Sweet KNN: An Efficient KNN on GPU through Reconciliation between Redundancy Removal and Regularity

Guoyang Chen, Yufei Ding, and Xipeng Shen
Computer Science Department
North Carolina State University
Raleigh, NC, USA 27519
Email: {gchen11,yding8,xshen5}@ncsu.edu

***Abstract*—Finding the k nearest neighbors of a query point or a set of query points (KNN) is a fundamental problem in many application domains. It is expensive to do. Prior efforts in improving its speed have followed two directions with conflicting considerations: One tries to minimize the redundant distance computations but often introduces irregularities into computations, the other tries to exploit the regularity in computations to best exert the power of GPU-like massively parallel processors, which often introduces even extra distance computations. This work gives a detailed study on how to effectively combine the strengths of both approaches. It manages to reconcile the polar opposite effects of the two directions through elastic algorithmic designs, adaptive runtime configurations, and a set of careful implementation-level optimizations. The efforts finally lead to a new KNN on GPU named *Sweet KNN*, the first high-performance triangular-inequality-based KNN on GPU that manages to reach a sweet point between redundancy minimization and regularity preservation for various datasets. Experiments on a set of datasets show that Sweet KNN outperforms existing GPU implementations on KNN by up to 120X (11X on average).**

I. Introduction

K-Nearest Neighbor (KNN) is an algorithm for finding the k points in a target set that are closest to a given query point. As a general-purpose mean of comparing data, KNN is commonly used in a variety of fields (information retrieval, image classification, pattern recognition, etc). It has been rated as one of the top-10 most influential data mining algorithms [1], and has received many attentions in data engineering [2]–[5].

The basic KNN algorithm is inherently expensive, requiring the computations of the distances from the query point to each target point. When there is not just one but a set of query points, the process could take a long time to finish. Such a problem setting is also known as *KNN join* problem—it is the focused setting in this paper.

There have been a number of studies trying to improve KNN efficiency. They fall into two main categories.

The first category focuses on minimizing the amount of distance computations. They are primarily about algorithm-level optimizations. Several previous studies, for instance, have shown that a clever usage of triangular inequality can avoid most unnecessary distance calculations in KNN [4], [6], [7]. Others have studied the usage of KD-tree [8]–[10], approximations [2], [11], [12], and other algorithmic optimizations for a similar purpose. While these methods try to minimize redundant distance computations, they often introduce irregularities into the computations. For instance, in Ding and others' triangular inequality work [4], several condition checks are used to filter out unnecessary distance calculations. The checking results may differ on different data points. Consequently, the computations (execution paths and sets of operations) may differ across data points.

The second category focuses on implementation-level optimizations, trying to better leverage underlying computing systems for acceleration. The most prominent example is the recent efforts in speeding up KNN through Graphics Processing Units (GPU) [13]–[15]. As these systems typically feature massive parallelism best suiting regular data-level parallel computations (i.e., the processing of all data points follows the same execution path), this category of efforts attempt to enhance the regularity in computations. These efforts however often introduce extra redundant computations. For instance, the state-of-the-art implementation of KNN on GPU [13]–[15] uses the matrix multiplication routine in a highly tuned linear algebra library CUBLAS [16] to compute the distances between the query and target point sets. Even though the approach may end up computing the distances between two points twice (e.g., when the query set equals the target set) due to the matrix multiplication formulation of the problem, the increased regularity allows CUBLAS to better take advantage of the GPU computing resource, achieving a high speed.

The two directions of efforts take opposite means, as illustrated in Figure 1. One tries to minimize redundancy through algorithmic optimizations, but introduces irregularities; the other tries to enhance regularity, but adds extra redundant computations. As a result, the two directions of efforts have been going separately. An exception is the work by Barrientos and others [14], which tries to implement a region-based KNN algorithm on GPU. It, however, shows only a modest speed (even much slowdown in some settings compared to prior work), thanks to the tension between regularity and redundancy.

This paper describes our efforts in battling the principled tension faced in merging the two directions of efforts to make KNN efficient. We specifically concentrate on developing an efficient triangular inequality-based KNN on GPU. We investigate a set of techniques to effectively reconcile the opposite

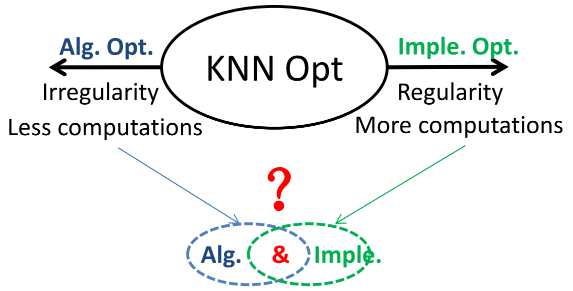


Fig. 1. Two main directions for speeding up KNN.

means of the two directions. At the algorithm level, we introduce an adaptive design, which, based on the properties of the current data sets, automatically adjusts the algorithm and parallelism on the fly to reach a sweet point between regularity preservation and redundancy minimization. At the implementation level, we explore optimizations of data layout and data placement on memory and thread-data remapping to remove irregularities in the computations.

Putting these techniques together, we create Sweet KNN, an efficient KNN implementation on GPU. By reaching a sweet point between redundancy minimization and regularity preservation, Sweet KNN achieves 11X average (up to 120X) speedups over the prior GPU implementations of KNN.

The rest of the paper is organized as follows. Section II first gives some background knowledge. Section III then describes our basic implementation of triangular inequality-based KNN on GPU. Section IV elaborates on our solutions for dealing with the regularity-redundancy tension to bring out the large potential of Sweet KNN. Section V reports the experimental results. Section VI concludes the paper with a short summary.

II. Background

This section presents some background on GPU, triangular inequality and its role in KNN.

A. GPU

As a massively parallel architecture, GPU features hundreds or thousands of cores. GPU is often equipped with several types of memory. On Tesla K20c, for instance, the memory consists of *global memory*, *texture memory*, *constant memory*, *shared memory*, and a variety of cache. These types of memory differ in size, access constraints and latency. When a GPU kernel gets launched, usually thousands of threads will be created and many of them start running on GPU concurrently. These threads are organized in a hierarchy: 32 consecutive threads form a *warp* and they execute in lockstep, a number of warps form a thread block, and all blocks form a grid. When a GPU function (called a GPU kernel) is launched, many GPU threads get created, which all execute the same GPU function. Thread ID is used in the kernel code to differentiate the behaviors of the threads.

There are two factors that critically affect the performance of a GPU program. The first is *memory coalescing* on global memory. Roughly speaking, when the memory locations accessed by all the threads in a warp fall into a small memory segment (128 bytes), the accesses will get coalesced and one memory transaction is sufficient to bring all. Otherwise, multiple memory transactions would be needed.

The second factor is called *thread divergence*. It happens when the threads in a warp diverge at the values of some condition checks, which lead them into executing different branches of the condition statements (e.g., some threads execute the "if" branch while others execute the "else" branch). Upon a thread divergence, the different groups of threads' executions get serialized. When one group is executing one branch, the other group has to wait in idle.

Both factors entail the importance of regularity for a GPU kernel to gain high performance on GPU. When a kernel contains lots of condition checks and irregular memory accesses, consecutive threads may end up diverging in control flows and memory segments to access, resulting in low performance.

B. Triangle Inequality (TI) and Landmarks

KNN involves extensive point-to-point distance calculations. Previous works [4] have shown that TI is a theorem very useful for avoiding some unnecessary distance calculations.

A formal definition of TI is as follows:
Let q, t, L represent three points and $d(point_A, point_B)$ represent the distance between $point_A$ and $point_B$ in some metric (e.g., Euclidean distance). *Triangular Inequality (TI)* states that $d(q,t) \leq d(q,L) + d(L,t)$. The assistant point L is called a *landmark*.

Directly from the definition, we could compute both the lowerbound (LB) and upperbound (UB) of the distance between two points as follows. Figure 2 gives an illustration.

$$LB(q,t) = |d(q,L) - d(t,L)| \tag{1}$$
$$UB(q,t) = d(q,L) + d(t,L) \tag{2}$$

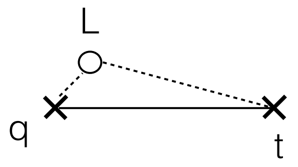
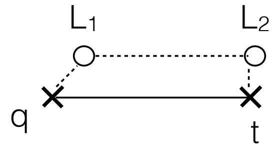


Fig. 2. Illustration of distance bounds obtained from Triangular Inequality through one *landmark* L.

The bounds can be used to approximate the distance between the query and the target point and avoid the need for computing their distances in KNN. For example, suppose that the so-far k^{th} closest distance to query point q is d_k, and t is the next target point to check. As long as $LB(q,t) \geq d_k$, we can simply omit the computation of the exact distance between q and t as t cannot be part of the k points closest to q.

A simple extension to the theorem leads to an alternative method for distance estimation. It uses two landmarks, with one close to the query and the other close to the target, as Figure 3 shows. The lowerbound (LB) and upperbound (UB) of the distance can be estimated as follows (assuming $d(L_1, L_2)$ is much larger than $d(q, L_1)$ and $d(L_2, t)$):

$$LB(q,t) = d(L_1,L_2) - d(q,L_1) - d(L_2,t) \tag{3}$$
$$UB(q,t) = d(q,L_1) + d(L_1,L_2) + d(L_2,t). \tag{4}$$



$$LB(q, t) = d(L1, L2) - d(q, L1) - d(L2, t)$$

$$UB(q, t) = d(q, L1) + d(L1, L2) + d(L2, t)$$

Fig. 3. Illustration of how two landmarks can be used for computing lower and upper bounds of distances.

The proof is shown in previous work [4].

C. TI-Based KNN

TI-based KNN uses the aforementioned triangular inequality (both one and two-landmark cases) to avoid some unnecessary distance calculations. Understanding the TI-based KNN algorithm is needed for following the rest of this paper.

Before reviewing TI-based KNN, we first introduce a set of notations to be used in the rest of this paper.

$\mathbf{Q}$:	a set of query points;
$\mathbf{T}$:	a set of target points;
k:	the number of nearest neighbors to find;
m^q:	the number of clusters formed on Q;
m^t:	the number of clusters formed on T;
$d_k(q, \mathbf{T})$:	The distance from a point q to its k^{th} nearest neighbor in a target data set $\mathbf{T}$. It is also called the k^{th} nearest neighbor distance of q.

Figure 4 outlines the pseudo-code of the TI-based KNN. It contains three main steps.

Step 1: Initialize Clusters

This step first, through function `detLmNum`($\mathbf{Q}$,$\mathbf{T}$) in Figure 4, determines the number of clusters (m^q and m^t) to form on the query set and the target set respectively. The method is to set the numbers to $3 * \sqrt{|\mathbf{Q}|}$ and $3 * \sqrt{|\mathbf{T}|}$ (if the space is not enough, use the largest possible numbers). It then creates landmarks for both the query set and the target set through sampling or other pivot selection techniques [3], [4], [17]. Our practice (`init`($\mathbf{Q}$, $\mathbf{T}$, m^q, m^t) in Figure 4) follows the previous work [4] as detailed in Section III-A. The `init` function then assigns each query or target point to the closest landmark, forming query clusters and target clusters. We also call the landmark of a cluster its center. For each query cluster, this step records the maximal distance from its members to the cluster center. For each target cluster, it records the distances from each of its points to the center, and sorts those points in descending order of the distances.

After Step 1, the algorithm starts a loop, treating each query cluster. There are two main steps in the treatment.

Step 2: Choose Candidate Clusters (level-1 filtering)

The purpose of this step is to exclude some target clusters. For one query cluster, there may be several target clusters which are far away from the query cluster such that points in those target clusters are impossible to be among the k nearest neighbors of any point in the query cluster. This step tries to find such target clusters and exclude them from further consideration. It does it in two substeps:

Step 2.1: Calculate the Upper Bound (`calUB`($\mathbf{q}$, $\mathbf{C^t}$, k) in Figure 4). This substep calculates the upper bound UB, which is a value no smaller than $d_k(q, \mathbf{C^t})$ for all $q \in \mathbf{q}$. It calls `getUBs`($\mathbf{q}$, $\mathbf{c_j}$, k) on each target cluster ($\mathbf{c_j} \in \mathbf{C^t}$), which returns k bounds, with the i^{th} bound being guaranteed to be no smaller than $d_i(q, \mathbf{c_j})$ for all $q \in \mathbf{q}$.

To get the k bounds, `getUBs`($\mathbf{q}$, $\mathbf{c_j}$, k) uses TI (the 2-landmark case). It is illustrated in Figure 5, in which, u, v, and w are three points in $\mathbf{t1}$ that are closest to the landmark c_1, while a is the point in $\mathbf{q}$ that is farthest from landmark c_q. The bounds returned by `getUBs`($\mathbf{q}$, $\mathbf{c_1}$, 3) would be $d(a, c_q) + d(c_q, c_1) + d(c_1, u)$, $d(a, c_q) + d(c_q, c_1) + d(c_1, v)$, and $d(a, c_q) + d(c_q, c_1) + d(c_1, w)$, per the second equation shown in Figure 3. (Step 1 has prepared the needed info for this step.)

Procedure `calUB`($\mathbf{q}$, $\mathbf{C^t}$, k) pools all these upper bounds of all target clusters together, and picks the k^{th} smallest among them as the UB for the query cluster[1]. That ensures that UB is no smaller than the k^{th} nearest neighbor distance of any query point.

Step 2.2: Filtering based on UB (`groupFilter`(UB, $\mathbf{q}$, $\mathbf{C}$) in Figure 4). This substep goes through every target cluster. For a given target cluster, it calculates the lowerbound (l) of the group-to-group distances from the query cluster to the target cluster (`getLB`($\mathbf{q}$, $\mathbf{t}$) in Figure 4). It does it by applying the triangular inequality (2-landmark case) to the points in $\mathbf{q}$ and $\mathbf{t}$ that are farthest to their centers. In Figure 5, for instance, `getLB`($\mathbf{q}$, $\mathbf{t1}$) returns $d(c_q, c_1) - d(a, c_q) - d(f, c_1)$ (a and f are the farthest points in $\mathbf{q}$ and $\mathbf{t_1}$ from c_q and c_1 respectively). If $UB < l$, then this target cluster is too far from the query cluster. Otherwise, it is chosen as a candidate target cluster for the query cluster for further examination.

Step 3: Point-level filtering (level-2 filtering)

This step (`pointFilter`($\mathbf{S}$, $\mathbf{q}$, UB) in Figure 4) examines the points in the candidate target clusters to find k nearest neighbors for each query point. It tries to avoid unnecessary distance calculations at each point. It first sorts the candidate target clusters in ascending order based on the distances from their centers to the query center. Recall that in Step 1, points in a target cluster are already sorted in a descending order of their point-to-center distances. In such an order, the algorithm examines all the candidate target points when treating each query point. The order is essential for effective filtering. When examining each of the candidate points (t), it applies triangle inequality (1-landmark case) to the target point (t) and the query point (q) as follows: $l = d(q, tc) - d(t, tc)$, where tc is the target center. The algorithm computes $d(q, t)$ only if $|l| \leq$ UB. If $l > UB$, thanks to the order of examination, this target point and all remaining points in the target cluster are too far from the query point (as their lower bounds

[1]There are some subtle optimizations: getUBs($\mathbf{q}$,$\mathbf{t}$,k) may terminate early if its newly attained upper bound is already greater than the k^{th} smallest bounds seen so far.

```
// Input:
//     Q: query dataset; T: target dataset;
//     k: # of nearest neighbors to find
// Output:
//     R: the set of the k nearest neighbors
Main Procedure of TI-based KNN
// $C^q$, $C^t$: grouping of Q and T by landmarks
1. [$m^q$, $m^t$] = detLmNum(Q, T); //# landmarks to create
2. [$C^q$, $C^t$] = init(Q, T, $m^q$, $m^t$); // cluster Q and T
3. foreach q in $C^q$ // foreach query cluster
4.     UB = calUB(q, $C^t$, k);
5.     candGroups = groupFilter(UB, q, $C^t$);
6.     R = R ∪ pointFilter(candGroups, q, UB));

// get UB of the $k^{th}$ nearest neighbor
// distances of a cluster q, regarding
// the set of target point clusters C
Procedure calUB(q, C, k)
1. W = {};
2. foreach $c_j \in$ C //foreach target cluster
3.     W = W ∪ getUBs(q, $c_j$, k);
4. return the kth smallest value in W

// get k UBs of the k shortest distances from q to t
Procedure getUBs(q, t, k )
  use 2-landmark TI to calculate k upper bounds,
  with the jth being the upper bound of the jth
  shortest point-to-group max distances from
  cluster q to cluster t.

// group-level filtering
Procedure groupFilter(UB, q, C)
1. V = {};
2. foreach $c_j$ in C
3.     l = getLB(q, $c_j$);
4. if (l < UB) // close enough
5.     V = V∪{$c_j$};
6. return V;

Procedure getLB(q, t)
  use 2-landmark TI to calculate the
  lower bound of group-to-group min
  distances from cluster q and cluster t.

// point-level filtering
// q: a cluster of query points
// S: a set of target point groups; points in each group are decreasingly ordered on point-to-center distances
Procedure pointFilter(S, q, UB)
1. S.sort(); // increasingly sort groups in S based on the distances from their centers to the center of c
2. foreach point q in q
3.     u = UB;
4.     foreach cluster e in S
5.         foreach point t in e
6.             l= d(q, ec) - d(t, ec); // ec is center of e;
7.             if (| l | ≤ u)  // a point possibly close enough
8.                 calculate distance to see whether t is indeed close enough;
9.                 if so, update u and the nearest neighbors of q with the kth nearest distances of q;
10.            else if (l > u)
11.                break; // no other points in e can be close enough due to the order of the points in e
12. return the collection of the k nearest neighbors of every query point.
```

Fig. 4. Pseudo-code of the basic TI-based KNN.

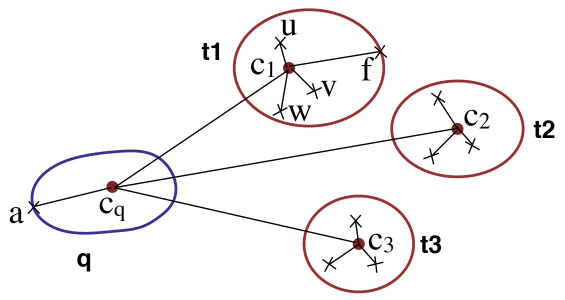


Fig. 5. Illustration of the use of TI by *getUBs()* and *getLB()* in Figure 4.

increase monotonically), and can hence be safely skipped from checking. (Note l can be negative; if $l < -UB$, the remaining points of the target cluster still need to be checked.) UB gets updated (tightened) throughout the process.

III. Basic Implementation of TI-Based KNN on GPU

Previous work [4] has shown that TI-based KNN can avoid a large portion of distance calculations, outperforming the standard KNN by orders of magnitude. But making it work efficiently on GPU is challenging for all the condition checks and irregular memory accesses the 2-level filtering incurs.

Our exploration starts with a basic implementation of the TI-based KNN on GPU, which is presented in this section. Next section will present our optimizations for reconciling the redundancy removal by TI and the irregularities it introduces. The description in this section follows the three steps outlined in the previous section.

A. Step 1: Initialize Clusters

Recall that this step is to create some landmarks for both the query and target sets, and assign each point to its closest landmark to form a number of clusters.

In our GPU implementation, the landmark generation follows the same algorithm as in the previous work [4]. For the query dataset, we create a GPU kernel to randomly generate $3 * \sqrt{n}$ candidate landmarks for a dataset containing n points and compute the sum S of all the pair-wise distances among these landmarks. The kernel repeats the process for several (empirically we find that 10 strikes a good tradeoff between the overhead and the clustering quality) times and choose the set of candidates that have the largest sum S as the landmarks to use.

To find the closest cluster center for each query point, $|Q|$ threads are created with each working on one query point. Each thread calculates the distances between the query point and every cluster center, and assigns the query point to the closest cluster center CQ_i. It also updates the maximal distance in this cluster CQ_i by using a user-defined floating-point atomic operation on GPU.

The work for the target clusters is more involved as it requires sorting the distances. The work consists of two main tasks: recognizing the members of each cluster and putting them into a container, and then sorting each container based on the distances from its members to the center of that cluster. A complexity is that the numbers of members in the clusters are unknown before the clustering, and could differ much between different clusters. Using dynamic data structures as the containers can circumvent the problem but would incur large overhead. In our implementation, we address the issue through two kernel calls. The first kernel creates $|T|$ threads with each calculating the distances between a target point and all cluster centers, finding the closest center, and recording the distance. It increases the corresponding cluster size by 1, and uses that size as the *local ID* of this target point in that cluster. By the end of the kernel, the size of every cluster becomes clear. The following CPU code then allocates a GPU array for each cluster accordingly. The second kernel then goes through all target points and puts each into the array of its cluster. The *local IDs* recorded in the first kernel helps avoid synchronizations in this step: For a target point with *local ID* equaling i, a thread just needs to put it into the ith location in the array of its cluster. Without *local IDs*, one may append a target point to the end of a list, which would require atomic operations because other threads may try to append other points to the end of the same list at the same time. The avoidance of synchronizations makes the implementation efficient.

B. Step 2: Choose Candidate Clusters (level-1 filtering)

As the previous section mentions, this step contains two substeps. The amount of parallelism of the two substeps differs.

In substep 1, to estimate the UB of the k^{th} nearest neighbor of each query cluster, we need to go through all target clusters. Due to the data dependencies involving the update to UB of a query cluster, we create $|CQ|$ threads with each working on one query cluster. (We use $|CQ|$ and $|CT|$ to represent the number of query clusters and the number of target clusters respectively.)

Substep 2 is to filter some clusters by comparing the lower bounds with the UB of the query points. Here the calculated lowerbound is for each pair of query cluster and target cluster. There is no data dependency between different pairs, so we create $|CQ| * |CT|$ threads with each working on one pair (CQ_i, CT_j). The pseudo code of filtering the clusters is shown in Algorithm 1.

Algorithm 1: KNN level-1 filtering on GPU

```
input : query clusters $CQ$, target clusters $CT$
output: candidate target clusters close to each query
        cluster

1 qc = getAQueryCluster();
2 tc = getATargetCluster();
3 qcDist = EuclDistance (CQ[qc].center, CT[tc].center); //
  distance between centers
4 if $qcDist$ - $CQ[qc].maxWithinDist$ -
  $CT[tc].maxWithinDist < CQ[qc].UB$ then
5     cnt = atomicAdd(&CQ[qc].candidatesCount, 1);
      CQ[qc].candidates[cnt] = {tc, qcDist};
```

C. Step 3: Point-level filtering (level-2 filtering)

Algorithm 2 shows the pseudo code of the GPU kernel for the level-2 filtering of TI-based KNN. Each GPU thread handles one query point (corresponding to one iteration of the loop at line 2 in Procedure `pointFiler`(**S**, **c**, UB) in Figure 4). The thread's ID (tid) is taken as the index of the query point in Q. It identifies the ID of the corresponding query cluster (cid) at line 2 in Algorithm 2. Recall that in Step 2, each query cluster already gets an upperbound of its k^{th} nearest neighbor distances. That upperbound (CQ[cid].UB) is used by the GPU thread as the initial upperbound for filtering (line 3 in Algorithm 2). The thread uses array $kNearests$ to track the k nearest neighbor distances of the query point. It is initially set to the k nearest neighbor distances of the query cluster (line 4 in Algorithm 2) and gets refined in the following loop (line 5).

Algorithm 2: KNN Level-2 Filtering Algorithm

```
input : $Q$ (query dataset), $T$ (target dataset), $CQ$, $CT$, $k$
output: $k$ nearest neighbors of each query point

1  tid = get_thread_id();
2  cid = $Q$[tid].$clusterID$; // cluster of the query
3  $\theta$ = $CQ$[cid].$UB$; //upperbound of the cluster
4  $kNearests$ = $CQ$[cid].$kUBs$; //upperbounds of the cluster
5  for $tc$ = 0 to $CQ[cid].candidateTargetClusters$ do
6      $q2tc$ = Edistance(Q[tid].point, CT[tc].center);
7      for $t$ = 0 to $CT[tc].membersize$ do
8          //apply triangular inequality
9          $q2t\_lb$ = $q2tc$ - $CT$[tc].member[t].distFromCenter;
10         if $q2t\_lb > \theta$ then
11             break;
12         else if $q2t\_lb < -1.0 * \theta$ then
13             continue;
14         else
15             q2t = Edistance(Q[tid].point,t);
16             if $q2t < kNearests$.max, evict $kNearests$.max,
               and put q2t into $kNearests$.
17             $\theta$ = $kNearests$.max;
18         end
19     end
20 end
```

Each iteration of that loop examines one candidate target cluster. Line 6 calculates the distance from the query to the target cluster center (function "Edistance" calculates the distances between two points). The loop at line 7 examines each point in the target cluster. It then applies 1-landmark TI: Line 9 gets the difference between the query-to-target center distance and the point-to-center distance of the target point. Lines 10 to 18 compares the result with the upperbound (θ) to determine whether it is necessary to calculate the distance between the query and the target. Only if $|q2t\_lb| < \theta$ (line 14), it is necessary. After getting that distance, if it is smaller than the current k^{th} distance (i.e., $kNearests$.max in Algorithm 2), it removes the k^{th} distance from $kNearests$, and puts this new distance into it. It then updates the upper bound with the new $kNearests$.max.

For easy understanding, our explanation assumes that each thread has its own $kNearests$ array. In our implementation, they are actually put together into one big array, as illustrated in Figure 6 (a) and (b). These two graphs illustrate two possible memory layouts of $kNearests$. Our experiments show that the second layout gives better performance than the first. The reason is that as the threads in a warp go through their respective part of the array, they access consecutive locations in the second case but not in the first case. Hence the second layout gives more coalesced memory accesses, which is hence used in our basic implementation.

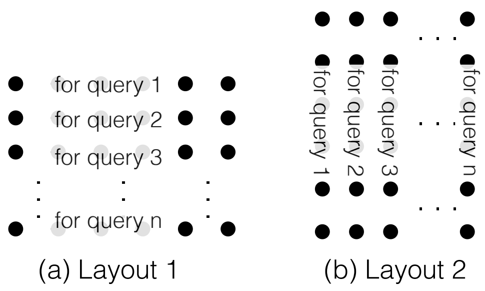


Fig. 6. Memory layouts for $kNearests$ (locations on one row are consecutive on memory).

IV. Sweet KNN

This section presents Sweet KNN. Sweet KNN builds upon the basic TI-based KNN design described in the previous section. It overcomes its limitations by optimizing the design in three aspects: making some key algorithmic design elastic, minimizing the impact of irregularities TI introduced by carefully matching the computations with GPU characteristics, and creating an adaptive scheme to automatically tailor the algorithm configurations on the fly for each problem instance. These optimizations help Sweet KNN strike a sweet point in the tradeoff between redundancy removal and regularity preservation.

A. Overview

The benefits of applying triangle inequality to KNN mainly come from two parts. First, it can reduce unnecessary distance calculations. When the dimensions of points are large, the cost of distance calculation is high. With triangle inequality, the benefits can be substantial. Second, in the prior KNN implementations on GPU that use CUBLAS [13], [15], the distance between each query point and each target point is stored in global memory. With triangular inequality, only distances between each query cluster and each target cluster and the point-to-center distances within each cluster need to be stored in memory, reducing the memory usage and the overall access latency.

However, the basic implementation described in the previous section sometimes even runs slower than the prior CUBLAS-based implementations (detailed in Section V). The primary reason is the irregularities introduced into the computations by the applications of triangular inequality. For instance, each level of TI-based filtering involves some condition checks, on which, threads in a warp could diverge. At level-1 filtering, the divergences could happen when different query clusters have different sets of candidate target clusters; at level-2 filtering, the divergences could happen when different queries have different updates to $kNearests$ array and differ in the comparisons with the upperbound. Similarly, the disparity in candidate clusters and candidate neighbors of the different queries could cause threads of a warp to access memory locations distant from one another, causing many none-coalesced memory accesses. In comparison, the prior CUBLAS-based implementation computes the distances from all queries to all targets and have a simple consistent computation pattern for all threads, much more GPU-friendly.

This section describes a set of techniques Sweet KNN uses to reconcile the redundancy minimization with its side effects on computation regularity. The key is two-fold: to make the design elastic such that the tradeoff between redundancy and parallelism can be adjusted on the fly through an adaptive scheme; to make the implementation better match the performance characteristics of GPU. As the adaptive scheme adjusts configurations at the levels of both algorithm and implementation, we postpone its description to the end of this section. We will first describe the enabled algorithm-level elasticity, and the set of implementation-level changes.

B. Enabling Algorithmic Elasticity

The elasticity we introduced into Sweet KNN is mainly on the 2-level filtering and the algorithmic parallelism, two most critical aspects on the performance.

1) Filter Design: The first aspect we make elastic is the strength of the level-2 filter.

The two levels of filtering in TI-based KNN involves many condition checks, updates to the upperbounds, and frequent accesses to the *kNearests* array. Our measurements show that when k is modest, the filtering benefit outweighs the negative performance impact of these complexities. But when k is large, the *kNearests* array becomes large, and updating it incurs lots of overhead. Also, each thread has more possible locations to access, hence more opportunities for threads to show disparities in behaviors.

The essence of our idea is to reduce the differences among threads and avoid *kNearests* update overhead by simplifying the filter design when k gets large. Specifically, we let the level-2 filtering use the UB obtained from the level-1 filtering without further updating it, and avoid using *kNearests* at all. All results calculated at line 15 in Algorithm 2 are stored into global memory, from which, a later launched GPU kernel finds the k minimal distances. We call this weakened filtering *partial level-2 filtering* and the original *full level-2 filtering*.

The design of the weakened filtering has two benefits. First, it directly reduces the number of memory accesses to these data structures, which are typically non-coalesced accesses. Second, the reduced filtering strength reduces the divergences of threads in the filtering checks. Our experiments (detailed in Section V) show that most distance computations could still be saved even with the weakened level-2 filtering.

When k is modest or small, the full filtering should be used as the partial filtering would leave some potential untapped. The details on choosing the strength of the level-2 filtering will be presented in Section IV-D.

2) Parallelism: Besides filter design, parallelism is a second dimension that we find important to be made adjustable. In our basic implementation, each GPU thread handles one query point. The amount of parallelism is determined by the number of query points. When the number of query points is modest, the parallelism could be insufficient to take the full advantage of the GPU computing resource.

We hence extend the basic KNN level-2 algorithm to make parallelism elastic. Besides the query-level parallelism, we also exploit the parallelism in the loops at lines 5 and 7 in

TABLE I. AN EXAMPLE OF WARP DIVERGENCE FOR KNN WITHOUT OPT

threadID	QpointID	QclusterID	Candidate ID
0	0	4	10, 8, 5, 1
1	1	5	9, 7, 3
...	...	...	...
100	100	4	10, 8, 5, 1
101	101	5	9, 7, 3
...	...	...	...
365	365	4	10, 8, 5, 1
366	366	5	9, 7, 3
...	...	...	...

TABLE II. AN EXAMPLE OF WARP DIVERGENCE FOR KNN WITH MAP

threadID	QpointID	QclusterID	Candidate ID
0	0	4	10, 8, 5, 1
1	100	4	10, 8, 5, 1
2	365	4	10, 8, 5, 1
...	...	...	...
64	1	5	9, 7, 3
65	101	5	9, 7, 3
66	366	5	9, 7, 3
...	...	...	...

Algorithm 2, by allowing multiple threads to work for one query point concurrently when the query-level parallelism is insufficient.

We use a lightweight runtime model to automatically determine the number of threads to create and how the iterations of the two-level nested loop shall be assigned to the threads. We describe the model when we explain our adaptive scheme in Section IV-D3.

With multiple threads processing candidates in parallel, updating the heap storing k-nearest neighbors may cause race conditions (if the full filtering is used). To solve it, we make each thread operate on its own local heap which stores the k-nearest neighbors the thread has seen so far. The upper bound θ in Algorithm 2 is kept shared among the threads working on the same query point, they use *atomicMin* to update θ. After the kernel finishes, for one query point, there will be a number of sorted heaps where each stores k nearest candidates. The final step is to launch $|Q|$ threads with each thread working to select the k minimal value from all the sorted heaps related with one query point. As each heap has been sorted, a technique similar to the one in merge sort is used in this step.

C. Implementation-Level Optimizations

At the implementation level, we develop three main optimizations to further mitigate the effects of the computation irregularities caused by the TI-optimizations.

1) Thread-Data Remapping: Thread-data remapping is a way to reduce thread divergences [18]. The basic idea is to adjust the assignments of tasks to threads such that threads in a warp, after the reassignment, could have no or minimum divergences. We implement this idea in Sweet KNN to further reduce the influence of the irregularities in computations.

We focus on the divergences at lines 5-20 in Algorithm 2 for its seriousness. Our basic TI-based KNN uses an intuitive way to process query points: Thread i handles the i^{th} query point. It works well for the sequential version on CPU [4], but causes many warp divergences on GPU.

Table I shows an example. Threads 0, 100 and 365, for instance, will need to examine the same candidates clusters (10,8,5,1) as they work for query point 0, 100, 365 respectively, which happen to be in the same query cluster. However, these threads are not in the same warp. Thread 1 is in the same warp as thread 0, whose candidate clusters that need to examine are totally different. As a result, warp divergences happen, leading to not only serialization in execution but also poor memory performance.

To address the issue, Sweet KNN creates a map between thread IDs and query point IDs that the thread will work for. The map is constructed such that to the largest degree, threads in the same warp work for query points in the same cluster and they iterate over the similar sets of candidate clusters in the same order. Table II illustrates the new map between threads and query points in our example. (The information of candidates—such as membersID, dist2cluster—can be shared among threads in the same warp.)

For the basic KNN, each query cluster only needs to record the maximum point-to-center distance. To create such a mapping, Sweet KNN records query members for each query cluster during initialization of clusters. Each query cluster copies its member IDs to a continuous segment of the map where the starting address is attained through the use of the atomic function in GPU *atomicAdd(&start_addr, memberSize)*.

2) Data Placement: GPU has various types of memory of different attributes as Section II mentions. Previous studies have shown that placing data onto the appropriate types of memory could have some large influence on GPU program performance. The best placement however is determined by many factors: the access patterns to the data, the data size, the effects of the placement on the overall GPU resource pressure, and so on.

We build data placement optimization into Sweet KNN. Particularly, we focus on the placement of the *kNearests* array(s) on memory. As we have seen, the array is frequently updated and read throughout the TI-based KNN. Its placement hence has some substantial influence on the overall performance.

Because the array is not read-only, it cannot be put onto some read-only types of memory (e.g., constant memory). Three options are valid: global memory, shared memory, and registers. Each has its pros and cons. For example, global memory has the largest memory size but has the longest access latency. Shared memory has a much limited size, but as it is on-chip memory, it is much faster to access than global memory is. Register file is the fastest to access among the three. Its size is also limited. If more registers are needed for a kernel, the registers may get spilled into L1 cache, L2 cache or global memory [19], causing a lot of overhead. Moreover, too much usage of shared memory and registers per thread could result in a situation where only a small number of threads could get deployed on GPU due to the limited overall resource on GPU. If the usage of registers are too large, it would be better to use global memory instead of registers.

Because of the dependence of the appropriate placements on the problem size and many other runtime factors, we develop a module in our adaptive scheme to decide the placement

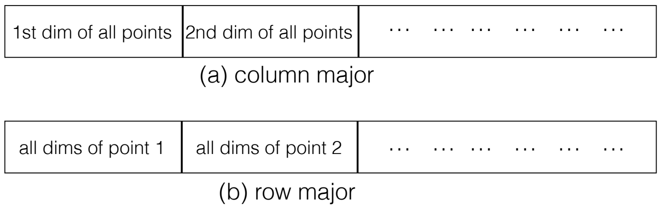


Fig. 7. Memory layouts of data points.

at runtime. The details are given when we explain the adaptive scheme in Section IV-D.

3) Data Layout: Previous implementations of the basic KNN on GPU use a column-major data layout format to store all query and target points in order to make memory accesses coalesced. The layout is shown in Figure 7 (a). That layout fits the computation patterns of the basic KNN well as threads in a warp need to simultaneously access the same dimension of different data points. The layout puts the same dimension of different points together, creating coalesced accesses.

That layout however does not work well for TI-based KNN because TI-based KNN avoid most distance computations. Their accesses to the data points tend to have some irregular strides. Our experiments show that the row-major layout illustrated in Figure 7 (b) can actually better fit the needs of TI-based KNN. In addition, to maximize the bandwidth efficiency, we use vector loading(float4) for each read.

D. Adaptive Scheme

This subsection describes the adaptive scheme that we have developed in Sweet KNN. By customizing the TI-based KNN algorithm and implementation on the fly to best fit the data sets, the adaptive scheme is a key for Sweet KNN to strike a good balance between redundancy elimination by TI and the regularity preservation of computations for GPU.

There are many factors that could affect the performance of TI-based KNN:

1) Some are the factors associated with KNN problem itself (e.g. query dataset $\mathbf{Q}$, target dataset $\mathbf{T}$, number of nearest neighbors to find k, data dimensions d, w/ or w/o index information for the results).
2) Some are about the GPU hardware (e.g., memory size, number of SMs, registers, etc.).
3) Some are parameters of the triangle inequality-based optimizations (e.g., # of clusters, parameters in the two-level filtering).

The influence of these factors are often coupled with one another. For instance, the number of points and GPU memory size limit the number of clusters that can be created to filter target points. The adaptive scheme in Sweet KNN tries to consider the most important factors to make practically appropriate decisions. As a scheme coded into Sweet KNN, it runs when Sweet KNN is invoked and quickly configures Sweet KNN based on the current dataset. It is hence important to make the scheme as lightweight as possible.

Figure 8 outlines the selection of three main aspects of our design. (If query datasets are partitioned to fit into GPU

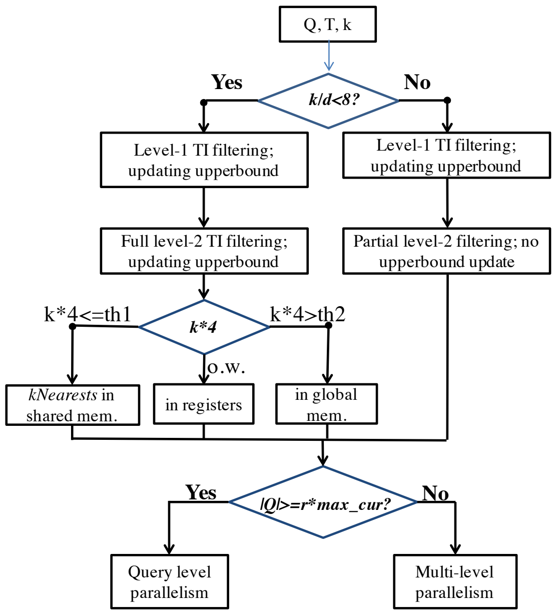

Q, T, k
k/d<8?
Yes
No
Level-1 TI filtering; updating upperbound
Level-1 TI filtering; updating upperbound
Full level-2 TI filtering; updating upperbound
Partial level-2 filtering; no upperbound update
k*4<=th1
k*4
k*4>th2
o.w.
kNearests in shared mem.
in registers
in global mem.
Yes
|Q|>=r*max_cur?
No
Query level parallelism
Multi-level parallelism

Fig. 8. Adaptive scheme used in Sweet KNN (th1 and th2 are two thresholds described in Section IV-C2.) Q: query dataset; T: target dataset; k: number of nearest neighbors to find; d: data dimension.

memory, Q in Figure 8 represents one of the partitions.) Based on the ratio between k and d (the data dimension), it decides whether the full 2-level filtering or the partial 2-level filtering (i.e., the one without frequent update of upperbounds) shall be used. As the full filtering needs to create local *kNearests* for each thread, the algorithm decides the placement of these local data structures based on the needed size ($sizeof(float)* k$) according to the discussions in Section IV-C2. Finally, it decides whether fine-grained (inside-query) parallelism shall be used. We explain each of them further as follows.

1) Filtering Strength and Number of Landmarks: Two most time-consuming parts in KNN are 1) computing distances, and 2) selecting k minimal distances for each query point. When *k/d* exceeds a certain level, the second part becomes more important than the first one. Using the partial filtering can help reduce the overhead of the second part. Based on empirical observations, we find that the scenarios for the partial filtering to outperform the full filtering is when $k/d > 8$ (the top part in Figure 8).

The number of landmarks determines the number of clusters to be formed for the TI-based filtering to function. A good setting helps eliminate 99% unnecessary distance calculations [3], [4]. According to a previous work [3], we set the number to $3\sqrt{N}$ for N points if the memory is large enough. If the memory is insufficient, Sweet KNN uses the whole global memory size as the budget to calculate the largest number of clusters allowed.

2) Data Placement: When the full filtering is used, each thread has a local *kNearests* array to store the k shortest neighbor distances that the thread has seen so far. As Section IV-C2 mentions, the placement of the array is important for performance. We develop a simple mechanism to decide the appropriate placements on a given problem instance. Through query APIs, the mechanism first gets the hardware information (shared mem size, number of registers). It then sets the first threshold to $th1 = shared\_mem\_size/max\_currPerSM$, where, $shared\_mem\_size$ represents the size of shared memory on one stream multiprocessor (SM) on GPU, and $max\_currPerSM$ represents the maximal number of threads that can run concurrently on an SM. Only if the size of one *kNearests* array is no greater than $th1$, it is considered for shared memory. To determine whether we should place *kNearests* onto register files, we set the second threshold to $th2 = max\_regPerThread * 4$ Bytes ($th2$ is greater than $th1$ on GPU because of the larger size of registers than shared memory). Here, $max\_regPerThread$ represents the maximal number of registers a thread can use. Only if *kNearests* is less than $th2$ (and greater than $th1$), it is declared as a local variable in the kernel such that it could possibly be placed into registers.

This design gives a higher priority to shared memory over register files. The rationale is that due to the usage of registers in the other parts of the kernel, the register file is more likely to be the resource limiting the number of threads per SM, while the shared memory is not used for any other data structures.

On Kepler GPU, for instance, the shared memory size is 48KB per SM, and the register file size on one SM is 64K. If the maximal number of threads can run concurrently on an SM is 2048, the two thresholds we get are $th1 = 24$ and $th2 = 1024$.

3) Parallelism: With the parallelism made adjustable, Sweet KNN may depend on the problem instance to decide whether to leverage only the cross-query parallelism or both the cross-query and inside-query parallelism. And in the latter case, how many threads to use for each query point and at which level of parallelism each thread shall work are also subject to runtime adaptation.

The adaptive scheme in Sweet KNN makes these decisions based on the GPU hardware limit and the properties of the problem instance. The total number of threads is set to ($r * max\_cur$), where, max_cur is the maximum number of threads that can be concurrently active on the GPU, and r is a *cache conflict factor*. The value of max_cur can be automatically calculated according to the GPU hardware properties (shared memory size, register file size, etc.) and the amount of shared memory and register usage in the GPU kernel [20]. Previous work [21] has shown that for memory-access intensive programs, using the maximum concurrency on GPU often causes serious conflicts in GPU cache accesses and hence low throughput. Our observation on KNN echoes it. We use r to factor in that consideration. Our empirical study shows that $r = 0.25$ consistently works well for various settings of KNN.

For $|Q|$ query points, when $|Q|$ is greater or equal to $r * max\_cur$, only query-level parallelism is exploited as it is sufficient to keep the GPU busy. Otherwise, there will be $\frac{r*max\_cur}{|Q|}$ threads working for each query point. We consider that the member size of each cluster is roughly $\frac{|T|}{|CT|}$ (recall, $|CT|$ is the number of target clusters). So the inner loop (line 7 in Algorithm 2) is parallelized by a factor of $\frac{|T|}{|CT|}$ while the outer loop (line 5 in Algorithm 2) is parallelized by a factor of $\frac{r*max\_cur}{|Q|} / \frac{|T|}{|CT|}$.

TABLE III. DATASETS FROM UCI

Data Set	Full name	Number of Points	Dimension
3DNet	3D spatial network	434874	4
kegg	KEGG Metabolic Reaction Network (Undirected)	65554	29
keggD	KEGG Metabolic Reaction Network (Directed)	53414	24
ipums	IPUMS Census Database	256932	61
skin	Skin Segmentation	245057	4
arcene	Arcene	100	10000
kdd	KDD Cup 1999 Data	4000000	42
dor	Dorothea Data	1950	100000
blog	Blog Feedback	60021	281

For the practical deployment of the adaptive scheme, we implement multiple versions of the relevant GPU kernels and insert dynamic checks into Sweet KNN to choose the appropriate versions to invoke at runtime.

V. EVALUATIONS

In this section, we report some experimental results we obtained when comparing the efficiency of Sweet KNN with other alternatives. The results are promising: We observe up to 120X speedups over the state-of-art GPU implementation of KNN (k=1 on a 3D spatial network dataset) with an average speedup as much as 11X. Our experiments cover a set of different input datasets. Besides reporting the speeds, we also analyze the impact of different settings of parameters on KNN.

A. Methodology

Our experiments use a system equipped with Intel Xeon E5-1607v2 processors and an NVIDIA K20c Kepler GPU. CUDA 7.5 is used. Table III shows 9 data sets from the UCI machine learning repository [22]. We select data sets by following 3 rules: (a) the value of each attribute should be numeric; (b) the datasets should cover a good spectrum of sizes; (c) the dimensions of the points in the datasets should cover a wide range as well. Without noting otherwise, the same dataset is used as both the query and target dataset in all the experiments, and the number of nearest neighbors to find (k) is set to 20. (Section V-C1 studies the sensitivity on different k values.)

The state-of-the-art GPU-based KNNs that are publically available employ one of two ways in their implementations: purely using CUDA or leveraging the matrix-matrix multiplication routine in some high-performance GPU library such as CUBLAS [16]. Our survey finds that the CUBLAS-based KNN in a package by Garcia and others [13], [15] gives the best performance, outperforming other CUDA-based implementations by up to 10X. We hence downloaded the code of that version [23] as the baseline in our comparisons.

That baseline version uses a two-stage scheme in the implementation. First, it uses a CUBLAS-based GPU kernel to compute all the distances from every query point to every

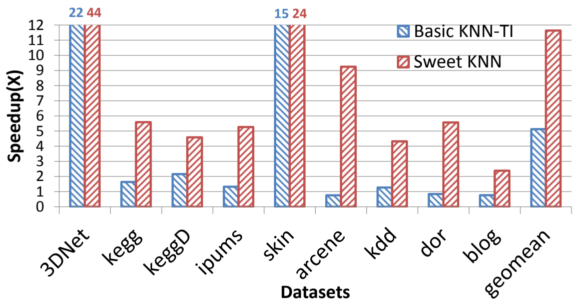


Fig. 9. Overall speedups over the CUBLAS-based basic KNN on GPU.

target point, and stores these distances on the GPU global memory. Then, it launches a second GPU kernel, with each GPU thread sorting the distances of a query point and finding the k target points that have the minimal distances to the query point. If the dataset is too large for the GPU memory, the basic KNN partitions the query dataset such that the memory usage of each partition can fit into the GPU memory.

When running our method, we use 256 as the thread block size. Tuning the size could possibly lead to even better performance; but our following results show that even without tuning, our method already outperforms the state of art significantly.

B. *Overall Performance*

Figure 9 shows the overall speedups over the CUBLAS-based basic KNN. The KNN-TI results are the performance of the basic implementations of triangular inequality-based KNN on GPU (i.e., the version in Section III). The Sweet KNN results are the performance of the TI-based KNN with the optimizations and adaptive algorithm in Section IV applied upon the basic KNN-TI implementation.

The calculations of the speedups have considered all the overhead (e.g., the preprocessing to form clusters and queries and targets). Each experiment is repeated for a number of times. As we did not observe any substantial fluctuations, we report the average results.

As we can see, the basic KNN-TI can provide on average 5X speedups over the baseline. It shows slight slowdowns on some datasets (arcene, dor, and blog). Sweet KNN-TI improves the performance of the basic KNN-TI version significantly. It outperforms the baseline substantially on every dataset, showing on average 11.5X speedups.

To help explain the speedups, Table IV shows the profiling details of the level-2 filtering algorithm (Algorithm 2) of the basic KNN-TI and Sweet KNN. We add a profiling variable to count every distance calculation. The saved computations is calculated as $(|Q| * |T| - count)/(|Q| * |T|)$. Warp efficiency is obtained from the NVIDIA hardware profiling tool (nvprof) [24], defined as the ratio of the average active threads per warp to the maximum number of threads per warp supported on a GPU multiprocessor. It characterizes the degree of utilization of the GPU cores.

TABLE IV. PERFORMANCE DETAILS OF KNN-TI AND SWEET KNN.

Data Set	Basic KNN-TI		Sweet KNN	
	saved comp.	warp effi.	saved comp.	warp effi.
3DNet	99.7%	16.3%	99.7%	29.4%
kegg	99.5%	8.7%	99.5%	42.4%
keggD	99.5%	10.1%	99.5%	35.5%
ipums	99.4%	11.8%	99.4%	33.3%
skin	99.7%	19.6%	99.7%	41.2%
arcene	26.9%	59.5%	1.82%	89.8%
kdd	99.6%	7.1%	99.6%	57.4%
dor	91.5%	20.9%	70.1%	78.6%
blog	99.5%	21.2%	99.5%	35.3%

As the results show, for datasets other than $arcene$ and dor, triangle inequality can save more than 99% distance computations. Sweet KNN has on average a 3X higher warp efficiency than KNN-TI, thanks to the reductions of thread divergences and non-coalesced memory accesses and the increases of parallelism brought by the optimizations in Sweet KNN.

On datasets $3DNet$ and $Skin$, both KNN-TI and Sweet KNN achieve the most significant speedups among all datasets. Because the baseline version of KNN computes and stores the distances between every query and every target point, the total memory space needed for it to handle either of the two datasets exceed the amount of memory on the GPU. The baseline KNN partitions the query points into a number of groups (e.g., 175 groups for $3DNet$) such that the memory can hold the results of each group. It processes these groups one by one. The GPU thread occupancy in the processing of each group is relatively low, while the amount of memory accesses is tremendous. Both KNN-TI and Sweet KNN avoid 99.7% of the distance calculations. That brings two other benefits: They can avoid most of the memory accesses and stores needed by the basic KNN, and fit the processing of more query points onto GPU in one kernel execution and hence more parallelism. Sweet KNN exploits even more parallelism for its multiple levels of parallelizations. It uses 434874 and 245057 threads to work for each query point in one kernel execution for 3DNet and SKin respectively. With all these benefits, Sweet KNN outperforms the baseline KNN by 44X and 24X on the two datasets respectively.

The size of the datasets $kegg$, $keggD$, and $blog$ are not as large as $3DNet$ and $skin$. KNN-TI does not show much speedup over the baseline KNN and even a slight slowdown on $blog$. Even though it still avoids over 99% distance computations, as Table IV shows, it has very low warp efficiencies on these datasets due to the irregular control flows and memory accesses the TI-optimizations incur and the limited parallelism in the smaller datasets. Sweet KNN boosts the warp efficiency substantially and improves the performance significantly: 5.7X versus 1.7X on $kegg$, 4.6X versus 2.1X on $keggD$, and 0.85X versus 2.3X.

Datasets $ipums$ and kdd have many query points and also relatively higher dimensions. The needed memory for processing all the query points exceed the GPU memory in all the three versions of KNN. They all partition the query datasets and process each partition each time. KNN-TI avoids

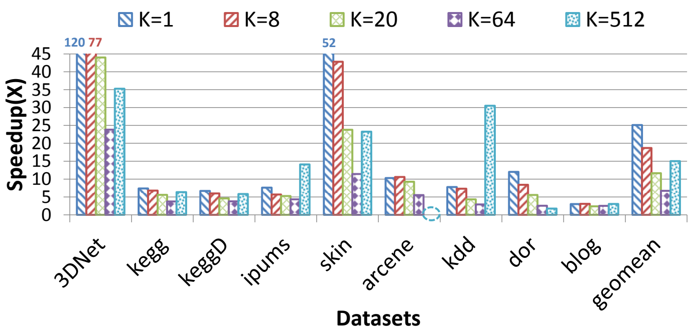


Fig. 10. Speedups of Sweet KNN on different k values (*arcene* has only 100 points and hence does not have results at $k = 512$).

over 99% distance calculations but is subject to less than 12% warp efficiency. It gives only about 1.2X speedups. Sweet KNN enhances warp efficiency to 33% and 57% respectively, yielding 5.2X and 4.2X speedups over the baseline KNN.

Datasets $arcene$ and dor have small numbers (100 and 1950) of query points, but each point is of high (10000 and 100000) dimensions. Because of the small number of points, the amount of unnecessary distance calculations is much smaller than on other datasets. KNN-TI avoids 26.9% and 91.5% distance calculations respectively, showing slight slowdowns over the baseline KNN. The adaptive scheme in Sweet KNN automatically chooses the reduced strength of filtering in these two cases. It avoids 1.82% and 70.1% distance calculations on them. However, because its adaptive scheme chooses to use many threads to process one query point for the limited parallelism at the levels of query points, it achieves much higher warp efficiencies (89.8% and 78.6% versus 59.5% and 20.9% in KNN-TI), and yields 9.2X and 5.6X speedups over the baseline KNN.

We next report a series of sensitivity studies of Sweet KNN by varying the problem settings.

C. *Sensitivity Study*

When studying the impact of one parameter, we fix other parameters' values.

1) Value of k: Here, k is the number of nearest neighbors the KNN tries to find for each query point. As k increases, the time spent on updating the local *kNearests* array increases. In this study, we increase k from 1 to 8, 20, 64, and 512.

Figure 10 shows the speedups of Sweet KNN at these different k values. Because $arcene$ has only 100 data points, it does not have $k = 512$ results. Overall, the speedups by Sweet KNN on the datasets decrease as k increases from 1 to 64, and then increases as k gets to 512.

The reasons for the decreasing part are as follows. The adaptive scheme in Sweet KNN selects the full 2-level filtering for all data sets when $k <= 64$ (except 3DNet when $k = 64$). As k becomes larger, $kNearests$ gets larger. Threads in a warp have more locations to access and hence more probabilities for them to diverge in both control flows and memory accesses. Secondly, that also increases the time for Sweet KNN to access and update $kNearests$.

TABLE V. PERFORMANCE OF SWEET KNN ON DATASETS WITH $k/d > 8$ WHEN k=512 (SWEET KNN CHOOSES THE PARTIAL FILTER IN THESE CASES)

Datasets		3DNet	kegg	keggD	ipums	skin	kdd
full filter	saved comp	99%	98%	98%	98%	99%	99%
	spdup(X)	23.5	1.3	2.7	10.9	10.3	5.9
partial filter	saved comp	96%	97%	97%	95%	96%	98%
	spdup(X)	35.3	6.3	5.8	14.1	23.2	30.5

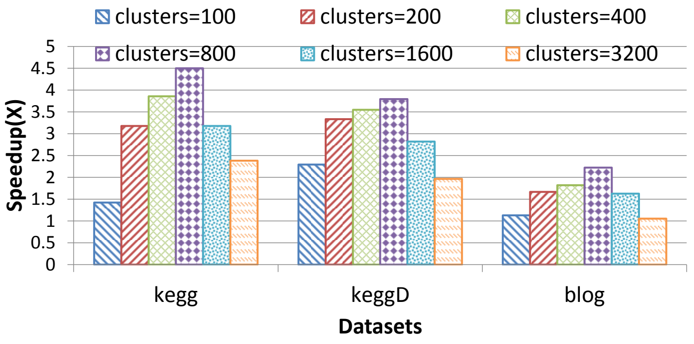


Fig. 11. Speedups by Sweet KNN on different # of landmarks (i.e. clusters).

While as k gets to 512, the adaptive scheme selects the reduced filtering strength for Sweet KNN to work on datasets *3DNet, kegg, keggD, ipums, skin*, and *kdd*, because k/d of these datasets are greater than 8. The filtering still remains quite effective. As Table V shows, the avoided distance computations are 95-98%, only up to 3% less than those avoided by the full filtering. On the other hand, the reduced filtering strength helps significantly improve the regularity in the computations and hence the warp efficiency, helping Sweet KNN produce significantly larger speedups (5.8X–35.3X) than what it would have produced if it still uses the full filtering, as Table V shows. The results indicate the efficacy of the adaptive scheme in helping Sweet KNN strike a good balance between minimizing redundancy and maintaining regularity.

2) Number of Landmarks: The number of landmarks determines the number of query clusters and target clusters. To examine the effectiveness of our method in choosing the right number of landmarks, we experiment with a spectrum of the values to see the effects. As the datasets *kegg, keggD* and *blog* have similar numbers of points but different dimensions, we focus on those three datasets and try 6 different numbers of landmarks.

Figure 11 shows the impact. The number of points is around 60000 in each of the datasets. According to Section IV-D, our implementation selects $3\sqrt{60000}$, which is 745, as the number of landmarks. As we can see from Figure 11, the performance of KNN-TI get improved when # of clusters increases from 100 to 800. But as the number of clusters increases further, the performance drops due to the overhead of unnecessary clusters and the clustering overhead. It offers an evidence on the efficacy of the selection method implemented in Sweet KNN.

3) Parallelism: When the number of query points is small, Sweet KNN uses multiple threads for processing one query point concurrently, as Section IV-B2 describes. We validate the appropriateness of the choices of the numbers of threads per query point by measuring the performance when some other

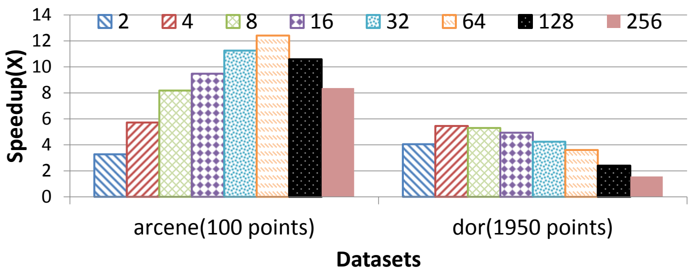


Fig. 12. Speedups by Sweet KNN on different # of threads for one query point.

numbers are used. We experiment on the datasets with only a small number of points: *arcene* and *dor*.

Figure 12 shows the observed speedups by Sweet KNN in the different settings. For *arcene*, the setting that our method chooses is 2048*13/(4*100) = 66 threads per query point. As we can see, when the setting increases from 2 to 64, the performance increases and reaches optimal when the setting is around 64. After 64, the performance drops due to the increased merge overhead and the much reduced strength of filtering.

The setting our method chooses for *dor* is: (2048*13)/(4*1950)=3.4 (rounded to 4). We also observe a peak performance around that setting. These results confirm the effectiveness of the method we use for selecting the appropriate parallelism levels.

VI. Conclusion

The computing efficiency of KNN is essential. This paper presents the design and implementation of Sweet KNN, a high-performance triangular-inequality-based KNN on GPU. Experiments on a set of datasets show that Sweet KNN consistently outperforms the state of the art of KNN on GPU significantly, regardless of the k value, the size or dimensions of the dataset, or other properties of the problem instance. The speedup is as much as 120X, and 11X on average, dramatically advancing the state of the art of efficient KNN.

At a high level, this work shows a systematic way to make a data mining algorithm automatically strike a good balance between redundancy minimization and regularity preservation. It shows the effectiveness of three principles: making design elastic, adding adaptivity, and closely matching the implementation with the properties of the underlying platform (e.g., GPU). We expect that these principles can help advance the computing efficiency of many other data mining algorithms on modern massively parallel systems.

References

[1] X. Wu, V. Kumar, J. R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, S. Y. Philip *et al.*, "Top 10 algorithms in data mining," *Knowledge and Information Systems*, vol. 14, no. 1, pp. 1–37, 2008.

[2] B. Yao, F. Li, and P. Kumar, "K nearest neighbor queries and knn-joins in large relational databases (almost) for free," in *Data engineering (ICDE), 2010 IEEE 26th international conference on*. IEEE, 2010, pp. 4–15.

[3] X. Wang, "A fast exact k-nearest neighbors algorithm for high dimensional search using k-means clustering and triangle inequality," in *Neural Networks (IJCNN), The 2011 International Joint Conference on*. IEEE, 2011, pp. 1293–1299.

[4] Y. Ding, X. Shen, M. Musuvathi, and T. Mytkowicz, "TOP: A framework for enabling algorithmic optimizations for distance-related problems," *Proceedings of the VLDB Endowment*, vol. 8, no. 10, pp. 1046–1057, 2015.

[5] C. Li, Y. Gu, J. Qi, G. Yu, R. Zhang, and Q. Deng, "Insq: An influential neighbor set based moving knn query processing system," in *2016 IEEE international conference on data engineering (ICDE)*, 2016.

[6] W. Lu, Y. Shen, S. Chen, and B. C. Ooi, "Efficient processing of k nearest neighbor joins using mapreduce," *Proceedings of the VLDB Endowment*, vol. 5, no. 10, pp. 1016–1027, 2012.

[7] T. Emrich, F. Graf, H.-P. Kriegel, M. Schubert, and M. Thoma, "Optimizing all-nearest-neighbor queries with trigonometric pruning," in *Scientific and Statistical Database Management, Springer*, 2010, pp. 501–518.

[8] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Transactions on Mathematical Software (TOMS)*, vol. 3, no. 3, pp. 209–226, 1977.

[9] B. Merry, J. E. Gain, and P. Marais, "Accelerating kd-tree searches for all k-nearest neighbours." in *Eurographics (Short Papers)*, 2013, pp. 37–40.

[10] V. Ramasubramanian and K. K. Paliwal, "Fast k-dimensional tree algorithms for nearest neighbor search with application to vector quantization encoding," *Signal Processing, IEEE Transactions on*, vol. 40, no. 3, pp. 518–531, 1992.

[11] K. Hajebi, Y. Abbasi-Yadkori, H. Shahbazi, and H. Zhang, "Fast approximate nearest-neighbor search with k-nearest neighbor graph," in *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, vol. 22, no. 1, 2011, p. 1312.

[12] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu, "An optimal algorithm for approximate nearest neighbor searching fixed dimensions," *Journal of the ACM (JACM)*, vol. 45, no. 6, pp. 891–923, 1998.

[13] V. Garcia, E. Debreuve, and M. Barlaud, "Fast k nearest neighbor search using GPU," in *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*. IEEE, 2008, pp. 1–6.

[14] R. J. Barrientos, J. I. Gómez, C. Tenllado, M. P. Matias, and M. Marin, "knn query processing in metric spaces using GPUs," in *Euro-Par 2011 Parallel Processing*. Springer, 2011, pp. 380–392.

[15] V. Garcia, E. Debreuve, F. Nielsen, and M. Barlaud, "K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching," pp. 3757–3760, 2010.

[16] "CUBLAS," `http://developer.download.nvidia.com/`.

[17] W. Lu, Y. Shen, S. Chen, and B. C. Ooi, "Efficient processing of k nearest neighbor joins using mapreduce," *Proceedings of the VLDB Endowment*, vol. 5, no. 10, pp. 1016–1027, 2012.

[18] E. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen, "On-the-fly elimination of dynamic irregularities for GPU computing," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[19] "Register Spilling," http://on-demand.gputechconf.com/gtc-express/2011/presentations/register_spilling.pdf.

[20] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style GPU programming for GPGPU workloads," in *Innovative Parallel Computing (InPar), 2012*. IEEE, 2012, pp. 1–14.

[21] G. Chen, B. Wu, D. Li, and X. Shen, "PORPLE: An extensible optimizer for portable data placement on GPU," in *Proceedings of the 47th International Conference on Microarchitecture*, 2014.

[22] M. Lichman, "UCI machine learning repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml

[23] V. Garcia, E. Debreuve, and M. Barlaud, "KNN CUDA," http://vincentfpgarcia.github.io/kNN-CUDA/.

[24] "NVIDIA Visual Profiler," http://docs.nvidia.com/cuda/profiler-users-guide/#visual-profiler.