =0
-(p-q)/||p-q||2
(p-q)/||p-q||2
x
w
t
q
(p+q)/2
p
y
s
o
right : node_k
v
u
left : node_j
(p-q)' * x/||p-q||2 - (p-q)' * t/||p-q||2 =0
(p-q)' * x/||p-q||2 - (p-q)' * s/||p-q||2 =0

Fig. 4. Find a hyper plane (red dash line) to divide P_i into $node_j$ and $node_k$, and refine it to be green line for $node_j$, because the green line is tightly close to $node_j$. Similarly, the blue line is the constraint hyper plane for $node_k$.

hyper plane (red dashed line);

(d) In order to build semi-convex hulls that cover all points in two sub nodes, respectively, it is necessary to move the red dashed line parallelly to point s and t, respectively.

- For $node_j$, move the hyper plane parallelly such that it passes point s, as the green line shows, which makes the green line tightly close to the left child node. It is also easy to determine the green hyper plane equation as: $\frac{(p-q)'*x}{\|(p-q)\|_2} - \frac{(p-q)'*s}{\|(p-q)\|_2} = 0$. Therefore, the green line is the constraint hyper plane for $node_j$, i.e., $\forall x \in P_j$, we have $\frac{(p-q)'*x}{\|(p-q)\|_2} - \frac{(p-q)'*s}{\|(p-q)\|_2} \geq 0$, and $P_j = \{p, v, u, x, y, w, s\}$.
- Similarly, the blue line is the constraint hyper plane for P_k, i.e., $\forall x \in node_k$, we have $\frac{(q-p)'*x}{\|(q-p)\|_2} - \frac{(q-p)'*t}{\|(q-p)\|_2} \geq 0$.

(2) Inherit all constraints from parent node and refine: Take $node_j$ for example again, suppose we have divided $node_j$ into two parts $node_m$ and $node_n$ by the same way as mentioned in step (1), and get the blue line as new constraint on $node_m$. Because $node_j$ is the parent node of $node_m$, then $P_m \in P_j$. Therefore, constraints on $node_j$ is also effective for $node_m$, i.e., $\forall x \in H_m$, we have $\frac{(p-q)'*x}{\|(p-q)\|_2} - \frac{(p-q)'*s}{\|(p-q)\|_2} \geq 0$. However, as we can see in Fig. 5, the green hyper plane is obviously not tightly close to $node_m$, therefore we can refine it to be the red hyper plane which pass point u, because u is the closest point in P_m to the green line. Untill now, $node_m$ has two constraints, i.e. the red line and the blue line. While for $node_n$ it is unnecessary to refine the green line, because the green line still pass $s \in P_n$, and $node_n$ also has two constraints. If $node_j$ has more than one constraints, then all constraints should be refined by the same way.

Algorithm 1 presents the process of building a semi-convex hull tree. It builds tree recursively, both time and space complexities are about $O(n)$.

IV. Semi-convex hull tree based NN query

Let H_i be the convex set on $node_i$, then $dist(node_i, q)$ is:

$$\min_x \ dist(x, q) \quad s.t. \quad \begin{cases} \alpha'_{i,1} * x \geq \beta_{i,1} \\ ..... \\ \alpha'_{i,l} * x \geq \beta_{i,l} \end{cases} \tag{1}$$

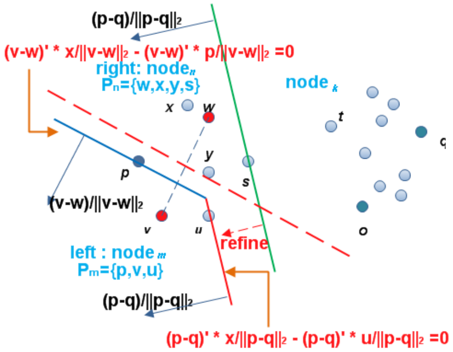


Fig. 5. Red dash line is a new hyper plane that divides $node_j$ into $node_m$ and $node_n$, and blue line is refined from red dash line for $node_m$. Green line is the constraint on $node_j$ and should be refined to the red line for $node_m$.

```
Algorithm 1 Build Semi-Convex Hull Tree: build_tree
Input: data $P$, min number of points for leaf node $MinPts$, parent node $pNode$
Output: node : $curNode$
create a NODE $curNode$
inherit and refine all hyper planes (constraints) of $pNode$ for $curNode$
if length($P$)> $MinPts$ then
    find a hyper plane $\{x|\alpha' * x + b = 0\}$ to divide $P$
    $P_1$ = points within left node
    leftNode=build_tree($P_1$, $MinPts$, $pNode$)
    $curNode.left = leftNode$
    $P_2$ = points within right node
    rightNode=build_tree($P_2$, $MinPts$, $pNode$)
    $curNode.right = rightNode$
else
    $curNode.isLeaf = true$
    $curNode.points = P$ % save points
end if
return $curNode$
```

where l is the number of constraints of $node_i$. Obviously, it is a classic quadratic programming problem, and $dist(node_i, q) \leq \min_{p \in P_i}(dist(q, p))$, because $P_i \in H_i$. Therefore, we can skip $node_i$ as well as all of its sub nodes, provided $dist(node_i, q)$ is larger than the current nearest distance. On CPU, we can start from root node and recursively visit the tree, if one node is skipped, all of its sub nodes are also filtered. However on GPUs, there are two drawbacks to perform by this way, as follows: (1) The original algorithm of quadratic programming (e.g., Gaspero: http://www.diegm.uniud.it/digaspero) is too complex to be implemented on GPUs directly, because OpenCL [17] restricts many standard operations defined in C/C++, e.g, *new* and *malloc*. (2) Recursively visiting tree and nodes filtering cause pointer jumping which leads to a nonsatisfying performance due to suboptimal memory accesses.

Therefore, we have to simplify quadratic programming so that it can run on GPUs, and only visits leaf nodes by order.

Algorithm 2 Initialize $kNNResult$

```
Input: semi-convex hull tree $nodes$; query point $q$.
Output: $kNNResult$ of $q$.
$curNode = nodes[0]$; %root is current search node
while $curNode$ is not leaf do
    $leftDist = dist(q, curNode.left.center)$
    $rightDist = dist(q, curNode.right.center)$
    $result = curNode.left$
    if $leftDist > right$ then
        $curNode = curNode.right$
    end if
end while
Find $kNNResult$ from $curNode$ %brute force
```

Simplified quadratic programming: Our goal is to resolve the quadratic programming problem approximately, which makes it as simple as possible.

Theorem 1. *Let AHP_i be a set that contains all active constraint hyper planes w.r.t q and $node_i$,* $\min_{p \in P_i} dist(q,p) \geq dist(node_i, q) \geq \max_{HP_{i,j} \in AHP_i} dist(q, HP_{i,j})$.

Proof. (1) Suppose $C_{i,j} \in AHP_i$, i.e., $C_{i,j}$ is active w.r.t q and $node_i$, then according to Definition 3 we know that the separating hyper-plane $HP_{i,j}$ divides point q from half space $\{x|\alpha'_{i,j} * x \geq \beta_{i,j}\}$, and $dist(HP_{i,j}, q)$ is the distance from q to $HP_{i,j}$, which also is the min distance from q to the half space. And $\because$ $H_i \subset \{x|\alpha'_{i,j} * x \geq \beta_{i,j}\}$, then $dist(node_i, q) \geq dist(q, HP_{i,j})$, $\therefore$ $dist(node_i, q) \geq \max_{HP_{i,j} \in AHP_i} dist(q, HP_{i,j})$. (2) $\because$ P_i is within $node_i$ then $P_i \subset H_i$, and then $\min_{p \in P_i} dist(q,p) \geq dist(q, HP_{i,j})$. □

Take Fig. 2 for example again, suppose $dist(q, P_i) = dist(q, w)$ and $dist(q, H_{i,1}) > dist(q, H_{i,2})$, because both $C_{i,1}$ and $C_{i,2}$ are active constraints w.r.t q and $node_i$, therefore, $dist(q, w) \geq dist(q, node_i) \geq dist(q, H_{i,1})$ holds.

The simplified quadratic programming only finds the max distance from q to all active constraint hyper planes, which has low complexity. While k-d tree only uses the difference in the splitting axis as the distance from q to a node (see Fig.1), which is usually far less than the result got by simplified quadratic programming, let alone $dist(node_i, q)$. Thus, this is one main advantage to k-d tree.

Initialize nearest neighbors: The first step of query nearest neighbors is to initialize K-Nearest Neighbors $kNNResult$, we perform this step on HOST which shown in Algorithm 2, which runs in $O(MinPts * \log(|nodes|))$ where $|nodes|$ is nodes number.

Visiting leaf nodes by order: As mentioned above, it is better for GPUs to visiting data by order, then we re-sort the original data points according to the order of leaf node. Suppose $P_{ln_1}, P_{ln_2}, P_{ln_3}...$ are the point sets within leaf nodes $node_{ln_1}, node_{ln_2}, node_{ln_3}...$, respectively, where $ln_1 < ln_2 < ln_3...$, then the sorted data is $SortP = \{P_{ln_1}, P_{ln_2}, P_{ln_3}, ...\}$. Work units of GPUs perform queries independently in parallel, and the algorithm performed in each work unit is shown in Algorithm 3. For each work unit, the complexity is $O(MinPts * log(\frac{2n}{MinPts}))$ given a fixed intrinsic dimensionality. Therefore, the total complexity is $O(\frac{MinPts * log(\frac{2n}{MinPts})}{comp\_units\_num})$, where $comp\_units\_num$ are thread number and compute units number of GPUs.

Algorithm 3 Fast K-Nearest Neighbor Query: $fKNN$

```
Input: sorted data $SortP$; the number of $K$; query point
    $q$; current K-Nearest Neighbors $kNNResult$ of $q$; sorted
    leaf node sets $node_{ln_1}, node_{ln_2}, node_{ln_3}...$
Output: K-Nearest Neighbors $kNNResult$ of $q$.
$cur\_min\_dist$ = min distance from $q$ to $kNNResult$
for each leaf node $node_{ln_i}$ do
    skip_node=false
    for each constraint $C_{ln_i,j}$ do
        if $C_{ln_i,j}$ is active w.r.t $q$ and $node_{ln_i}$ then
            if $dist(q, HP_{ln_i,j}) > cur\_min\_dist$ then
                skip_node=true
                break
            end if
        end if
    end for
    if not skip_node then
        brute force compute distance from $q$ to all points in
        $P_{ln_i}$, update $kNNResult$ and $cur\_min\_dist$
    end if
end for
```

V. Experiments

In this section, we conduct experiments to evaluate the proposed algorithms on Windows 10 64-bit with Intel(R) Core (TM) i5-4590 CPU @3.30GHz 3.30GHz, 4 GB memory, and make comparisons with buffer k-d tree, brute force (gpu) and cover tree on different data sets. All codes of theses algorithms were compiled in C++ on TDM-GCC-32. We set $MinPts = n * percent$ for semi-convex hull tree where n is the cardinality, and $percent \in (0, 1]$. The configurations of GPU devices and OpenCL are the following: (1) Intel(R) HD Graphics 4400 with 20 parallel compute units, and max global memory size is 14,488 MB, OpenCL is Intel SDK for OpenCL (windows) (2) Nvidia Geforce GTX 1050 TI, 2 GB memory with 6 multiprocessors + NVIDIA CUDA8.0 for windows (OpenCL is included).

Data sets come from UCI (https://archive.ics.uci.edu/ml/index.php), including *PAM* (PAMPA2: dim=4, n=3,850,505), *HOUSE* (Household: dim=7, n=2,049,280), *USCENCUS 1990* (dim=8, n=3.65 $*$ 10^5), *HANDPOSTURE (dim=15, n=78,095)*, *REACTION (dim=29, n=33,913)*, *APS (dim=70, n=60,000)* and *FONT* which includes *BODONI* (dim=70, n=3,964) and *CALIBRI* (dim=70, n=16,364)). All same points are removed, all missing values are set to be 0, and each dimension is normalized to $[0, 10^5]$.

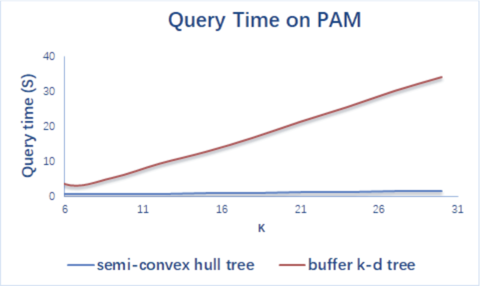


Fig. 6. Comparison of runtime for random 20,000 queries on PAM with K varies at Intel GPU. (percent =0.1%; depth=6 for buffer k-d tree.)

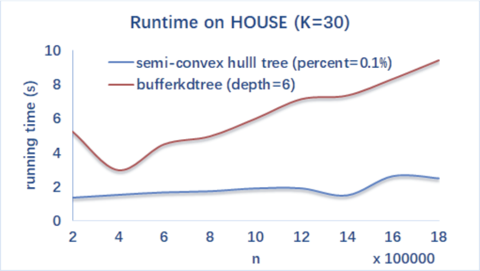


Fig. 7. Comparison of runtime for random 20,000 queries on PAM with n varies at Intel GPU. ($K = 30$; percent =0.1%; depth=6 for buffer k-d tree.)

A. *Runtime comparisons*

In this part, we compare the query time for 20,000 queries with K varies on *PAM*, fix $percent = 0.1\%$ for the proposed algoithm, and $depth = 6$ for buffer k-d tree. Each query is generated by: (1) selected from source data set randomly; (2) for each dimension, add a noise by 'rand()/100' in C++. (The purpose of adding noise is to avoid duplicate queries).

As Fig. 6 show, we can see that both algorithms run linearly with K increasing. However, it also addresses that the proposed algorithm runs far faster than buffer k-d tree.

B. *Distance computations comparisons*

In order to compare distance computations, we conduct three experiments on whole $HOUSE$, PAM and $USCENCUS$. The total distance computations of the proposed algorithm includes two parts, (1) distance computations from query points to constraints hyper planes, (2) normal distance computations between two points. While buffer k-d tree only counts normal distance computations.

As Tab. I shows, the proposed algorithm filter far more distance computations than buffer k-d tree. In PAM, semi-convex hull tree improves it greatly when K is small, e.g, the speedup is more than 100 from K=9 to K=15. In $HOUSE$ the speedup also varies from 13.17 to 35.86 with K changing from 30 to 90, and similar things happens in $USCENCUS$.

It is also inferred, due to the "curse of dimensionality" semi-convex hull tree still works well in low dimension, and its superiority to buffer k-d tree becomes weak in high dimension.

TABLE I
COMPARISONS OF DISTANCE COMPUTATIONS FOR RANDOM 20,000 QUERIES. (PERCENT =0.1%; DEPTH=9 FOR BUFFER k-D TREE). THE TOTAL DISTANCE COMPUTATIONS OF SEMI-CONVEX HULL TREE = (1) DISTANCE COMPUTATIONS FROM QUERY POINTS TO CONSTRAINTS HYPER PLANES + (2) NORMAL DISTANCE COMPUTATIONS.

Data Set	K	semi-convex	bufferkdtree	speedup
PAM	9	201,808,076	22,871,116,448	113.33
	15	234,965,281	25,675,140,084	109.27
	21	366,390,416	28,007,582,503	76.44
	30	2,230,881,456	31,056,015,629	13.92
HOUSE	30	401,557,607	14,400,253,026	35.86
	50	656,272,039	14,624,010,688	22.28
	70	700,681,446	14,624,010,688	20.87
	90	1,135,037,232	14,950,763,584	13.17
USCENCUS	30	318,820,278	4,150,426,89	13.02
	50	383,999,453	4,307,871,400	11.22
	70	418,892,656	4,368,218,220	10.43
	90	470,540,381	4,530,582,291	9.63

It is notable that the improvement of query time is not so remarkable as that of distance computations, the reason is that Algorithm 3 does not perfectly abide by the specification of OpenCL, (e.g., data align, coalesced access and data vectorization etc.), which leads to a nonsatisfying performance due to conditional computations and suboptimal memory accesses. While buffer k-d tree solves this problem by reorganizing the data structure, and utilizing a lazy strategy which firstly only collects the leaf that needs to be processed next instead of push all leaf to GPUs. Therefore, we will improve our algorithm by similar way in future.

TABLE II
RUNTIME COMPARISONS OF 10^6 QUERIES ON DIFFERENT DATA SET AT INTEL(R) HD GRAPHICS 4400, K=30. (BKT: BUFFER KD TREE; SCT: SEMI-CONVEX TREE; BRUTE: BRUTE-FORCE; COVER: COVER TREE)

Data Set	Intel GPU			CPU
	BKT/depth=9	**SCT/percent**	**BRUTE**	**COVER**
PAM	85.64	**49.07** /0.001	5595.35	54.86
HOUSE	148.39	**95.22** /0.001	2989.04	126.69
USCENCUS	217.06	**68.60** /0.005	331.87	81.86
HAND	342.35	85.92 /0.005	121	**53.92**
REACTION	583.47	90.20/0.005	**76.78**	642.56
APS	565.1	**174.02**/0.005	205.91	259.03
BODONI	1120.15	40.15/0.005	**31.24**	46.64
CALIBRI	694.9	96.88/0.005	87.62	**53.54**

C. *Comparison of massive queries*

(1) **Run on Intel(R) HD Graphics 4400**: We compare the runtime of massive queries for all algorithms on several data sets, as shown in Table II, K is fixed to 30, the depth of buffer k-d tree and the leaf threshold percent of semi-convex hull tree are both well chosen such that runtime in table is nearly the best result, and it is observed: (a) On high dimensional small data set (e.g, *BODONI*, *REACTION* and APS etc), the performances of our algorithm and cover tree are close to brute force (gpu), while buffer k-d tree doesn't work well, that's because the general rule $n >> 2^d$ is not satisfied. (b) However, on low dimensional massive data set (e.g, *PAM*, *HOUSE*), buffer k-d tree, cover tree and our algorithm won brute force

algorithm a lot. The proposed algorithm outperforms buffer k-d tree and cover tree on these data sets, and yields valuable speedups over buffer k-d tree.

(2) **Run on Nvidia Geforce GTX 1050 Ti**: Due to the memory limitation for each work unit on Nvidia Geforce GTX 1050 Ti, launching large scale queries with large K at one time often yields out of resource for buffer k-d tree and the proposed method, therefore we test the three methods on subsets of PAM, $HOUSE$, $USCENCUS$ and $HANDPOSTURE$ with a relative large K (200), respectively. In the case of K is large, the results are shown in Tab. III, and it is observed: (a) buffer k-d tree is quite inferior to cover tree and the proposed method; (b) in the case of very low dimension, e.g 4 dim (*PAM*) and 7 dim (*HOUSE*), semi-convex hull tree is close to cover tree, however, the proposed algorithm outperforms cover tree with dimension increasing, e.g, semi-convex hull tree has obvious advantage to cover tree on $HANDPOSTURE$ and $USCENCUS$.

From all experiments in this subsection, we can see that the proposed method can perform well and outperform buffer k-d tree within about 15 dimension, but in high dimensional data set semi-convex hull tree is close to GPU brute force.

TABLE III
RUNTIME COMPARISONS OF 1000 QUERIES WITH K=200 ON DIFFERENT DATA SETS AT NVIDIA GEFORCE GTX 1050 TI. ALL PARAMETERS ARE WELL CHOSEN FOR EACH ALGORITHM.

Data Set	n	bufferkdtree	semi-convex	cover tree
PAM	100,000	2.38	0.32	**0.30**
	600,000	3.51	**0.32**	0.37
	1,000,000	2.51	**0.32**	0.39
HOUSE	100,000	2.08	0.38	**0.34**
	600,000	3.20	**0.47**	0.46
	1,000,000	2.51	**0.50**	0.67
USCENCUS	36,000	1.43	**0.26**	0.38
	216,000	2.49	**0.25**	0.78
	360,000	3.35	**0.29**	0.93
HAND POSTURE	7,800	0.66	**0.16**	0.33
	46,800	1.76	**0.31**	1.35
	78,000	2.12	**0.31**	2.39

VI. CONCLUSION

In this paper, we propose a novel data structure, semi-convex hull tree, which is used to retrieve exact nearest neighbors on GPUs for large scale queries. Each node represents a semi-convex hull which is made of a set of linear inequations (non-axis-parallel hyper planes), and a leaf node saves a set of data points within it. In order to perform the task of querying nearest neighbors on GPUs, approximate quadratic programming algorithm is invented and used to filter unnecessary distance computations. We conduct a set of experiments and prove that the proposed algorithm is far superior to k-d tree in that it can greatly reduce many unnecessary distance computations.

However, there is a main deficiency of the proposed method, that is its data scheduling strategy on GPU does not perfectly abide by specification of OpenCL or CUDA, which leads to a nonsatisfying performance due to conditional computations and suboptimal memory accesses, and yields that the runtime of NN query is not as good as that of theoretical analysis (e.g, Tab. I). Therefore, our future work is to go on improving the proposed algorithm.

ACKNOWLEDGMENT

This work was supported by the National Science Foundation of China (No.61673186,71771094,61572205); the Open Project Program of the State Key Lab of CAD&CG(Grant No.A1722), Zhejiang University; the Open Project Program of the National Laboratory of Pattern Recognition (NLPR) (NO. 201700002); the Natural Science Foundation of Fujian Province (No.2016J01303); Project of science and technology plan of Fujian Province of China (No.2017H01010065).

REFERENCES

[1] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Commun. ACM*, vol. 51, no. 1, p. 117C122, 2008.

[2] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa, "The quickhull algorithm for convex hulls," *ACM Transactions on Mathematical Software (TOMS)*, vol. 22, no. 4, pp. 469–483, 1996.

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

[4] A. Beygelzimer, S. Kakade, and J. Langford, "Cover trees for nearest neighbor," in *Proceedings of the 23rd international conference on Machine learning*. ACM, 2006, pp. 97–104.

[5] Y. Chen, J. P. Singh, L. Zhou, and N. Bouguila, "Frs: Fast range search by pruning unnecessary distance computations based on k-d tree," in *IEEE International Conference on Data Mining Workshops*, 2017, pp. 1160–1165.

[6] Y. Chen, S. Tang, L. Zhou, C. Wang, J. Du, T. Wang, and S. Pei, "Decentralized clustering by finding loose and distributed density cores," *Information Sciences*, vol. 433-434, pp. 510–526, 2018.

[7] Y. Chen, S. Tang, N. Bouguila, C. Wang, J. Du, and H. L. Li, "A fast clustering algorithm based on pruning unnecessary distance computations in dbscan for high-dimensional data," *Pattern Recognition*, 2018 (https://doi.org/10.1016/j.patcog.2018.05.030.).

[8] M. De Berg, M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf, "Computational geometry," in *Computational geometry*. Springer, 2000, pp. 1–17.

[9] F. Gieseke, J. Heinermann, C. E. Oancea, and C. Igel, "Buffer kd trees: Processing massive nearest neighbor queries on gpus." in *ICML*, 2014, pp. 172–180.

[10] B. Leibe, K. Mikolajczyk, and B. Schiele, "Efficient clustering and matching for object class recognition." in *BMVC*, 2006, pp. 789–798.

[11] D. G. L. Marius Muja, "Scalable nearest neighbor algorithms for high dimensional data," *IEEE Transactions on Pattern Analysis And Machine Intelligence*, vol. 36, no. 11, pp. 2227–2240, 2014.

[12] D. Nister and H. Stewenius, "Scalable recognition with a vocabulary tree," in *Computer vision and pattern recognition, 2006 IEEE computer society conference on*, vol. 2. Ieee, 2006, pp. 2161–2168.

[13] F. P. Preparata and M. Shamos, *Computational geometry: an introduction*. Springer Science & Business Media, 2012.

[14] T. B. Sebastian and B. B. Kimia, "Metric-based shape retrieval in large databases," in *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, vol. 3. IEEE, 2002, pp. 291–296.

[15] G. Shakhnarovich, P. Indyk, and T. Darrell, *Nearest-neighbor methods in learning and vision: theory and practice*, 2006.

[16] C. Silpa-Anan and R. Hartley, "Optimised kd-trees for fast image descriptor matching," in *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*. IEEE, 2008, pp. 1–8.

[17] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[18] J. Wang, N. Wang, Y. Jia, J. Li, G. Zeng, H. Zha, and X.-S. Hua, "Trinary-projection trees for approximate nearest neighbor search," *IEEE Transactions on Pattern Analysis And Machine Intelligence*, vol. 36, no. 2, pp. 388–403, 2014.