CLOVER: A GPU-native, Spatio-graph-based Approach to Exact kNN

Victor Kamel
University of Toronto
Toronto, ON, Canada
vkamel@cs.toronto.edu

Hanxueyu Yan
University of Victoria
Victoria, BC, Canada
hyan76131@uvic.ca

Sean Chester
University of Victoria
Victoria, BC, Canada
schester@uvic.ca

Abstract

Finding the k nearest neighbours (kNN) of every point in a dataset is a key primitive in many GPU applications. Unfortunately, algorithmic techniques for kNN do not translate well to GPUs, require (offline) preprocessing, sacrifice accuracy, or require low query volume. Recently, ray-tracing cores have been proposed to accelerate exact kNN, but it is not well understood how these compare to grid-based methods.

This work introduces a novel approach to exact kNN for spatial data that constructs and then traverses a graph from a random voronoi tesselation. On an NVIDIA V100, we answer ten million exact 30-NN queries with no prior preprocessing in 2.71s, about 4× faster than an optimised grid-based method, 10× faster than a GPU tree, and 230× faster than FAISS. Furthermore, we show on an RTX card that RT-core methods are uncompetitive when query volume is high.

CCS Concepts

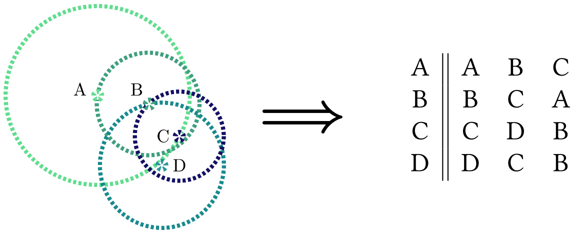
• **Computing methodologies** → **Massively parallel algorithms**; *Graphics processors*; • **Information systems** → *Top-k retrieval in databases*; Spatial-temporal systems.

Keywords

nearest neighbours, GPGPU, ray tracing cores, voronoi

ACM Reference Format:
Victor Kamel, Hanxueyu Yan, and Sean Chester. 2025. CLOVER: A GPU-native, Spatio-graph-based Approach to Exact kNN. In *2025 International Conference on Supercomputing (ICS '25), June 8–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3721145.3730415

ACM ISBN 979-8-4007-1537-2/2025/06
https://doi.org/10.1145/3721145.3730415



A	A	B	C
B	B	C	A
C	C	D	B
D	D	C	B

Figure 1: Illustration of an all-points 3-NN problem. The input (left) is an $n \times d$ matrix of n points in $\mathbb{R}^d$. The output (right) is an $n \times k$ matrix in which row i contains the ids of the k points closest to point p_i. Here, the 3-NN for each of four 2d points is visualised by a circle whose radius is the distance to the 3rd closest neighbour.

1 Introduction

Finding the k nearest neighbours to a point (kNN) for many points simultaneously is a ubiquitous problem, known as a *kNN join* in databases and data mining [5, 26], *all-points kNN* in computational geometry [34], and *kNN graph construction* in machine learning [19, 22]. Figure 1 illustrates an example.

A kNN join is a bottleneck kernel in many GPU applications for low-dimensional spatial data, such as k-means clustering, iterative closest point methods, remeshing with centroidal voronoi tesselation (CVT) [32], and detecting interactions among agents in animations and simulations [41]. In iterative problems such as these, points are *dynamic* and *GPU-resident*; so, one cannot rely on a data structure pre-built on the CPU: **construction time matters**.

A natural and common approach would be to use high-throughput, massively parallel scans. State-of-the-art GPU kNN scan-based methods use tiling together with clever priority queues stored in registers that are updated with warp-level synchronisation primitives [20, 22, 31, 35, 36, 38]. Indeed, for small query batches (e.g., $n \leq 10^3$) that barely saturate the parallelism in a GPU device, these are blazingly fast, requiring tens of microseconds in our experiments (Section 4). However, as n grows, the poor $\Omega(n^2 d \lg k)$ work complexity catches up to scans, despite the high throughput. Figure 2 reports a preliminary experiment, showing on the

y-axis the proportion of 94 3d meshes from Thingi10k [46] for which our proposed method requires a given proportion of work performed by a scan-based method. It shows that we need just 1–2% of the work; i.e., scans need an impractical 50–100× better throughput to match performance.

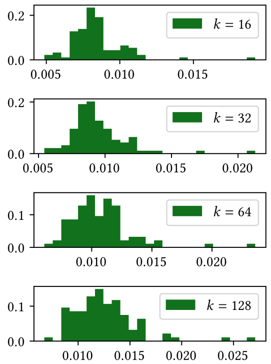


Figure 2: Proportion of points scanned on 94 3d meshes ($|\mathcal{P}| \geq 100\,000$) .

On the CPU and with preprocessing, kNN in 3d can be done efficiently using trees like the kd-tree in nanoflann [3]. *Ignoring the expensive tree construction step*, one can process 100K nearest neighbour queries in dozens of milliseconds [41]. However, when processing should be done on the GPU, these tree-based techniques perform quite poorly, despite reasonable effort to adapt them [2, 12, 29, 45]. The irregular data layout and branch-heavy traversal strategies impair throughput [15]. We show that, though they can improve on scans, they are still uncompetitive.

More recently, kd-trees have been superceded by RT-core-accelerated bounding volume hierarchies (BVHs) [11, 25, 28, 47]. However, this introduces overhead by performing problem reductions prior to BVH construction and we find they degrade as query volume, and therefore the BVH, grows.

As it turns out, the simple grid method used for the kNN step of CVT [32] is much faster than these methods. However, it is sensitive to the data distribution. We are left in need of a new approach to dramatically reduce work that can adapt to skewed data without an expensive preprocessing step.

This work proposes CLOVER (Section 3): a lightweight graph data structure that can be rapidly built on the GPU. CLOVER partitions data into random voronoi cells. It then constructs a graph over voronoi cells whose edge weights bound the distances between any two points in those cells. This gives a partitioning with similar properties to a grid method, but that dynamically adapts to the underlying data.

The approach resembles data structures used in computer vision [23], machine learning [7], information retrieval [4], and robotics [14] for matching high-dimensional feature vectors (ANN). However, it is not heuristic like hierarchical graph-based [16, 30, 44] and quantization-based methods [17, 20], and is optimized for 3d data. We evaluate CLOVER against state-of-the-art scan-based [20], tree-based [15], grid-based [32], voronoi-based [6], and RT-core [25] methods.

In this paper, after reviewing background and related work (Section 2) and before concluding (Section 5), we:

- Propose optimisations to truncated bitonic sort to utilise warp-level synchronisation primitives (Figure 12)
- Introduce the CLOVER data structure and corresponding kNN query algorithm for efficient and high-throughput GPU performance (Section 3)
- Comprehensively evaluate CLOVER against scan-based, grid-based, tree-based, voronoi-based, and RT-core alternatives on a range of synthetic and real datasets (Section 4), which additionally is the first comparison between CUDA core grid-based and RT-core methods.

2 Background & Discussion

Let $\mathcal{P}$ be a set of $n = |\mathcal{P}|$ points in $\mathbb{R}^d$ and $d(p_i, p_j)$ be a metric distance function between points $p_i, p_j \in \mathcal{P}$, assumed to be L_2. A kNN query transforms a query point q into a set of the k closest points in $\mathcal{P}$ to q; i.e.,

$$k\text{NN}(q, \mathcal{P}, k) = \underset{S \subseteq \mathcal{P}, |S|=k}{\arg\min}\ \Sigma_{p \in S} d(q, p).$$

One usually returns both the distances to and indices of the kNN, ordered by ascending distance. We study massively parallel $kNN(p_i, \mathcal{P}, k)$ queries, e.g., each $p_i \in \mathcal{P}$ is a query.

2.1 Scan-based approaches

Scan-based techniques emphasise throughput on the GPU and require very little additional memory. They obtain excellent utilisation, but complete $\Theta(n^2 d \lg k)$ work.

Early scan-based techniques are characterized by 2 stages: computing distances to the query points and then sorting those distances [42]. More effective is to maintain and incrementally update a sorted top-k priority queue (PQ), as in `pytorch3d` [31]. Parallel merge sorts can exploit that the PQ is sorted and have been shown to quickly process a single kNN query on 2^{29} data points [35]. Repeated sorting is expensive, even for low k. Buffering updates with queues outperforms truncated sorts [38]. Modern scan-based kNN algorithms first compute an $n \times q$ distance matrix using an optimized GEMM kernel, then maintain a priority queue such as a heap along the rows [12, 40]. State-of-the-art scans, e.g., FAISS [20] and RAFT [43], will additionally apply tiling to improve throughput, which then facilitates tensor cores [19].

From this class, we compare to an optimisation of FAISS.

2.2 Grid-based approaches

Grid-based methods reduce work by hashing points to grid cells and then only comparing points that are in nearby cells. Because the partitioning is simple and data-parallel, throughput can be very high; however, performance is sensitive to the data distribution. Figure 3 illustrates the general idea of

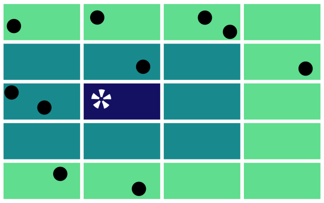


Figure 3: Searching concentric rings with a static grid.

a grid-based method, such as [32].[1] One normalises the data to fill a statically-defined grid, reducing skew and providing O(1) lookup of a point's grid cell based on its coordinates. Cell widths are selected to fit 3.1 points per cell, on average. Cells are ordered and points are physically sorted to enable coalesced reads. Given a query point, such as the white ✻ in the figure, one reads points in the same cell as the query point, then in the neighbouring cells, then in the cells neighbouring those, and so on until the minimum distance to a cell "ring" is greater than to the kNN so far.

Throughput could be imparied if threads diverge on the number of rings. The number of cells per ring grows very fast, following the series $[1, 26, 98, 218, 386, \ldots]$[2] in 3d and $(2n+1)^d - (2n-1)^d$ in general. Also, shared memory usage is high and can impair occupancy. As a result, grid methods are more common for GPU range queries (c.f., `cuNsearch`[3] and NVIDIA's spatial processing `warp` library[4]) than for exact kNN, as then the search range is explicitly bounded and all query points require examining the same number of rings.

From this class, we compare to [32], which supports $d \leq 3$.

2.3 Hierarchical spatial partitioning

Trees, such as the `nanoflann` kd-tree [3], are successful on CPUs due to good cache hit ratios and an effective reduction in work. However, they depend on expensive preprocessing and lack both data-level parallelism and read coalescing.

Nonetheless, kd-trees [15], oct-trees [33] and R-trees have been explored on the GPU [42]. A variant of the R-tree, the SS-tree, can be constructed in a parallel bottom-up manner, and traversed in parallel [29]. A list-of-clusters method and an SSS-Index have also been tried on the GPU [2]. Recently, [48] introduced a new GPU-based tree for high-d metric spaces. Like scan-based methods, it is based on pre-materialisation of a distance matrix. Tree construction is consistently demonstrated to be too slow for online processing [9].

From this class, we compare to a well-maintained and mature GPU kd-tree library, Treeology [15].

2.4 Flat voronoi partitioning approaches

Flat voronoi approaches induce a voronoi tesselation on the data space and then use spatial pruning techniques based on the triangle inequality. TOP [10] provides a framework for applying this idea to a range of spatial problems, including kNN. The Sweet KNN [6] method applies this directly to the GPU, attempting to balance several "sweet spots" between throughput and efficiency with dynamic configurations of parameters like shared memory.

CLOVER overlaps this class in that it also uses voronoi partitioning. We compare against Sweet KNN [6].

2.5 Ray-tracing (RT) core approaches

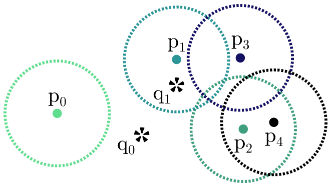


Figure 4: Using RT cores for *truncated* kNN. A BVH is constructed over the two query points $\{q_0, q_1\}$ and a fixed-radius range query is executed for every data point p_0–p_4 instead.

A special class of tree-based methods use hardware acceleration to traverse a bounding volume hierarchy (BVH). This can suggest higher throughput, but involves several caveats. First, RT cores are not available on data centre GPUs (e.g., V100, A100, H100). Moreover, on an RTX card, they may be better allocated to expensive rendering tasks than kNN. These methods do not support beyond 3d data and confine one to a BVH, which is not the preferred software-only tree for kNN. Finally, one incurs the cost of a reduction to cast kNN into a range query and then a ray tracing problem and back again.

These methods are all built off of two core ideas from Evangelou et al. [11]. First, one can perform a *truncated* kNN as a range search instead, in which one introduces a hyperparameter, $\tilde{r}$, and only returns results within a distance of $\tilde{r}$. "[T]his truncation comes out of necessity rather than choice" [11], requires a sufficiently large choice of $\tilde{r}$, and necessitates a filter step to remove false positives.

Second, if one assumes that n is orders of magnitude larger than the number of queries, then one can index the queries in the BVH and execute range queries with the data points instead. This leads to more, fine-grained parallel tasks traversing shallower BVHs, and therefore faster parallel time. However, it now requires a fixed threshold, $\tilde{r}$, to be used on all queries, whereas the distance to the kth neighbour could vary quite a lot across query points. Figure 4 illustrates this.

Subsequent work has tried to address the heuristic nature of this approach and the difficulty of setting $\tilde{r}$. RTNN [47]

[1] This description is based on source code inspection, not the paper.
[2] https://oeis.org/A010014
[3] https://github.com/InteractiveComputerGraphics/cuNSearch
[4] https://github.com/NVIDIA/warp/blob/main/warp/native/hashgrid.h

spatially groups queries to nearby rays to reduce branch divergence. It also partitions queries to create multiple, smaller BVHs, improving parallel ray-tracing time at the expense of BVH construction overhead. TrueKNN [28] produces correct kNN results by first sampling the data to estimate a good $\tilde{r}$, then repeatedly executing RT range queries with increasing diameters until it contains k points. Finally, Arkade [25] proposes reductions for non-Euclidean distances and demonstrates a 1.3×–33.1× speedup over FastRNN for L_1 distance.

From this class, we compare to the latest method, Arkade [25], which reports much faster times than TrueKNN [28].

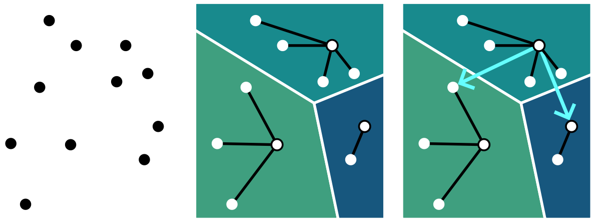


Figure 5: The lower bound concept. $H = 3$ random points are selected as "hubs" and all points are assigned to the nearest hub (i.e., a random voronoi partitioning). Then, as shown for the top cell, the asymmetric $d(h_i, h_j)$ is set to the distance from h_i to the closest point in h_j.

3 CLOVER: Spatio-Graph-Based kNN

Both data structure construction and exact 3d kNN queries need to be massively parallel and efficient. We combine spatial- and graph-based ideas with a random voronoi partitioning and geometric identities to prune the search space. We then in parallel determine which hub every point is closest to, denoted with black lines, as in a graph-based method [24, 27]. This produces random voronoi partitions.

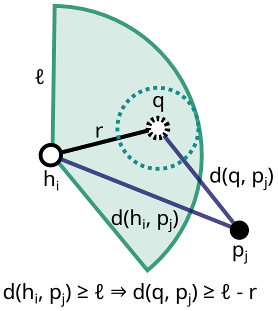


Figure 6: Application of the triangle inequality in CLOVER.

Consider the set of points in the leftmost subfigure of Figure 5. We select a random subset of reference points that we call "hubs," denoted as the hollow points in the middle subfigure. Although we draw the boundaries between the voronoi cells with white lines in the figure, they are only conceptual: we do not ever compute these cell boundaries.

The first concept in our technique is to create an asymmetric distance bound between all of the hubs, similar to GNAT [4]. Let P_i be the set of points closest to hub h_i and likewise and P_j for h_j. The (minimum) distance from h_i to h_j is defined as $d(h_i, h_j) = \min_{p_j \in P_j} d(h_i, p_j)$, i.e., the distance from hub h_i to the closest of all points in h_j's cell. This is illustrated by arrows for one hub in the right subfigure.

This enables a lower bound on the actual distance between any two arbitrary points, q and p_j, as illustrated in Figure 6. Let h_i be the hub to which q is closest and h_j, to which p_j is closest. Then, from Euclid's triangle inequality, we know that $d(q, p_j) \geq d(h_i, h_j) - d(h_i, q)$, since $d(h_i, p_j) \geq d(h_i, h_j)$. The figure illustrates that this distance is minimised when the triangle is degenerate.

In fact, if this bound holds for some point, p_j, then it holds for all points p'_j in the same voronoi cell. We use these lower bounds to prune a substantial number of points when we know that their distance to a specific hub exceeds the distance to the kth closest already-seen point [34].

Those ideas have been used before. The second concept, however, is that we transform the partitions into a graph: the voronoi cells become the nodes, the asymmetric distance becomes the edge weights, and the payload at each cell becomes the set of points within it. We sort each adjacency list by ascending edge weight, illustrated in Figure 7.

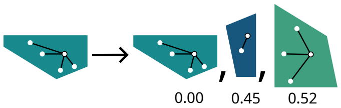


Figure 7: The novel graph concept. An adjacency list to other voronoi cells is ordered by edge weight.

The graph is stored in compressed sparse row format [21], along with a mapping from points to cells and the distance to their hubs. We call the data structure *Contiguous Lower-bound-Ordered Voronoi cells for Exact Retrieval* (CLOVER 🍀), as our key insight is to reorder voronoi cells by ascending lower bound, $d(h_i, h_j)$, to terminate queries early.

kNN queries are answered by traversing CLOVER. Query point $q = p_i$ is mapped to a voronoi cell; then, cells are iterated in best-first, sequential order. The traversal ends once we visit k points that are closer than the distance bound implies for the next—and remaining—voronoi cell(s).

Figure 8 conceptually illustrates four concurrent queries with one CLOVER. Points in the same cell have the same colour, the query point is denoted by ✲, and the query point's hub is denoted by ∘. Each query traversal begins at ∘ and includes all points that have the same hub as ✲. The traversal then expands concentrically to the voronoi cell that has the next closest point to ∘. On each step, all points of one colour, i.e., in one voronoi cell, are scanned collaboratively by a warp, coalescing reads. Then the bound is rechecked to confirm if the next concentric ring is closer to ✲ than to any k visited points. The number of cells scanned depends on k. For the

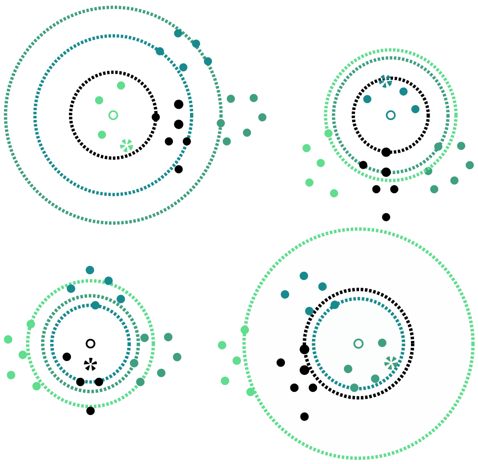


Figure 8: Traversing the same rapidly-built CLOVER data structure from the view of four concurrent k-nearest neighbour queries on separate GPU warps.

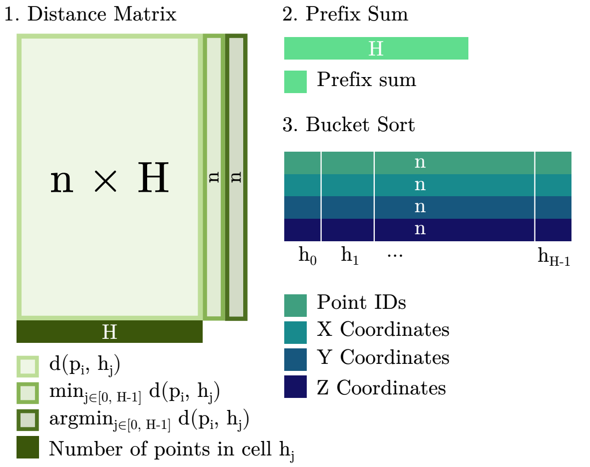


Figure 9: Sequence of kernels to reorder points contiguously by voronoi cell. Matrices & vectors with a border are not used as input in the following kernel.

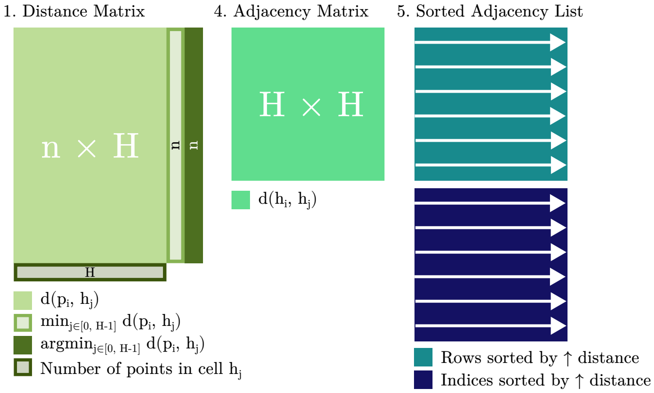

1. Distance Matrix
n × H
n
n
H
d(p_i, h_j)
min_{j∈[0, H-1]} d(p_i, h_j)
argmin_{j∈[0, H-1]} d(p_i, h_j)
Number of points in cell h_j
4. Adjacency Matrix
H × H
d(h_i, h_j)
5. Sorted Adjacency List
Rows sorted by ↑ distance
Indices sorted by ↑ distance

Figure 10: Sequence of kernels to order adjacency lists of voronoi cells by lower bound. Vectors with a border are not reused in the following kernel.

query in the bottom left, only two cells/colours must be scanned for $k \in [2, 5]$ as the fifth closest point to ✲ (including both ✲ and ◦) is nearer to ✲ than the second ring is.

3.1 Data-parallel construction of CLOVER

Six kernels construct CLOVER: one to quickly select random hubs, then three to build the bimapping from points to hubs and two others to build the sorted adjacency lists.

Figure 9 illustrates the first stream of kernels. After H hubs are randomly selected, each thread calculates and records the distance of one point, p_i, to all hubs, h_j, keeping track of both which and how close is the closest hub, h_j. When the thread is finished, it atomically increments the jth counter of cell counts. After this kernel, we have a mapping from points to hubs with distances, we have materialised the $n \times H$ distance matrix for reuse, and we know the number of points in each voronoi cell. Because d is small, we physically reorganise points by hub in a struct-of-arrays format to coalesce reads during queries. The second kernel performs a prefix sum over the cell sizes to transform it into an array of offsets. The third kernel uses a copy of the prefix sum array to physically transform points and counting sort them by voronoi cell. In total, these kernels require $\Theta(dnH)$ work and $\Theta(dH)$ span.

Figure 10 illustrates the second stream of kernels. Note that the first kernel was already executed in Figure 9. In the fourth kernel, we launch a thread block per row h_i of the $H \times H$ edge weight matrix in order to fit the row in shared memory. Threads cooperatively read a column of the $n \times H$ distance matrix and use atomic min operations to identify the lower bound distance to each other hub. Finally, the fifth kernel physically sorts each row by distance bound using one warp, registers and bitonic sort. Together, these kernels do $\Theta(dnH + H^2 \lg^2 H)$ work with $\Theta(dH + \lg^2 H)$ span.

Once constructed, CLOVER consists of the following:

- the row min and argmin vectors from the first kernel for $\Theta(1)$ look up of a query point's hub and distance;
- the prefix sum and reordered data from the second and third kernels to determine the contiguous list of points assigned to a given hub in $\Theta(1)$ time;
- the $H \times H$ edge weight matrix from the fourth kernel;
- the $H \times H$ adjacency lists produced in the fifth kernel.

Thus, the memory footprint for CLOVER construction is $\Theta(H(n+H))$ and the data structure requires $\Theta(H^2+nd)$ space. As H should be a small multiple of the thread block size and fit in shared memory, it is effectively a small constant.

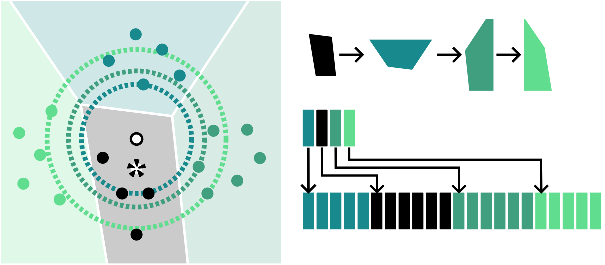


Figure 11: Traversing voronoi partitions by lower bound to q's hub, h_q.

Algorithm 1 Process a kNN query with CLOVER.

```
Function CloverQuery(CloverDataStructure, query id, k)
1:  Q ← k × (∞, ∅) (dist, id) pairs for initial state
2:  q ← point_ids[query id] to group queries in same block
3:  [$h_q$, $d_q$] ← hub id and distance $d(q, h_q)$ for query q
4:  sorted_cells ← (ref to) sorted adjacency list for $h_q$
5:  for each hub $h_i$ in sorted_cells do
6:      if queue[k-1].d < $d(h_q, h_i) - d(q, h_q)$ then return Q
7:      for each warp step in [offsets[$h_i$], offsets[$h_i + 1$]) do
8:          lane i calculates distance to point i
9:          if __any_sync(dist i < queue[k-1].d) then
10:             insert (new distance, new point id) into queue
```

3.2 Warp-collaborative kNN with CLOVER

Figure 11 illustrates and Algorithm 1 describes a kNN query executed by a warp using CLOVER. First, query point ✲ looks up its hub ○ and the distance $d($✲$, \circ)$ thereto using the materialised map. In the figure, it is in the black voronoi cell; so, the traversal will access cells in the order of the black cell's sorted adjacency list. We check if the ring is farther from ✲ than the kth element in our priority queue. If not, we retrieve the start and end indices of points in that cell ($i \in [5, 11)$ for that first black cell and $i \in [0, 5)$ for the next one) and then the whole warp scans (strip mines) the cell.

To scan a cell, each thread steps with a stride of 32. It grabs its next index, x-, y-, and z-coordinate (if there are enough left) from the struct-of-arrays layout, coalescing reads, and calculates the distance from this next point to ✲. All threads then compare their newly computed distance to the kth best score in the priority queue, which they have each locally stored in a register. If any thread in the warp votes to update the priority queue, all threads invoke a push heap function with their point to update the top-k correctly.

To track the top-k requires a priority queue (Q) data structure and careful consideration of state and register pressure. One level of indirection (Line 2) reorders queries per the sort order of points so that warps in the same block usually process queries from the same voronoi cell. We want to avoid use of shared memory to maximise the amount of L1 available for temporal locality within the block. Moreover, updates to Q and reads are really "hot" code and k is small. So, we prefer to maintain Q's state in warp registers. Each thread locally keeps $\lceil k/32 \rceil$ keys (indices) and values (distances), which will be synchronised with ballots and shuffles.

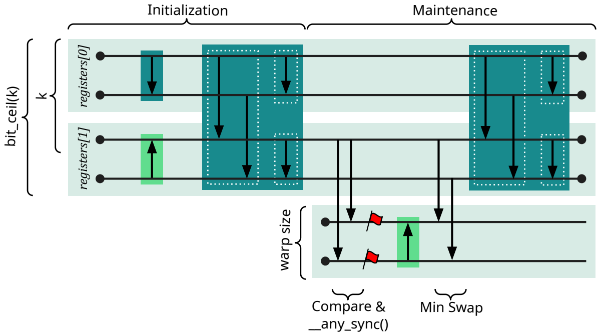


Figure 12: Truncated bitonic sort as a priority queue in registers. Illustrated with a warp size of two threads. Insertions only require the last register for each thread.

We use the per-thread buffering ideas [38] of FAISS [20]. We expect this to cost extra registers both for the buffers themselves and the flushing logic to manage them, leading to spills. Thus, we also design a register-native, warp-level version of the truncated bitonic sort of [35, 36] (Figure 12).

To prove that the query will not miss one of the k nearest neighbours, p_j, assume for the sake of contradiction that actually this has happened. Then either p_j was not scanned, or it was not inserted into the priority queue because of the ballot on Line 9. The latter is not possible, because otherwise the thread processing p_j would have voted to enter the loop. The former can only occur if p_j is in a voronoi cell h_j for which $d(h_q, h_j) - d(q, h_q)$ is greater than k already-scanned points. However, $d(h_q, h_j) - d(q, h_q)$ should be a lower bound on $d(q, p_j)$ and $d(q, p_j)$ is among the k smallest distances from q. Thus, we arrive at a contradiction.

4 Experiments

In this section, we evaluate the performance of our proposed CLOVER for exact kNN on the GPU.

4.1 Experiment design

4.1.1 Software. We compare eight methods implemented in CUDA/C++.[5] Unless otherwise noted, each query is processed with one warp and four warps per block.

[5] Source code is publicly available at https://github.com/ampslab/clover-knn.

Table 1: Properties of datasets used in experiments.

Name [source]	abbrv	n	d
Stanford Bunny [39]	BUN	362 272	3
KITTI-1M [13]	KIT	1 000 000	3
Household [18]	HH	2 049 200	4
Happy Buddha [8]	BUD	4 586 124	3
Galaxy-10M [37]	GXY	9 925 229	3
Uniform Synthetic	UNI	$\log(n) \in \{4, 5, 6\}$	3
Gaussian Mixed Model	GMM	$\log(n) \in \{4, 5, 6\}$	3

i. **FAISS**: The off-the-shelf `bfKnn()` scan-based method from FAISS [20] with modest tuning to the device, such as constraining registers to improve occupancy. This is a tiled approach that does not use one warp per query.
ii. **WarpSelectScan**: A scan-based baseline that invokes WarpSelect from FAISS [20], targeting low dimensions. It uses neither tiling nor materialised distance matrices.
iii. **Treeology**: A well-maintained GPU *kd*-tree method used as a baseline in [25] with one query per thread.
iv. **SweetKNN**: The straight-forward application of voronoi-based techniques to the GPU [6]. It processes one query per thread and stores priority queues in shared memory.
v. **Arkade**: The state-of-the-art, ray-parallel, BVH-based method on ray-tracing cores [25] (subsection 4.3 only).
vi. **rsll18**: The grid-based method used in CVT [32] (Figure 3). It processes one query per thread and stores priority queues in shared memory.
vii. **BitonicClover**: Our CLOVER proposal using truncated bitonic sort as described by [35] but in registers with our `__any_sync()` optimisation (Figure 12).
viii. **WarpSelectClover**: Our CLOVER proposal linking the same priority queue as WarpSelectScan.

4.1.2 Datasets. We use three types of datasets, the properties of which are shown in Table 1. The first type are real datasets used in recent ray-tracing kNN papers [28, 47] and spatial processing for comparability. The Stanford Bunny is a raw 3d scanned point cloud ubiquitous in computer graphics. The KITTI-1M [13] dataset is a collection of LiDAR points generated from self-driving cars. Household [18] is a time series dataset of electric power consumption for individual households that we project onto the active power, reactive power, voltage, and intensity attributes. The Happy Buddha [8] is a larger point cloud from the Stanford 3d scanning repository. Galaxy [37] is our largest dataset, stressing scale.

The second set of experiments evaluate scalability in a controlled fashion. We generate uniform synthetic points using a `std::mt19937` random engine and fixed seed (UNI). We likewise generate data per a gaussian mixed model (GMM) distribution. The GMM data is uniformly distributed between $(-1000, 1000)$ over the x- and y-coordinates; for the z-coordinate, 1000 peaks[6] are selected at random from the range $(-1000, 1000)$ and coordinate values are distributed around those peaks with a standard deviation of 100. This simulates a point cloud for a smooth but hilly 3d surface and enables us to test sensitivity to non-uniformity in the data.

4.1.3 Environment. We test in two environments: an NVIDIA Tesla V100-SXM2-32GB Volta (GV100) data centre GPU (subsection 4.2) and an NVIDIA RTX 4070 Laptop Ada Lovelace (AD106) RT-core-equipped GPU (subsection 4.3).

In both environments, timings are started with `std::chrono` just after calling `cudaDeviceSynchronize()` when input data has been transferred to pre-allocated device arrays. They end just after the final kernel invocation and another call to `cudaDeviceSynchronize()`, but before transferring data back to host. This reflects the scenario where kNN is part of a GPU pipeline, rather than using the GPU as an accelerator to CPU-centric processing. By default, the number of neighbours is $k = 30$ and CLOVER uses $H = 1024$ cells.

4.2 Results & analysis on data centre cards

4.2.1 Speedups on all datasets. Figure 13 reports the relative performance of all methods for 30-NN on all datasets, relative to our proposed WarpSelectClover method. Timings include data structure creation, if applicable, and the total time to complete all n queries. Execution times for WarpSelectClover are provided in the caption and slowdown values are provided as data labels; so, the execution time of a particular method on a particular dataset can be obtained by multiplying the WarpSelectClover time by the given slowdown. Datasets are ordered first by d and then by n. Note that Treeology and rsll18 do not support 4d datasets.,

Observe first that the y-axis is split so that we can visualise both small and extreme differences. Although WarpSelectScan provides a marked improvement over FAISS, both are uncompetitive, with FAISS regularly being over 200× slower on large datasets. This is unsurprising, given the poor work efficiency. Without data structures to reduce the amount of work, the quadratic complexity is overwhelming. On the 10-million point GXY dataset, FAISS exceeds our ten-minute time limit and WarpSelectScan barely finishes.

Treeology is also uncompetitive, ranging from 10–57× slower than WarpSelectClover. While the tree provides greater efficiency than the scan-based methods, concurrent traversals by multiple threads lead to both flow and memory divergence on the GPU; high throughput is thus elusive.

SweetKNN performs better than Treeology, but runs out of memory on datasets with $n > 1$M.

[6] We exclude an experiment that varied the number of peaks, i.e., the amount of non-uniformity, from 500 to 5000, due to minimal effect and observations.

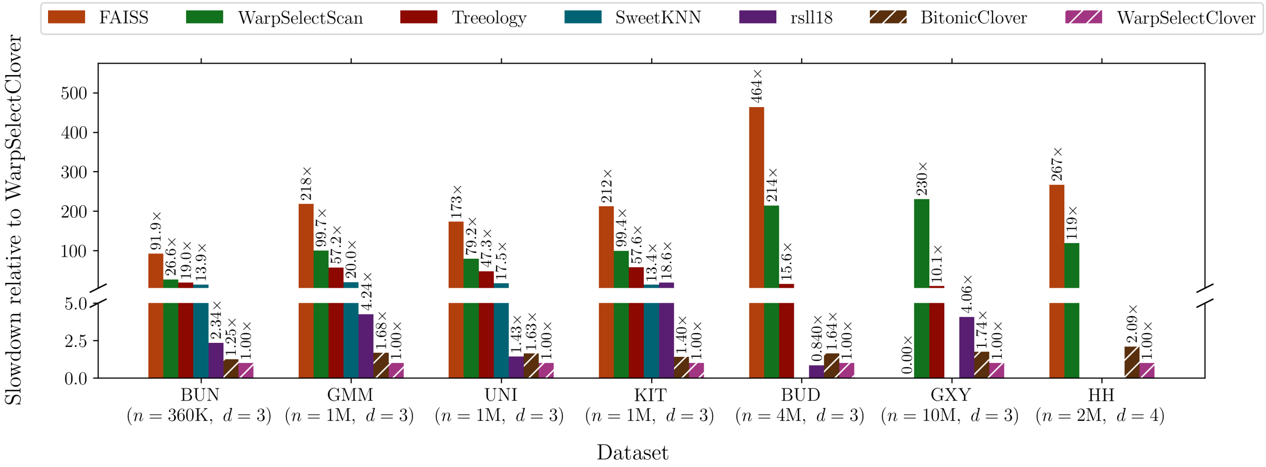


Figure 13: Slowdowns relative to WarpSelectClover ($k = 30$, NVIDIA V100), i.e., time(algorithm, dataset) / time(WarpSelectClover, dataset). 0.0× **indicates methods that did not complete. WarpSelectClover timings: 21.4ms (BUN); 56.9ms (GMM); 80.2ms (UNI); 63.9ms (KIT); 617ms (BUD); 2.71s (GXY); 213ms (HH).**

The most competitive methods are those based on lightweight data structures, including the static grid of rsll18 and the CLOVER methods. WarpSelectClover outperforms rsll18 on all datasets except Buddha, including a surprisingly high 18.6× outperformance on the KITTI dataset. The use of the WarpSelect priority queue from FAISS provides a 25–109% improvement over truncated bitonic sort. The extra register pressure that we expected does not materialise.

We observe the same general trends on the 4d HH dataset, except that Treeology and rsll18 cannot be run.

Table 2 evaluates the impact of data skew, reporting for each algorithm the performance on 1M 3d GMM points divided by the performance on 1M 3d UNI points. Predictably, the scan-based methods, FAISS and WarpSelectScan, are unaffected as they must scan all points for all queries. The grid-based rsll18 method degrades appreciably, requiring more than twice as long, as the number of rings to traverse varies more between threads. Conversely, Treeology observes a minor 4% improvement, likely due to better partitionability of the data. The CLOVER methods run 25% faster on skewed data because the $H = 1024$ randomly selected centroids adapt (in expectation) to the underlying data distribution.

Table 2: Effect of data skew. Performance for each method on 10^6 GMM points relative to 10^6 UNI points.

FAISS	WS.Scan	Treeology	rsll18	WS.Clover
1.01×	1.00×	0.96×	2.36×	0.79×

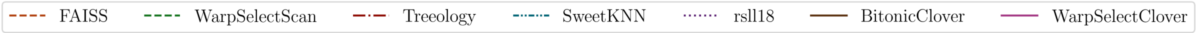
4.2.2 Scalability analysis. Figure 14 reports the total time of each method for 30-NN as the number of UNI data points (and thus also query points), n, increases from 10K to 1M. Both axes are log-scale. The rsll18 implementation throws runtime exceptions on cases where $\sqrt[3]{n/3.1} < 16$ as this throws off the fine-tuned, hard-coded hyperparameters. So, its curve starts at $n > 12697$.

The scan-based methods perform best on small meshes, but exhibit a quadratic performance curve. By $n = 1$M, neither scan-based method completes in less than a second. Treeology and rsll18 degrade at an apparently linear rate, demonstrating better work-efficiency; however, Treeology still requires over a second from $n \geq 1$M. CLOVER has an overhead for data structure creation of 1–2ms, but then scales gracefully. By $n = 1$M, WarpSelectClover is 43% faster than the grid-based approach and ≥ 47× faster than the others.

Comparing the two CLOVER instantiations, WarpSelectClover consistently outperforms BitonicClover. High register pressure leads to spills (per verbose ptxas output) in WarpSelectClover when we set launch bounds to maximise occupancy. Yet, the register spills do not impair performance.

Figure 15 repeats the scalability experiment with 3d GMM data. Other than the previously reported lifts and penalties due to data skew, we do not observe any significant differences in trend from the uniformly distributed data. Although it is not shown, the same is true of 4d UNI and GMM data, though in those cases we cannot run Treeology or rsll18.

Figure 16 instead holds $n = 1$M fixed and varies the output size from $k = 2^5$ to $k = 2^7$. Broadly, we observe a performance degradation with respect to increasing k on the faster methods (rsll18, BitonicClover, and WarpSelectClover); however, for the slower methods, the cost of in-register (FAISS)



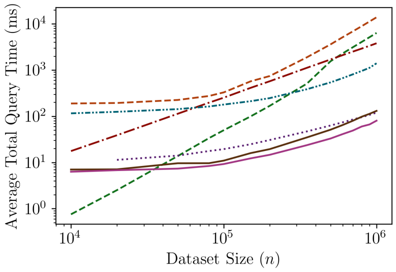


Figure 14: Total time relative to number of 3d UNI points ($k = 30$).

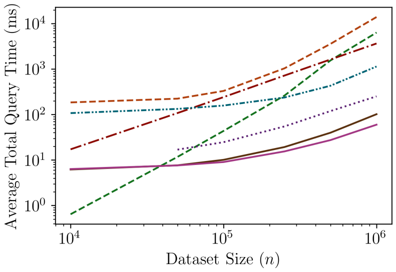


Figure 15: Total time relative to number of 3d GMM points ($k = 30$).

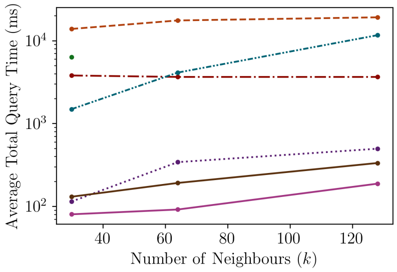


Figure 16: Total time relative to num. neighbours (3d UNI, $n = 10^6$).

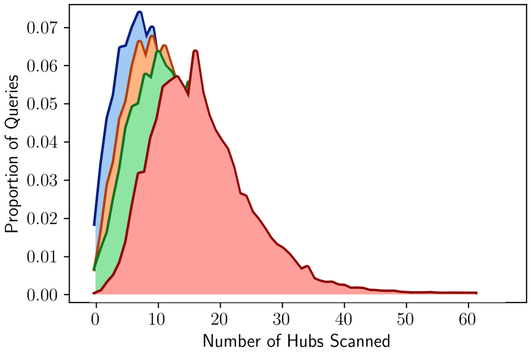


Figure 17: Histogram of hubs scanned per query.

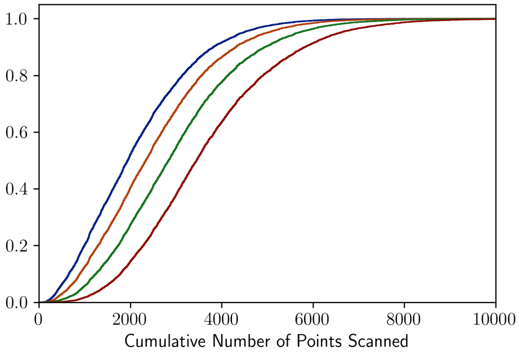


Figure 18: CDF of points scanned per query.

or shared memory (Treeology) top-k maintenance is a negligible factor compared to the global memory read operations that dominate cost. WarpSelectScan did not run for increased k due to excess resources requested in the launch configuration, but was already non-competitive at $k = 10$.

SweetKNN outperformed Treeology for large n, but only when k was small, likely due to its use of shared memory.

4.2.3 Work analysis. The performance of CLOVER comes, in part, from reducing work. Queries end as soon as finding a kth best distance smaller than the lower bound for all remaining hubs. Figure 17 shows a distribution of the number of voronoi cells scanned, out of $H = 512$, per query for $n = 100$K and $k \in \{16, 32, 64, 128\}$ on UNI. Due to the random partitioning, voronoi cells have non-uniform point populations. So, Figure 18 also measures work, but instead counts the number of points scanned and illustrates the information as a cumulative distribution. Note that this experiment is independent of the choice of priority queue used for the top-k, i.e., is the same for BitonicClover and WarpSelectClover.

Reading horizontally in Figure 18 shows the number of points scanned by a given percentage of queries. At $k = 16$, 50% of queries scan less than 2000 (2% of) points; 75% of queries scan less than 3% of points; 99% of queries scan less than 4% of points; and all queries scan at most 10% of points. As k increases, it takes longer to find a kth best point that can prune all remaining hubs, but the trend is the same. Even at $k = 128$, 50% of queries still scan < 4% of points.

Regarding tail latency, the longest running query accesses $52/512 = 10.1\%$ of hubs and $\approx 10^4/10^5 = 10\%$ of points, representing a 90% reduction in work compared to scans. The spatial bound is more effective for query points closer to their own hubs. Future work might reduce this tail latency with a more systematic approach to selecting hubs, as done with clustering methods like kmeans++ [1].

In summary, CLOVER incurs a small overhead for construction that makes scan-based methods faster on small inputs, but that regularly prunes over 96% of points, leading to marked improvements on larger inputs. This suggests an adaptive approach that uses CLOVER for large n.

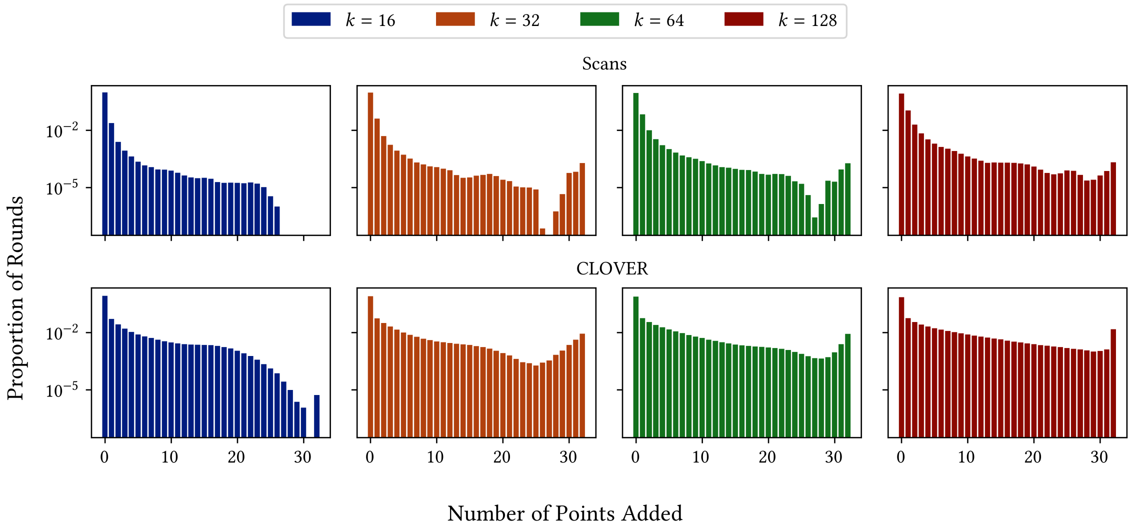


Figure 19: Distribution of how many threads encounter points that improve the top-k, per warp step.

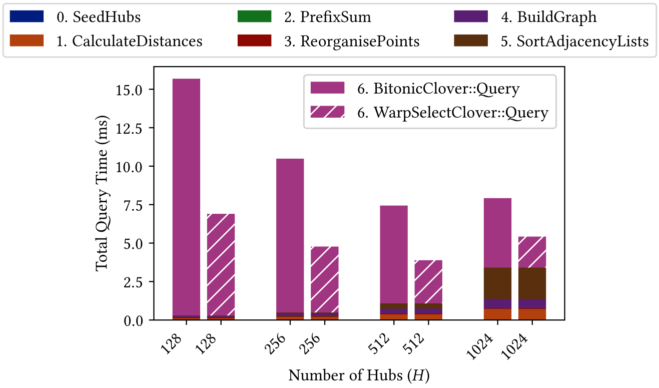


Figure 20: Execution time per kernel relative to H.

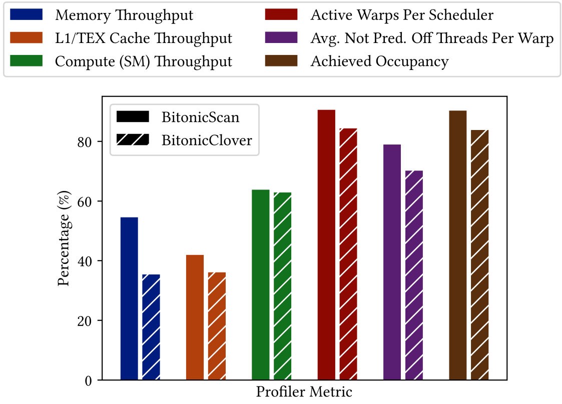


Figure 21: Throughput metrics from `ncu`.

4.2.4 Deeper analysis & profiling. We next investigate what drives the behaviour of the algorithms. Our objective with CLOVER was to balance construction and query time to minimise end-to-end processing time. Figure 20 evaluates the that goal by showing the relative time spent in each kernel for the two CLOVER methods. It also enables studying the effect of H, the number of voronoi cells.

Observe that CLOVER is orthogonal to the priority queue method that is chosen. The time for all construction kernels is the same for both CLOVER instantiations, but we see that WarpSelect halves query time. As H increases, execution time predictably shifts from queries to construction. At $H = 1024$, where the shared memory usage is high enough to suppress L1 cache hits, construction and queries are balanced already for BitonicClover. To understand why it is expected that increasing H shifts time from queries to construction, consider when $H = 1$ (or, similarly, a pathologically bad random partitioning). Then, CLOVER queries would degenerate to a scan-based method like WarpSelect. With a finer-grained partitioning of space, our spatial bounds are more effective.

Looking closer at the construction kernels, we observe that the dominant cost is the stream for building the voronoi adjacency lists. These are the two kernels for which our implementation relied on atomic operations. A better design with parallel reductions may be able to bring down these costs. Nonetheless, at $H = 512$, it only requires 1ms to construct the entire data structure. If the CLOVER data structure were constructed offline, we would continue to gain query performance at $H = 1024$, outperforming by an even larger

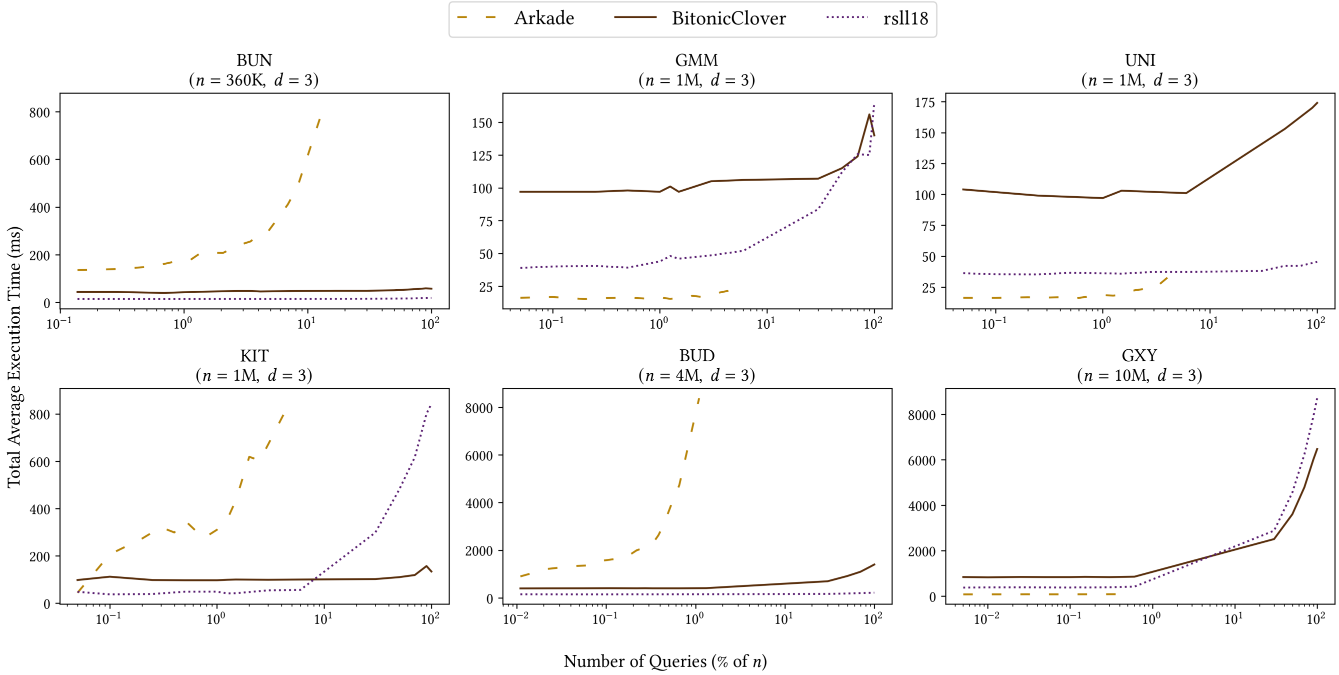


Figure 22: Total execution time relative to percent of points queried for 3d datasets (RTX 4070L, k = 10).

margin, though a larger range of baselines with offline index construction, such as [19], should be evaluated, then.

Figure 19 looks closer at the workload on the priority queues. We additionally run a method, BitonicScan, that uses the truncated bitonic sort priority queue within a simple scan-based method to isolate effects. BitonicScan, WarpSelectScan, BitonicClover, and WarpSelectClover all process 32 points at a time, using one thread to check the distance to one new point. A ballot instruction produces a mask of which threads were assigned points that were closer than the kth closest point already in the priority queue. The popcount of this mask indicates how many updates to the priority queue are required for this chunk. (This is exactly our `__any_sync()` optimisation on Line 9 of Algorithm 1: skip all update logic when the popcount is zero.) This experiment shows the histogram of those popcounts per warp step.

The first thing to note is that in BitonicScan, adding even one point in a step is rare. Over 97% of the time at k = 16, there is no update and 2.4% of the time, only one thread finds a new point. Even at k = 128, 85% of warp steps have no updates and 96% find at most one. Performance mostly comes down to read throughput to calculate distances.

Finally, Figure 21 reports relevant throughput metrics collected with NVIDIA Nsight Compute CLI (`ncu`). We compare BitonicScan and BitonicClover as representatives of scan- and CLOVER-based methods, because they have fully-fused query kernels that are amenable to profiling. Because CLOVER introduces memory indirection and predication, we focus on metrics related to those. Achieved occupancy as well as memory, L1 cache, and compute throughput are reported as a percentage of the device's peak theoretical throughput. Active warps are reported relative to device capacity (16 warps/SM). We report the percentage of 32 threads per warp that are active and not predicated off due to divergence.

In general, CLOVER accepts a small drop in throughput relative to a comparable scan on predication- and occupancy-related metrics. Compute throughput is unaffected. Memory, though not cache, throughput drops by 35%, but because the volume of memory transfers also decreases by > 95%.

In summary, most warp steps of kNN algorithms just confirm that the next 32 points are not better than any already seen. This supports optimising throughput over top-k maintenance logic. It supports even better CLOVER's design strategy to trade off a little throughput to spend 1–2ms building a data structure that avoids > 97% of warp steps altogether.

4.3 Results & analysis on an RTX card

Henceforth, we use an RTX 4070L (AD106) card to compare against an RT-core method, namely Arkade [25]. RT methods can only support 2d and 3d datasets. We repeat only the most competitive methods from subsection 4.2. For Arkade, we use the radius-tuning method described for TrueKNN [28], which is not included in the reported times.

Table 3: Performance degradation at $k = 30$ relative to at $k = 10$ per dataset/algorithm (RTX 4070L, $|Q| = 10^4$).

Algorithm	BUN	GMM	UNI	KIT	BUD	GXY
Arkade	1.27×	2.99×	2.17×	3.34×	1.03×	0.95×
rsll18	1.18×	1.15×	1.12×	0.96×	1.11×	1.16×
Bit.Clover	0.98×	1.01×	1.03×	1.00×	1.00×	1.00×

4.3.1 *Execution time on 3d datasets.*

Figure 22 reports the total execution time in milliseconds for BitonicClover, rsll18, and Arkade on the six 3d datasets, one per subplot. The x-axes are logarithmic. In this experiment, we fix $k = 10$, and vary the number of queries as a proportion of the total number of data points in the dataset.

The first point to observe is that Arkade generally cannot scale beyond 50k queries on this small but modern card. Recall that the RT-core methods generally build a BVH over the queries and then execute ray-parallel tree iterations using the data points. Thus, as the number of queries increases, so too does the size of the BVH and the span of the parallel algorithm. This method benefits most when the number of queries is many orders of magnitude smaller than n, in which case substantial fine-grained parallelism is exposed.

The second observation is the inconsistent performance of Arkade. It completes 50k queries on GMM and UNI in dozens of milliseconds, but takes over one second on BUD. This, we conjecture, is explained by the regularity of the data: if the distance to the kth nearest neighbour is similar for most queries, then methods like TrueKNN and Arkade require fewer rounds. Indeed, the best performance for Arkade is observed on regular, synthetic data. By contrast, the performance of BitonicClover and rsll18 is more consistent, because those methods answer the kNN query directly and therefore never require additional iterations to refine results.

We also observe that the relative performance between rsll18 and Clover is substantially affected by the GPU card. On BUN, for example, BitonicClover takes 58ms on the 4070L and 27ms on the V100; rsll18, by contrast, takes 19ms on the 4070L and 50ms on the V100. The 4070L has 15% fewer CUDA cores than the V100 and is optimised for thermal management and power efficiency, all of which handicap BitonicClover. On the other hand, it offers 5.3× more L2 cache, which benefits the memory-bound rsll18 method.

Lastly, we observe inflection points on larger datasets for rsll18 and BitonicClover, typically around 100K queries on skewed data (GMM, KIT, GXY). This is pronounced for rsll18 on KIT and often leads to BitonicClover outperforming rsll18 when executing all queries. We conjecture that this point is where the GPU device is saturated and the effect of workload balance shows up as a larger number of CTA barrier stalls.

4.3.2 *Effect of k.*

The previous experiment used a very low value of k to run more queries with Arkade. Table 3 reports the relative slowdown for executing 10^4 queries at $k = 30$ instead of $k = 10$. Because BitonicClover is warp-collaborative, it is unaffected by the increase in k; in fact, at $k = 10$, there were 22 threads that were idling during the maintenance of the top-k priority queue.

rsll18 observes a muted 11–18% degradation in performance, except on KIT where it performs slightly better. It uses a thread-parallel priority queue in shared memory, so increases in k increase the size of thoses heaps and therefore also the cost of any divergence on queue updates.

Arkade, in contrast, is affected unpredictably. It sees a slight improvement in GXY but a 3× degradation on GMM and KIT. Changing k affects the tuning of $\tilde{r}$, yielding a better radius on GXY. An increase to k also leads to more inconsistency across queries to find the ideal radius search that will contain the kNN. As a result, especially on skewed data like GMM and KIT, more refinement iterations are likely needed in order to obtain the correct result.

4.3.3 *RTX Summary.*

The RTX card makes ray tracing cores available for 3d datasets. Arkade, the latest RT method, can perform well when the number of queries and the number of neighbours is small, though results are inconsistent and depend on hyperparameter tuning. The grid-based rsll18 method benefits from extra L2 cache and our proposed voronoi-based Clover method still offers better performance on half of the datasets, despite the lower CUDA core throughput. These massively parallel methods solve kNN directly with efficient data structures, which gives them an inherent advantage over the indirect RT-core methods that perform a reduction from kNN to range queries to ray tracing.

5 Conclusion

We proposed CLOVER, a novel spatio-graph-based approach to answer exact kNN queries on a GPU by building a lightweight index on-the-fly. We demonstrated that CLOVER is 4× faster than an optimised grid-based method, 10× faster than a GPU tree-based method, and 230× faster than an optimised scan-based variant of FAISS on the largest of our datasets—including the time to construct the data structure. Moreover, we demonstrate scalability and reliability problems with the recently proposed line of research on RT-core-accelerated methods. CLOVER-enabled kNN search enables handling much larger scale for GPU-centric data processing.

Acknowledgments

This work was supported in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada through the Discovery Grants program and by the BC DRI Group, Compute Ontario, and Digital Research Alliance of Canada.

References

[1] David Arthur and Sergei Vassilvitskii. 2007. k-means++: the advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (New Orleans, Louisiana) *(SODA '07)*. Society for Industrial and Applied Mathematics, USA, 1027–1035.

[2] Ricardo J. Barrientos, José I. Gómez, Christian Tenllado, Manuel Prieto Matias, and Mauricio Marin. 2011. kNN Query Processing in Metric Spaces Using GPUs. In *Euro-Par 2011 Parallel Processing*. 380–392.

[3] Jose Luis Blanco and Pranjal Kumar Rai. 2014. nanoflann: a C++ header-only fork of FLANN, a library for Nearest Neighbor (NN) with KD-trees. https://github.com/jlblancoc/nanoflann.

[4] Sergey Brin. 1995. Near Neighbor Search in Large Metric Spaces. In *Proc. of the 21st VLDB Conference*. 574–584.

[5] Christian Böhm and Florian Krebs. 2004. The k-Nearest Neighbour Join: Turbo Charging the KDD Process. *Know. Inf. Sys.* 6 (2004), 728–749. https://doi.org/10.1007/s10115-003-0122-9

[6] Guoyang Chen, Yufei Ding, and Xipeng Shen. 2017. Sweet KNN: An Efficient KNN on GPU through Reconciliation between Redundancy Removal and Regularity. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 621–632. https://doi.org/10.1109/ICDE.2017.116

[7] Felix Chern, Blake Hechtman, Andy Davis, Ruiqi Guo, David Majnemer, and Sanjiv Kumar. 2022. TPU-KNN: K Nearest Neighbor Search at Peak FLOP/s. In *Advances in Neural Information Processing Systems*, Vol. 35. 15489–15501. https://proceedings.neurips.cc/paper_files/paper/2022/file/639d992f819c2b40387d4d5170b8ffd7-Paper-Conference.pdf

[8] Brian Curless and Mark Levoy. 1996. A volumetric method for building complex models from range images. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. 303–312.

[9] Tomáš Davidovič, Jaroslav Křivánek, Miloš Hašan, and Philipp Slusallek. 2014. Progressive Light Transport Simulation on the GPU: Survey and Improvements. *ACM Trans. Graph.* 33, 3, Article 29 (June 2014), 19 pages. https://doi.org/10.1145/2602144

[10] Yufei Ding, Xipeng Shen, Madanlal Musuvathi, and Todd Mytkowicz. 2015. TOP: a framework for enabling algorithmic optimizations for distance-related problems. *Proc. VLDB Endow.* 8, 10 (June 2015), 1046–1057. https://doi.org/10.14778/2794367.2794374

[11] Iordanis Evangelou, Georgios Papaioannou, Konstantinos Vardis, and Andreas A. Vasilakis. 2021. Fast Radius Search Exploiting Ray Tracing Frameworks. *Journal of Computer Graphics Techniques (JCGT)* 10, 1 (5 February 2021), 25–48. http://jcgt.org/published/0010/01/02/

[12] Vincent Garcia, Eric Debreuve, and Michel Barlaud. 2008. Fast k nearest neighbor search using GPU. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*. 1–6. https://doi.org/10.1109/CVPRW.2008.4563100

[13] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. 2013. Vision meets Robotics: The KITTI Dataset. *International Journal of Robotics Research (IJRR)* 32, 11 (2013), 1231–1237.

[14] Bryant Gipson, Mark Moll, and Lydia E. Kavraki. 2013. Resolution Independent Density Estimation for motion planning in high-dimensional spaces. In *2013 IEEE International Conference on Robotics and Automation*. 2437–2443. https://doi.org/10.1109/ICRA.2013.6630908

[15] Michael Goldfarb, Youngjoon Jo, and Milind Kulkarni. 2013. General Transformations for GPU Execution of Tree Traversals. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. 1–12. https://doi.org/10.1145/2503210.2503223

[16] Fabian Groh, Lukas Ruppert, Patrick Wieschollek, and Hendrik Lensch. 2023. GGNN: Graph-based GPU Nearest Neighbor Search. *IEEE transactions on big data* 9, 1 (2023), 1–1. https://doi.org/10.1109/TBDATA.2022.3161156 Place: Piscataway Publisher: IEEE.

[17] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating large-scale inference with anisotropic vector quantization. In *Proceedings of the 37th International Conference on Machine Learning (ICML'20)*. JMLR.org, Article 364, 10 pages.

[18] Georges Hebrail and Alice Berard. 2006. Individual Household Electric Power Consumption. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C58K54.

[19] Zhuoran Ji and Cho-Li Wang. 2022. Efficient exact K-nearest neighbor graph construction for billion-scale datasets using GPUs with tensor cores. In *Proceedings of the 36th ACM International Conference on Supercomputing (ICS '22)*. Article 10, 12 pages. https://doi.org/10.1145/3524059.3532368

[20] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2021), 535–547. https://doi.org/10.1109/TBDATA.2019.2921572

[21] Terence Kelly. 2020. Compressed Sparse Row Format for Representing Graphs. *;login:* 45, 4 (2020), 76–82.

[22] Ivan Komarov, Ali Dashti, and Roshan M. D'Souza. 2014. Fast k-NNG Construction with GPU-Based Quick Multi-Select. *PLOS ONE* 9, 5 (05 2014), 1–9. https://doi.org/10.1371/journal.pone.0092409

[23] David G Lowe. 2004. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision* 60 (2004), 91–110. https://doi.org/10.1023/B:VISI.0000029664.99615.94

[24] Yu A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 42, 4 (apr 2020), 824–836. https://doi.org/10.1109/TPAMI.2018.2889473

[25] Durga Keerthi Mandarapu, Vani Nagarajan, Artem Pelenitsyn, and Milind Kulkarni. 2024. Arkade: k-Nearest Neighbor Search With Non-Euclidean Distances using GPU Ray Tracing. In *ICS '24: Proceedings of the 38th ACM International Conference on Supercomputing*. 14–25. https://doi.org/10.1145/3650200.3656601

[26] Takazumi Matsumoto and Man Lung Yiu. 2015. Accelerating Exact Similarity Search on CPU-GPU Systems. In *2015 IEEE International Conference on Data Mining*. 320–329. https://doi.org/10.1109/ICDM.2015.125

[27] Marius Muja and David G. Lowe. 2009. Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration. In *International Conference on Computer Vision Theory and Application (VISSAPP'09)*. INSTICC Press, 331–340.

[28] Vani Nagarajan, Durga Mandarapu, and Milind Kulkarni. 2023. RT-kNNS Unbound: Using RT Cores to Accelerate Unrestricted Neighbor Search. In *Proceedings of the 37th ACM International Conference on Supercomputing* (Orlando, FL, USA) *(ICS '23)*. Association for Computing Machinery, New York, NY, USA, 289–300. https://doi.org/10.1145/3577193.3593738

[29] Moohyeon Nam, Jinwoong Kim, and Beomseok Nam. 2016. Parallel Tree Traversal for Nearest Neighbor Query on the GPU. In *2016 45th International Conference on Parallel Processing (ICPP)*. IEEE, Philadelphia, PA, USA, 113–122. https://doi.org/10.1109/ICPP.2016.20

[30] Hiroyuki Ootomo, Akira Naruse, Corey Nolet, Ray Wang, Tamas Feher, and Yong Wang. 2023. CAGRA: Highly Parallel Graph Construction and Approximate Nearest Neighbor Search for GPUs. arXiv:2308.15136 [cs.DS]

[31] Nikhila Ravi, Jeremy Reizenstein, David Novotny, Taylor Gordon, Wan-Yen Lo, Justin Johnson, and Georgia Gkioxari. 2020. Accelerating 3D Deep Learning with PyTorch3D. arXiv:2007.08501 [cs.CV] https://arxiv.org/abs/2007.08501

[32] Nicolas Ray, Dmitry Sokolov, Sylvain Lefebvre, and Bruno Lévy. 2018. Meshless voronoi on the GPU. *ACM Trans. Graph.* 37, 6, Article 265 (Dec. 2018), 12 pages. https://doi.org/10.1145/3272127.3275092

[33] Radu Bogdan Rusu and Steve Cousins. 2011. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, Shanghai, China.

[34] Michael Ian Shamos and Dan Hoey. 1975. Closest-point problems. In *16th Annual Symposium on Foundations of Computer Science*. 151–162. https://doi.org/10.1109/SFCS.1975.8

[35] Anil Shanbhag, Holger Pirk, and Samuel Madden. 2018. Efficient Top-K Query Processing on Massively Parallel Hardware. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. 1557–1570. https://doi.org/10.1145/3183713.3183735

[36] Nikos Sismanis, Nikos Pitsianis, and Xiaobai Sun. 2012. Parallel search of k-nearest neighbors with synchronous operations. In *2012 IEEE Conference on High Performance Extreme Computing*. 1–6. https://doi.org/10.1109/HPEC.2012.6408667

[37] Volker Springel, Simon D. M. White, Adrian Jenkins, Carlos S. Frenk, Naoki Yoshida, Liang Gao, Julio Navarro, Robert Thacker, Darren Croton, John Helly, John A. Peacock, Shaun Cole, Peter Thomas, Hugh Couchman, August Evrard, Jörg Colberg, and Frazer Pearce. 2005. Simulations of the formation, evolution and clustering of galaxies and quasars. *Nature* 435 (2005), 629–636. https://doi.org/10.1038/nature03597

[38] Xiaoxin Tang, Zhiyi Huang, David Eyers, Steven Mills, and Minyi Guo. 2015. Efficient Selection Algorithm for Fast k-NN Search on GPUs. In *2015 IEEE International Parallel and Distributed Processing Symposium*. 397–406. https://doi.org/10.1109/IPDPS.2015.115

[39] Greg Turk and Marc Levoy. 1994. Zippered Polygon Meshes from Range Images. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*. 311–318. https://doi.org/10.1145/192161.192241

[40] Polychronis Valentzas, Michael Vassilakopoulos, Antonio Corral, and Christos Antonopoulos. 2023. GPU-Based Algorithms for Processing the k Nearest-Neighbor Query on Spatial Data Using Partitioning and Concurrent Kernel Execution. *International Journal of Parallel Programming* 51, 6 (Dec. 2023), 275–308. https://doi.org/10.1007/s10766-023-00755-8

[41] Jordi L. Vermeulen, Arne Hillebrand, and Roland Geraerts. 2017. A comparative study of k-nearest neighbour techniques in crowd simulation. *Computer Animation and Virtual Worlds* 28, 3–4 (2017), e1775. https://doi.org/10.1002/cav.1775

[42] Bo Xiao and George Biros. 2016. Parallel Algorithms for Nearest Neighbor Search Problems in High Dimensions. *SIAM Journal on Scientific Computing* 38, 5 (Jan. 2016), S667–S699. https://doi.org/10.1137/15M1026377

[43] Jingrong Zhang, Akira Naruse, Xipeng Li, and Yong Wang. 2023. Parallel Top-K Algorithms on GPU: A Comprehensive Study and New Methods. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, CO, USA) *(SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 76, 13 pages. https://doi.org/10.1145/3581784.3607062

[44] Weijie Zhao, Shulong Tan, and Ping Li. 2020. SONG: Approximate Nearest Neighbor Search on GPU. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1033–1044. https://doi.org/10.1109/ICDE48307.2020.00094 ISSN: 2375-026X.

[45] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. 2008. Real-time KD-tree construction on graphics hardware. In *ACM SIGGRAPH Asia 2008 papers*. ACM, Singapore, 1–11. https://doi.org/10.1145/1457515.1409079

[46] Qingnan Zhou and Alec Jacobson. 2016. Thingi10K: A Dataset of 10,000 3D-Printing Models. arXiv:1605.04797 [cs.GR]

[47] Yuhao Zhu. 2022. RTNN: accelerating neighbor search using hardware ray tracing. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 76–89. https://doi.org/10.1145/3503221.3508409

[48] Yifan Zhu, Ruiyao Ma, Baihua Zheng, Xiangyu Ke, Lu Chen, and Yunjun Gao. 2024. GTS: GPU-based Tree Index for Fast Similarity Search. arXiv:2404.00966