Hybrid KNN-Join: Parallel Nearest Neighbor Searches Exploiting CPU and GPU Architectural Features

Michael Gowanlock*

School of Informatics, Computing, and Cyber Systems, Northern Arizona University, Flagstaff, AZ, U.S.A., 86011

Abstract

K Nearest Neighbor (*KNN*) joins are used in scientific domains for data analysis, and are building blocks of several well-known algorithms. *KNN*-joins find the *KNN* of all points in a dataset. This paper focuses on a hybrid CPU/GPU approach for low-dimensional *KNN*-joins, where the GPU may not yield substantial performance gains over parallel CPU algorithms. We utilize a work queue that prioritizes computing data points in high density regions on the GPU, and low density regions on the CPU, thereby taking advantage of each architecture's relative strengths. Our approach, HybridKNN-Join, effectively augments a state-of-the-art multi-core CPU algorithm. We propose optimizations that (*i*) maximize GPU query throughput by assigning the GPU large batches of work; (*ii*) increase workload granularity to optimize GPU utilization; and, (*iii*) limit load imbalance between CPU and GPU architectures. We compare HybridKNN-Join to one GPU and two parallel CPU reference implementations. Compared to the reference implementations, we find that the hybrid algorithm performs best on larger workloads (dataset size and K). The methods employed in this paper show promise for the general division of work in other hybrid algorithms.

Keywords:
GPGPU, Heterogeneous Systems, In-memory Database, Nearest Neighbor Search, Query Optimization

1. Introduction

This paper studies the *KNN* self-join problem, which is outlined as follows: given a database, *D*, of points, find all of the *K* nearest neighbors of each point. We focus on the self-join because it is a common task in scientific data processing workflows (e.g., within an astronomy catalog, find the closest five objects of all objects within a feature space [1]). *KNN* searches are used in many applications, such as the k-means [2], and Chameleon [3] clustering algorithms. Consequently, *KNN* searches have been well studied [4, 5, 6], including algorithms targeting the Graphics Processing Unit (GPU) [7].

There are several *KNN* research thrusts in the literature. *KNN* searches are employed in both low and high dimensional contexts. An example *KNN* query in the low dimensional case is as follows: find the closest *K* restaurants to my current position, where the feature vectors contain 2-dimensional coordinates of the locations of nearby restaurants. An example *KNN* search in the high dimensional context is image classification [8], where image pixel intensities are converted to feature vectors, which may contain hundreds or thousands of features.

To find the *KNN* of each point (or feature vector) in a dataset, one option is to perform a brute force search between all data points, which yields a quadratic complexity. Another option is to use an indexing data structure (e.g., kd-tree [9], or R-tree [10]), which prunes the search for points which are nearby a given query point (we refer to a query point as a point being searched to find its *K* nearest neighbors). In low dimensionality, the indexing data structures perform quite well and are able to discern between data points in each dimension, which reduces the quadratic complexity of the brute force algorithm [4, 5, 6, 7].

The high dimensional context leads to several problems concerning the "curse of dimensionality" [11]. In higher dimensionality, index searches typically become more exhaustive, where a *KNN* search for a given query point needs to compare to a substantial fraction of the points in the entire dataset. Thus, index searches become ineffective, and may even degrade performance relative to a brute force search, because searching the index incurs some degree of overhead. The exhaustive nature of high dimensional *KNN* searches led to the development of approximate algorithms [5, 6] that return *K* nearby neighbors of a given query point, but they may not be the exact *K* nearest neighbors.

In this work, we focus on exact *KNN* searches in low dimensionality. The performance of low dimensional *KNN* searches is limited by the memory bottleneck. However, the high aggregate memory bandwidth of modern GPUs [12] results in roughly an order-of-magnitude increase in memory bandwidth over the CPU. Therefore, GPUs are well-suited to data-intensive workloads. However, data transfers to and from the GPU are a well-known bottleneck, which can decrease the performance advantages that are potentially afforded by the GPU. Additionally, many data-dependent workloads, such as the *KNN*-join studied in this work, have irregular execution patterns that make

*Corresponding author. Michael Gowanlock, School of Informatics, Computing, and Cyber Systems (Building #90) 1295 S. Knoles Dr., Flagstaff, AZ, 86011, U.S.A. E-mail: michael.gowanlock@nau.edu the GPU unsuitable for the algorithm due to thread divergence and serialization that degrades performance [13]. Thus, it is not clear that the GPU will lead to performance gains over multi-core CPU algorithms designed for *KNN* searches in low dimensionality.

Despite the great potential of GPU-accelerated *KNN* algorithms, much of the literature focuses on optimizing brute force approaches which highlight performance in high dimensional feature spaces and often compute a distance matrix [14, 15, 16, 17]. The key idea is to compute the distance between a query point and all other points in D, then select the K neighbors with the smallest distances to the query point.

We depart from the distance matrix approach, and focus on low dimensional *KNN* searches within a dimensionality regime that can employ indexing data structures to prune the search for potential neighbors of each query point in the dataset. Given the above context, we summarize the goals of the paper as follows.

Addressing Low-Dimensionality on the GPU: The above-mentioned brute force *KNN* searches in high-dimensional feature spaces are clearly well-suited to the GPU as the many independent distance calculations can easily exploit the GPU's massive parallelism. But, it is not clear that the GPU can significantly outperform parallel CPU approaches in low dimensionality. We address *KNN* searches in up to 6-D, which is largely the domain of CPU *KNN* algorithms that employ indexing data structures.

Transforming the GPU-Accelerated Similarity Join into the KNN-Join: Recent work has proposed a similarity self-join for the GPU that finds all points within a search distance ϵ of a query point using an index [18]. The similarity join can be used to construct part of a *KNN* search by searching within a distance ϵ of a query point, and if there are $\geq K$ neighbors within ϵ, order the neighbors by distance and select the nearest K neighbors. We leverage an efficient GPU similarity join algorithm in our approach.

Concurrent Exploitation of CPU and GPU Resources: In contrast to GPU-only approaches, we use both the CPU and GPU by assigning query points to either architecture to find their respective *KNN*. We leverage the distance similarity join described above for the GPU to process high data density regions, and a parallel CPU *KNN* algorithm for processing low density regions.

To our knowledge, our algorithm is the first to split *KNN* searches between architectures as described above. The contributions and primary findings of the paper are outlined as follows:

- We propose a hybrid CPU/GPU approach for solving the *KNN* self-join problem that combines a distance similarity join for the GPU with a multi-core CPU *KNN* algorithm.
- The GPU component of our HybridKNN-Join algorithm solves the *KNN* problem using distance similarity searches. We show how to select a search distance, ϵ, such that the GPU join is likely to find at least K neighbors for each query point.
- We present a work queue to distribute query points to the CPU and GPU. The work queue prioritizes assigning query points with significant computation to the GPU.
- The throughput-oriented GPU requires processing large quantities of query points in batches to achieve peak performance. This can lead to load imbalance between the CPU and GPU. We propose methods to mitigate load imbalance between architectures.
- We compare the hybrid approach to one GPU and two multi-core CPU reference implementations. We find that HybridKNN-Join outperforms the reference implementations on most scenarios, particularly at larger values of K.

The paper is organized as follows. Section 2 presents background material. Section 3 recaps leveraged GPU self-join literature. Section 4 presents the hybrid *KNN* self-join and optimizations. Section 5 evaluates our approach. And finally, Section 6 concludes the paper and discusses future work directions.

2. Background

2.1. Problem Statement

The *KNN* self-join is outlined as follows. Let D be a database of n-dimensional points (or feature vectors) denoted as $p_i \in D$, where $i = 1, 2, \ldots, |D|$. For each point in the database, $p_i \in D$, we find its K nearest neighbors, excluding the point itself. To compute the distance between two points, p_a and p_b, we use the Euclidean distance as follows: $dist(p_a, p_b) = \sqrt{\sum_{j=1}^{n}(x_j^a - x_j^b)^2}$, where x_j^a and x_j^b denote the coordinates in dimension j of point a and b, respectively. The *KNN* self-join is denoted as $D \bowtie_{KNN} D$. However, the *KNN* self-join problem and optimizations are also directly applicable to the case where there are two datasets R and S that are joined, $R \ltimes_{KNN} S$. Table 1 summarizes notation that is used throughout multiple sections of the paper (notation that is self-contained within a section is not reported in the table).

Many data analytic and machine learning algorithms assume that processing can occur entirely in-memory. We make the same assumption here, and do not consider out-of-core (disk-based) solutions. For *KNN* searches, the result set size increases with K; therefore, the result set typically has the largest memory footprint of the algorithm. Since the result set may exceed GPU global memory capacity, we batch the execution across multiple GPU kernels to ensure that global memory is not exceeded. This allows our algorithm to process result set sizes that would otherwise exceed global memory capacity. We assume that the entire dataset can fit within the global memory of the GPU because the size of the dataset is much smaller than the result set.

2.2. Related Work

We present an overview of several categories of related work below.

Table 1: Algorithm components and summary of notation.

Names of Algorithms/Components	
HybridKNN-Join	The proposed CPU/GPU approach.
Hybrid-CPU	Parallel CPU component of HybridKNN-Join.
Hybrid-GPU	GPU component of HybridKNN-Join.
Notation	
D	The input dataset.
$p_i \in D$	Data points in the dataset.
n	The dimensionality of the data.
K	The number of nearest neighbors found for each $p_i \in D$.
ϵ	The search distance for Hybrid-GPU that may be dynamically expanded.
ϵ_{min}	The initial search distance selected by Hybrid-GPU.
n_{large}	Initial monolithic batch size for Hybrid-GPU.
n_{small}	Small batch size for Hybrid-GPU.
n_{Cwin}	Window size of reserved queries for Hybrid-CPU during monolithic batch rounds.
n_{CPU}	Batch size for Hybrid-CPU.
t	Number of threads assigned to each query point for Hybrid-GPU.

2.2.1. *Hybrid Algorithms*

A significant fraction of research on GPU algorithms and applications compares CPU vs. GPU approaches. However, many GPU algorithms are unsuitable for all application scenarios, where parallel CPU algorithms may outperform the GPU in some instances. Consequently, using both multi-core CPUs and the GPU is needed to achieve peak performance in heterogeneous systems (see [19] for a survey of hybrid algorithms). Thus, hybrid CPU/GPU algorithms aim to maximize resource utilization in computer systems, and/or are designed such that they can take advantage of the relative strengths of each architecture. This paper is one such example of a hybrid algorithm that both splits the work between the CPU and GPU which maximizes resource utilization and also assigns work to each architecture to exploit the CPU and GPU's relative strengths.

Several works split the work between the CPU and GPU at runtime. For instance, Li et al. [20] parallelize Cryo-EM 3D reconstruction, and assign tasks to the CPU or GPU depending on the workload. Deshpande et al. [21] filter images based on the degree of parallelism that varies across image regions, where the GPU is assigned the highly parallel regions and the CPU is assigned the remaining regions. Similarly to the abovementioned works, HybridKNN-Join dynamically schedules the query points onto the architecture most suitable for the workload. To our knowledge, our preliminary work [22] on hybrid *KNN*-joins is the only such *KNN* algorithm that uses a hybrid approach. This comprehensive paper extends our preliminary work [22].

2.2.2. *KNN Searches and Joins*

KNN searches are a fundamental machine learning algorithm. Consequently, there have been many works on optimizing the *KNN* search and join [4, 5, 23, 24, 25, 26, 14, 15, 27, 16, 28, 17, 6, 7, 29, 30]. In this section, we discuss the related work on *KNN* searches. For clarity, we note that we only consider those algorithms that are capable of performing *exact* searches. Approximate *KNN* searches are typically employed at higher dimensionality than that addressed in this paper.

We describe several *KNN* algorithms designed for the CPU as follows. An R-tree is used to find the *KNN* in [4] that prunes the search for nearby candidate points to a given query point. The algorithm performs a branch-and-bound tree traversal, which first gets an estimate of the *KNN* and then performs backtracking on subtrees to find the exact neighbors. Backtracking in tree-based solutions [4] is used to ensure that at least K nearest neighbors are found.

The Approximate Nearest Neighbors (ANN) algorithm can be used to efficiently find both the approximate and the exact neighbors [5]. Approximate solutions are motivated by prohibitively expensive high-dimensional exact *KNN* searches. Related to ANN is the Fast Library for ANN (FLANN) [6], which achieves good performance using a parallel search over a randomized kd-forest. While FLANN outperforms ANN for one scenario in [6], the comparison was between a parallel (FLANN) and sequential (ANN) algorithm. Since ANN is considered a state-of-the-art exact *KNN* algorithm, we parallelize and incorporate it into HybridKNN-Join.

There have been several efforts on parallelizing *KNN* searches on the GPU. We omit discussing the distance matrix based approaches [14, 15, 16, 17] described in Section 1, as we focus on lower dimensionality where indexing data structures are effective at pruning *KNN* searches. A *KNN* GPU implementation is presented by Nam et al. [7] that employs an R-tree index. Their algorithm performs backtracking in the R-tree to guarantee that K neighbors are found for each query point. The algorithm is optimized to reduce warp divergence that occurs when executing branch instructions that are necessitated by tree traversals.

A grid-based indexing solution for 3-dimensional *KNN* searches on the GPU was proposed by Leite et al. [27]. For a given query point, the algorithm expands the number of grid cells searched to ensure that at least K neighbors are found. While the approach in [27] has some similarity to the work in this paper (both use a grid-based index), the algorithm uses a query-centric approach that expands the search radius when $< K$ neighbors are found. In contrast, in this paper, we avoid using a query-centric approach and instead elect to execute all *KNN* searches in a given batch of queries with a fixed search distance to minimize warp divergence. Our batched execution relaxes the constraint that $\geq K$ neighbors need to be found by each query point computed by the GPU.

Reference Implementation: A GPU *KNN* search algorithm that uses a buffer kd-tree has been proposed by Gieseke et al. [28]. Similarly to the R-tree algorithm described above [7], the buffer kd-tree algorithm [28] avoids several drawbacks of the GPU's architecture. In particular, they search for queries in batches that are co-located within the same leaf. The algorithm delays execution by waiting for sufficient work to be accumulated into a buffer before accessing leaf nodes. This improves the SIMT parallelism of the algorithm. Their algorithm also focuses on improving the fraction of coalesced memory accesses by having threads within a warp access either consecutive or nearby memory addresses. In our experimental evaluation, we compare our hybrid algorithm to this GPU algorithm

Table 2: Categorization of several *KNN* search algorithms in the literature. Categories: dimensionality, use of indexing methods, brute force searches, exact and approximate solutions, and architecture.

Reference	Low to Moderate Dimensionality	High Dimensionality	Indexing	Brute Force	Exact	Approx.	CPU	GPU	Hybrid CPU/GPU
Roussopoulos et al. [4]	X		X		X		X		
Arya et al. [5]	X		X		X	X	X		
Gieseke et al. [28]	X		X		X			X	
Nam et al. [7]	X		X		X			X	
Patwary et al. [29]	X		X		X		X		
Ryoo et al. [30]	X		X		X			X	
Leite et al. [27]	X		X		X	X		X	
Gowanlock [22]	X		X		X				X
Muja & Lowe [6]		X	X			X	X		
Garcia et al. [14]		X		X	X			X	
Arefin et al. [15]		X		X	X			X	
Jian et al. [16]		X		X	X			X	
Komarov et al. [17]		X		X	X			X	

that we denote as BufferKDTree.

Table 2 categorizes related work on *KNN* searches by the target dimensionality of the algorithm, whether the algorithm uses an indexing data structure to prune the search, or uses a brute force approach, the accuracy of the search (exact or approximate), and the target architecture (CPU, GPU or hybrid CPU/GPU). From Table 2, we observe that the majority of the low/moderate dimensionality works use indexing schemes, whereas the high dimensional algorithms focus on brute force approaches. As described in Section 1, this is because the efficacy of indexing schemes to prune the search degrades at high dimensionality.

2.2.3. Indexing Techniques

Central to our approach is using an appropriate index for the architecture. Indexes for the CPU have been designed to be work-efficient and data-aware, where spatial partitions are computed based on the input data. Examples include well-known tree-based indexes, such as kd-trees [9], quad-trees [31], and R-trees [10, 32]. In contrast, there are data-oblivious methods, such as statically partitioned grids [18].

With the proliferation of general purpose computing on graphics processing units (GPGPU) there has been debate whether the community should use the tree-based approaches, or data-oblivious methods for the GPU. The disadvantage of index-trees is that searching the index using tree traversals requires performing many branch instructions, which can reduce the parallel efficiency of the GPU due to the SIMT architecture. Consequently, the abovementioned GPU *KNN* algorithms that employed index-trees [28, 7] were optimized to avoid divergence.

While there is little consensus regarding the type of index that should be employed on the GPU, we highlight two results. A GPU R-tree [33] was proposed and optimized to reduce thread divergence. Later, the same research group showed that it is better to perform the tree traversal on the CPU and perform the scanning of the leaf nodes on the GPU [34]. This shows that the GPU should be leveraged through the use of regularized instructions, yielding low thread divergence. Therefore, we elect to use a non-hierarchical indexing technique for the GPU component of our *KNN*-join algorithm.

2.2.4. Range Queries and Joins

Our hybrid approach uses range queries (distance similarity searches) on the GPU to perform *KNN* searches. A join operation with a distance predicate can be implemented as several range queries. The multi-core CPU join algorithm in [35] uses a non-materialized grid, and exploits the data distribution to efficiently perform a similarity join over a search distance, ϵ, and the algorithm was shown to outperform the E^2LSH [24], and LSS [36] algorithms. A GPU self-join was presented in [18] that was shown to be efficient on low-dimensional data. We leverage some of the optimizations in the GPU self-join work [18] as they are effective for executing range queries that can be used to solve *KNN* searches on the GPU.

2.2.5. Distributed Memory KNN Searches

Distributed-memory approaches have been proposed to improve the performance of *KNN* searches. For instance, MapReduce [37] implementations for *KNN* joins [38, 39] have been proposed. The authors in [38] optimize the mapping function to prune distance calculations, which reduces the cost of the shuffling operation and computation. The authors in [39] propose exact and approximate *KNN* join solutions, where they show that in their approximate solution, only a linear number of reducers are needed, which is a prerequisite for achieving good scalability. In contrast to scaling out the computation using distributed memory, we scale up the computation using the GPU.

2.3. Application Scenario of this Paper

There is a wide range of related work and application scenarios for *KNN* searches. In this paper, we focus on low to moderate dimensionality *KNN* searches (2–6 dimensions), where the *curse of dimensionality* [11] does not prohibit indexing schemes from being effective at pruning *KNN* searches. This scenario is common in the literature (e.g., see the *KNN* search papers in Table 2 by Roussopoulos et al. [4], Arya et al. [5], Gieseke et al. [28], Nam et al. [7], Patwary et al. [29], Ryoo et al. [30], and Leite et al. [27]). Additionally, we focus on *exact* and not approximate searches, as the approximate searches are designed for much higher dimensionality than that considered in this paper. Furthermore, we reiterate that we depart from the literature by splitting the work between CPU and GPU architectures.

3. Recap of Previous Self-Join Work

HybridKNN-Join leverages the distance similarity self-join work of Gowanlock & Karsin [18], which was evaluated on up to $n = 6$ dimensions. The authors used an efficient indexing scheme and batching scheme from [40], and proposed a technique to reduce the number of duplicate distance calculations. The approach was shown to outperform a state-of-the-art multi-core approach across many experimental scenarios; therefore, we employ their work in the GPU component of HybridKNN-Join. We outline the optimizations from [18], that we use to efficiently solve the *KNN*-join on the GPU.

3.1. Indexing Technique

We use a grid-based indexing scheme for the GPU (see [18] for more detail) with cells of length ϵ. The index is constructed on the host/CPU, and only stores non-empty grid cells, as indexing all cells may exceed the memory capacity of the GPU. The index, denoted as G, uses a series of lookup arrays to find relevant points in the index. A range query around a query point is carried out by performing distance calculations between points in each adjacent cell of the query point (and the cell containing the query point). The number of adjacent cells is 3^n (e.g., in 2-D there are 9 total adjacent grid cells). We make one minor change to the index described in previous work [18], by removing the masking arrays, which were used to filter out cells that did not contain any points in a given dimension. While the masking arrays may be useful in some scenarios (such as datasets with a bimodal distribution in a single dimension), in practice, we find that they had a negligible impact on performance. The space complexity of the index is $O(|D|)$. This small memory footprint allows for larger datasets and result set sizes to be processed on the GPU. While we use this grid-based indexing technique, we modify its construction, as discussed in Section 4.6.

3.2. Batching Scheme

We give a brief overview of the GPU batching scheme in [18]. The size of the total result set for a join operation, which contains the neighbors of each point within a distance ϵ, can be larger than the GPU's global memory capacity. To process large datasets or values of ϵ, a batching scheme is needed to incrementally process the join, by querying a fraction of D at each kernel invocation until range queries have been performed on all $p_i \in D$. We select a number of batches to execute by first estimating the total result set size (using a lightweight kernel), which yields an estimate, e, of the total result set size. Given a buffer size of b_s (the size of a buffer to store the result set of a batch), we compute the total number of batches to be $n_b = \lceil e/b_s \rceil$. This obviates failure-restart strategies that can waste computation. We use 3 CUDA streams (a minimum of $n_b = 3$), which overlaps the execution of the kernel and data transfers to exploit bidirectional PCIe bandwidth, and concurrent computation on the host. We use $b_s = 10^8$ for each stream. In our experiments, on the larger workloads (higher dimensionality and K), using multiple streams is able to hide most of the host-GPU communication, whereas on the smaller workloads, the algorithm is bound by memory transfer operations.

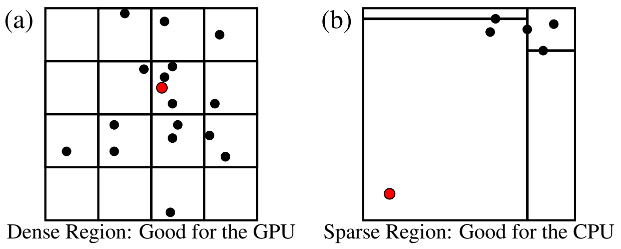

(a)
(b)
Dense Region: Good for the GPU
Sparse Region: Good for the CPU

Figure 1: Example query points assigned to either the GPU or CPU and possible indexing strategies for each. (a) The GPU is proficient at processing high density regions with a non-hierarchical grid. (b) The CPU is proficient for low density regions with an index-tree (kd-tree partitioning shown).

4. HybridKNN-Join and Optimizations

4.1. Splitting Work Between Architectures

As discussed in Section 1, we focus on a hybrid CPU/GPU approach that performs the *KNN* search using the CPU and GPU.

A similarity search finds all points, $p_i \in D$, within a search distance, ϵ, of a query point. Thus, to construct a *KNN*-join using a range query, there are several facets of the problem to consider. The ϵ search distance is required to ensure that the nearest points from a query point are found. For a given search that returns $> K$ neighbors, the distances between points are compared to determine which of the points are nearest to the query point. However, while a range query will return all points within ϵ of a batch of query points, there is no guarantee that all (or any) of the query points will have K neighbors. In principle, the selection of ϵ could be large such that all points have at least K nearest neighbors; however, this would lead to significant computational overhead, as some points in the dataset may find a large fraction of the entire dataset necessitating a significant number of distance calculations.

Figure 1 shows an example of a spatially partitioned region with query points shown as larger red points. In Figure 1(a), there are many nearby neighbors to the query point; thus, a significant number of distance calculations need to be computed to find the K nearest neighbors. However, in Figure 1(b), the query point is located in a sparse region. Thus, a large range query would be needed to find at least K neighbors. Spatially partitioning the data using a grid in Figure 1(a) is reasonable, as it is likely K neighbors will be found by checking adjacent cells (e.g., assume $K = 3$). In contrast, in Figure 1(b), the grid is not effective. Had a grid been used, the adjacent cells would not contain any nearby points. In this case, a data-aware index (e.g., kd-tree [9] partitioning shown in Figure 1(b)) is better suited to finding data in sparse regions. Furthermore, as there are fewer points nearby the query point in Figure 1(b), there is a low degree of candidate point filtering overhead.

Given this illustrative example, the GPU and associated indexing scheme in Section 3.1 is good for processing the scenario in Figure 1(a) due to the large amount of filtering overhead needed (the massive parallelism of the GPU is well-suited to distance calculations), and low index search overhead; whereas the scenario in Figure 1(b) is good for finding the *KNN* on the CPU due to the low degree of filtering overhead and associated data-aware indexing scheme for low density regions. Therefore, the motivation for splitting the work between CPU and GPU is based on the suitability of each architecture to find the *KNN* of a given query point.

4.2. Hybrid KNN-Join Overview

We exploit the relative strengths of CPU and GPU architectures. The GPU is proficient at processing large batches of queries when the kernel can exploit the high memory bandwidth and massive parallelism afforded by the architecture. The CPU is better at processing irregular instruction flows, and thus, is well-suited to tree-based indexes that are comprised of many branch instructions.

*4.2.1. CPU KNN Component (*Hybrid-CPU*)*

We use the publicly available[1] ANN CPU implementation [5] that uses a kd-tree index. The algorithm is efficient for both approximate and exact solutions to the *KNN* problem, and we execute the algorithm such that we obtain the exact nearest neighbors. As noted in other work [29], ANN uses global variables in its functions, which are not conducive to shared-memory parallelism. We obviate this limitation by parallelizing ANN using MPI where the K nearest neighbors of query points are found independently by each process rank. The results are written directly to an MPI shared memory window and thus we avoid explicit communication between process ranks. We refer to the multi-core CPU approach of HybridKNN-Join as Hybrid-CPU.

*4.2.2. GPU-Join Component (*Hybrid-GPU*)*

In CPU-based *KNN* searches [4], backtracking is used to ensure that K neighbors are found for each query point searched. Likewise, the E^2LSH [24] CPU algorithm for range queries has been used for *KNN* searches by expanding the search radius until $\geq K$ neighbors are found for each point. As an example of expanding the search radius, Figure 2(a) shows where $K = 5$ neighbors are found when $\epsilon = 1$, whereas Figure 2(b) shows an example where ϵ needs to be expanded to $\epsilon = 2$ to find at least $K = 5$ neighbors. Backtracking or expanding the search radius is a query-centric approach that is beneficial for modern CPUs that can take advantage of the memory hierarchy (e.g., benefiting from locality during tree traversals), but is unsuitable for a batched GPU execution.

To transform range queries with a distance ϵ into a *KNN* search that considers the throughput-oriented nature of the

[1]ANN can be found here: `http://www.cs.umd.edu/~mount/ANN/`.

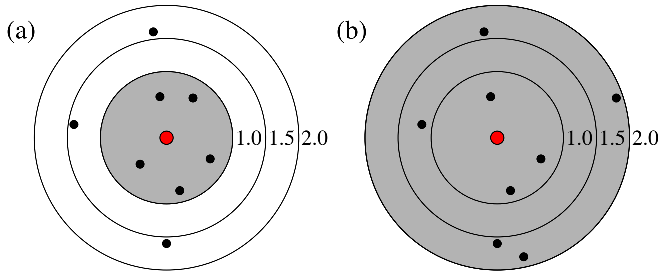


Figure 2: A *KNN* search around two query points (larger red points at the centers) where $K = 5$. Shaded region denotes the range required to find $K = 5$ points. (a) $K = 5$ neighbors are found with $\epsilon = 1$. (b) $K = 5$ neighbors are found when the search distance is expanded to $\epsilon = 2$.

GPU, we use a batched execution that allows our GPU component, Hybrid-GPU, to fail to find at least K points for each point searched. The overall idea that we will outline in Section 4.5 is the following: (*i*) the failed queries are added back to a work queue to be processed by either Hybrid-GPU or Hybrid-CPU in the future; and (*ii*) we dynamically re-index Hybrid-GPU with an increased ϵ value when it reaches a threshold number of searches that did not yield $\geq K$ neighbors per point. Thus, each query point assigned to Hybrid-GPU is not guaranteed to find its *KNN* because we use a single ϵ-distance when executing the kernel. Therefore, we refrain from using the query-centric approaches (e.g., backtracking, or increasing ϵ for individual point searches) on the GPU because this would lead to increased divergence in the kernel and intra-warp load imbalance.

4.3. Algorithm Overview

We present the pseudocode of HybridKNN-Join to provide an overview of the technical details of the algorithm which are described later in this section. We outline HybridKNN-Join in Algorithm 1 as follows. Obtaining the process rank and importing the dataset occurs on lines 2–3. We use an MPI implementation and have 1 master GPU rank and several CPU ranks which begin their primary execution on lines 4 and 23, respectively. For brevity, we do not show the work queue rank, as it simply assigns query points to the GPU and CPU ranks.

The Hybrid-GPU rank initializes the result set (line 5), and query failure set (line 6). Next, the value of ϵ_{min} is selected (Section 4.4) on line 7, and then ϵ is set using this value on line 8 (we use ϵ_{min} later, which is why we declare both ϵ and ϵ_{min}). Next, we construct the index, G, as a function of D, and ϵ on line 9. Then, the algorithm gets a number of queries from the work queue rank on line 10 and stores them in Q_{GPU}. A while loop is entered on line 11 that iterates until there are no more queries to compute (i.e., $|Q_{GPU}| = 0$). Using the batch estimator, the number of GPU batches is computed on line 12 (recall from Section 3.2 that the batch estimator computes the total number of batches so that Hybrid-GPU can process result sets larger than global memory). For clarity, note that these batches differ from the batches of queries obtained from the work queue (Q_{GPU}).

The algorithm loops over all of the batches (line 13). At each iteration, the GPUJoinKernel is executed (line 14), which com-

Algorithm 1 HybridKNN-Join Algorithm

```
1: procedure HybridKNN-Join($K$, $b_s$)
2:     myRank ← getRank()
3:     $D$ ← importData()
4:     if myRank = GPU Master Rank then                              ▷ GPU Rank
5:         KNNresult ← $\emptyset$
6:         $Q_{Fail}$ ← $\emptyset$
7:         $\epsilon_{min}$ ← selectEpsilon($D$)
8:         $\epsilon$ ← $\epsilon_{min}$
9:         $G$ ← constructIndex($D$, $\epsilon$)
10:        $Q_{GPU}$ ← getWork()
11:        while $|Q_{GPU}| > 0$ do
12:            $n_b$ ← computeNumGPUBatches($b_s$, $Q_{GPU}$, $\epsilon$)
13:            for $i \in 1, 2, \ldots, n_b$ do
14:                kernResult[i] ← GPUJoinKernel($D$, $Q_{GPU}$, $G$, $\epsilon$, $i$)
15:                KNNresult ← KNNresult ∪ filterKeys(kernResult[i])
16:                $Q_{Fail}$ ← $Q_{Fail}$ ∪ findFailedPnts(kernResult[i], $Q_{GPU}$)
17:            addFailuresToWorkQueue($Q_{Fail}$)
18:            if $|Q_{Fail}|/|Q_{GPU}| > 0.25$ then
19:                $\epsilon$ ← $\epsilon + 0.5\epsilon_{min}$
20:                $G$ ← constructIndex($D$, $\epsilon$)
21:            $Q_{GPU}$ ← getWork()
22:            $Q_{Fail}$ ← $\emptyset$
23:    else                                                          ▷ CPU Ranks
24:        $Q_{CPU}$ ← getWork()
25:        while $|Q_{CPU}| > 0$ do
26:            KNNresult ← KNNresult ∪ Hybrid-CPU ($Q_{CPU}$, myRank)
27:            $Q_{CPU}$ ← getWork()
28:    return

29: procedure GPUJoinKernel($D$, $Q_{GPU}$, $G$, $\epsilon$, $i$)
30:    resultSet ← $\emptyset$
31:    gid ← getGlobalId($i$)
32:    queryPoint ← getPoint(gid, $Q_{GPU}$)
33:    adjCells ← getAdjCells($G$, queryPoint)
34:    for cell ∈ adjCells.min,…,adjCells.max do
35:        pntResult ← pntResult ∪ calcDistancePts(queryPoint, cell, $\epsilon$)
36:    resultSet ← resultSet ∪ pntResult
37:    return resultSet
```

putes the result set for a single batch. On line 15 the result of the join operation is filtered (the result is in the form of key/value pairs which are filtered to reduce duplicate keys), and store only points in Q_{GPU} that have at least K neighbors. On line 16, those query points in the batch that have $< K$ neighbors are added to the Q_{Fail} set, and these queries are added back to the work queue on line 17.

On lines 18–20, the algorithm will dynamically re-index Hybrid-GPU with a larger ϵ value if $\geq 25\%$ of points in Q_{GPU} found $< K$ neighbors (Sections 4.2.2 and 4.5). And finally, on line 21, the rank retrieves work for the next batch from the work queue, and the Q_{Fail} buffer is reset on line 22.

Regarding Hybrid-CPU, on line 24, queries are obtained from the work queue rank. If there are queries to process, a while loop is entered on line 25, which computes the result of the *KNN* search for its batch of queries on line 26. The next batch of work is obtained from the work queue rank on line 27, and the loop continues until their are no additional queries to compute.

We describe the Hybrid-GPU join kernel, but refer the reader to [18] for more detail. We make two minor changes to the self-join kernel to accommodate HybridKNN-Join. First, we add a query set, as we do not want to compare all points to each other, as range queries are only needed for those points in Q_{GPU}. Second, we allow multiple threads to process an individual point (Section 4.7). In the GPU join kernel shown in Algorithm 1, the result set is initialized (line 30), and then the global thread id is computed (line 31). Next, the query point assigned to the thread is stored (line 32), and a loop iterates over all adjacent cells (lines 33–34). The point assigned to the thread is compared to all points in the adjacent cells, where a result is stored when a point is found to be within ϵ of the query point (lines 35–36). The result is stored as key/value pairs, where the key is the query point id, and the results are both the point id within ϵ of the key, and the distance between the points.

If more than one thread computes the distance between a query point and points in neighboring cells, then each thread only computes a fraction of the points in the cell on line 35 (Section 4.7).

4.4. Hybrid-GPU*: Selecting the Search Distance*

The input parameter to a *KNN* search is K; but Hybrid-GPU needs an ϵ-distance which is expected to find at least K neighbors for each point. Analytically deriving ϵ is feasible when the input data distribution is known. However, real-world datasets have data distributions that make an analytical approach intractable.

Consider a search distance, ϵ_{min}, that *on average* finds K neighbors per $p_i \in D$. Therefore, some points will find $\geq K$ neighbors, and some will find $< K$ neighbors. We derive ϵ_{min} which is used as an initial search distance for Hybrid-GPU.

We rely on the execution of two GPU kernels that sample the dataset to determine a good value of ϵ. First, we simply sample 0.1% of D, and compute the mean distance between points, denoted as ϵ_{mean}. Next, we define a number of bins, n_{bins}, that store the frequency of the distances between pairs of points that fall within the distance bin, where the width of each bin is ϵ_{mean}/n_{bins}. We then select a sample of the points in the dataset and compute the distance between each of these points and every other point in D, and store the distances in the respective bin, where any distance $> \epsilon_{mean}$ is not stored (using a search distance of ϵ_{mean} will return a large fraction of the dataset; much larger than any reasonable value of K).[2] We compute the cumulative number of points in each bin. Let $\mathcal{B}_d$ denote the distance bins, where $d = 1, 2, \ldots, n_{bins}$. Each $\mathcal{B}_d$ stores: (*i*) its distance range denoted as $[\mathcal{B}_d^{start}, \mathcal{B}_d^{end})$, where $\mathcal{B}_d^{start} = (d-1)\cdot(\epsilon_{mean}/n_{bins})$, and $\mathcal{B}_d^{end} = d\cdot(\epsilon_{mean}/n_{bins})$; (*ii*) the number of points found within its distance range $[\mathcal{B}_d^{start}, \mathcal{B}_d^{end})$, denoted as $\mathcal{B}_d^n$; (*iii*), and the cumulative number of points in the bin (including bins with points at lower distances), denoted as $\mathcal{B}_d^c$, where $\mathcal{B}_d^c = \sum_{a=1}^{d} \mathcal{B}_a^n$. This yields a relationship between the search distance and the average number of neighbors that will be found. ϵ_{min} corresponds to the query distance that yields

[2]In the experimental evaluation, the datasets range from 10^7 and 2.5×10^7 data points. We simply sample 500 points regardless of the dataset size. Since each of these points is compared to every other point in the dataset, a small sample size is sufficient to compute with high accuracy the cumulative number of points in each bin.

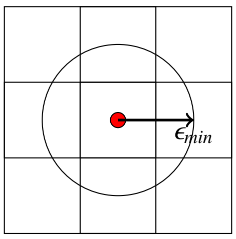


Figure 3: A 2-D example of the search radius ϵ_{min}, which probabilistically contains K neighbors per $p_i \in D$.

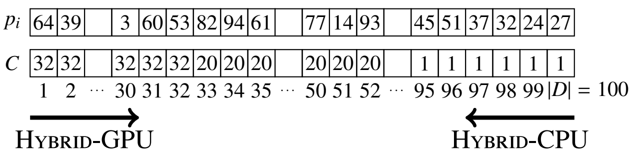


Figure 4: Example of a work queue with $|D| = 100$ data points. An array, C, stores the number of points within each cell for each $p_i \in D$. C is sorted in non-increasing order, where Hybrid-GPU is assigned points with the greatest amount of work, and Hybrid-CPU is assigned points with the least amount of work.

K cumulative neighbors, where $\epsilon_{min} = (\mathcal{B}_d^{start} + \mathcal{B}_d^{end})/2$, where $\mathcal{B}_{d-1}^c < K \leq \mathcal{B}_d^c$.

We select $\epsilon = \epsilon_{min}$, which on average finds K neighbors for each searched point. Figure 3 shows a 2-D example of a search within the grid, where the grid cell length is equal to the search radius and thus the search is bound to adjacent cells (Section 3.1).

4.5. Assigning Work using a Work Queue

The GPU should execute range queries for points in dense regions, and the CPU should perform the *KNN* search in sparse regions (Figure 1). We begin by estimating the total amount of work required to execute each $p_i \in D$. We repurpose the grid index that is sent to the GPU (Sections 3.1 and 4.4) to estimate the total work. For each $p_i \in D$, we check the total number of points that are found within the point's grid cell. This information requires simply performing a scan over the GPU index's non-empty grid cell array. For each point found within a given cell, the total number of points found within the cell are assigned to each point as an approximation of the amount of work that will need to be computed for that point. Then, we sort this array in non-increasing order by the number of points in each cell. Since the number of points in a cell will trace the data density in the immediate region around each point, this yields an estimate of the total amount of work for each point. Alternatively, we could count the number of neighbors in each point's cell and the adjacent cells to obtain a more accurate estimate of the amount of work computed by each query point; however, this would require substantial work, and thus we employ the simple procedure outlined above to estimate the work required of each point.

The GPU is efficient at performing distance calculations in high density regions, and the CPU is efficient at computing the lower density regions. Figure 4 shows a work queue illustration, where an array C stores the number of points within the cell of each point $p_i \in D$. For example, p_{64} and p_{53} both have 32 points in their cell. In contrast, the last point in C, p_{27}, only has a single point in its cell (itself). The work queue assigns Hybrid-CPU query points starting at $C[|D|]$ in decreasing order, and assigns Hybrid-GPU query points starting at $C[1]$ in increasing order. Thus, the queries assigned to the CPU progressively require more work, and the queries assigned to the GPU progressively require less work. Depending on the data distribution, Hybrid-GPU may only compute the *KNN* of a small fraction of D, but perform similar levels of work as Hybrid-CPU.

We outline several work queue performance considerations as follows.

- *Load Imbalance* – Performance degrades while one architecture (CPU or GPU) waits for the other to finish processing their queries.
- *Work Queue Overhead* – While the smallest work unit (a single query point) would lead to the best load balancing, there is overhead when accessing a work queue, and thus assigning batches of queries reduces work queue overhead. This is independent of the architecture requesting work to compute.
- *Maintaining GPU Throughput* – The GPU requires large batches of queries to maintain high query throughput, as executing a single query point on the GPU will underutilize its resources. In contrast, the CPU does not suffer from this limitation.

The work queue performance considerations are similar to the classical trade-off between load imbalance and work queue overhead (e.g., static vs. dynamic scheduling of for loops in OpenMP [41]). However, this scenario is different than this classical scenario, as the GPU requires larger query batches than the CPU to maintain high throughput. This can negatively impact load balancing, as the GPU may be assigned a large batch of points to compute towards the end of the computation, which would leave the CPU cores idle while waiting for the GPU to complete its work.

We propose several design decisions for the work queue to mitigate load imbalance while maintaining high GPU query throughput.

We allow Hybrid-GPU to be assigned two types of batches: (*i*) large monolithic batches containing a substantial fraction of $p_i \in D$; and, (*ii*) small batches. For a derived ϵ value (Section 4.4), Hybrid-GPU may not find the *KNN* for each point assigned to it (Section 4.2.2). Each $p_i \in D$ that fails to find its *KNN* is added back to the work queue, and may be found by either Hybrid-GPU (when ϵ is expanded) or Hybrid-CPU in the future. At each monolithic batch round, we reduce the batch size by a factor of two. We denote n_{large} to be the size of the monolithic batch as a fraction of $|D|$.

A drawback of the monolithic batches is that Hybrid-GPU can request many query points to compute and starve the CPU (Hybrid-CPU) of work. Consequently, we implement a window

Table 3: Summary of HYBRIDKNN-JOIN algorithm reconfiguration operations that occur at runtime.

	When	Why	Where
Increasing ϵ and Re-indexing	The number of failed queries on the previous batch exceeds 25% of the queries assigned in that batch.	Re-indexing occurs because the search distance, ϵ, is increased to reduce the number of failed searches.	GPU
GPU Decreasing Batch Size: Monolithic Batches	At each batch assigned to the GPU.	The GPU is initially assigned a very large batch of work to compute (n_{large}). The batch size assigned to the GPU decreases at each batch to mitigate against load imbalance between the CPU and GPU until $n_{large} = 0$ indicating that the monolithic batch rounds have completed.	CPU (work queue) and GPU
GPU Small Batch Sizes	At each batch assigned to the GPU after $n_{large} = 0$.	The GPU executes smaller batches of size n_{small}.	CPU (work queue) and GPU
CPU and GPU Smallest Batch Sizes	95% of the queries have been completed.	CPU and GPU batch sizes are decreased to $n_{CPU}/2$ and $n_{small}/2$, respectively, to mitigate against load imbalance between the CPU and GPU.	CPU and GPU

of reserved query points for the CPU to compute during monolithic batch processing. Thus, each time the GPU requests a monolithic batch, the work queue manager determines the maximum number of GPU points that can be assigned to HYBRID-GPU, such that the CPU has at least a minimum number of points to compute. We denote the size of the fraction $|D|$ points reserved for HYBRID-CPU as n_{Cwin}.

Using n_{Cwin}, and the fraction $|D|$ points that have already been processed by the CPU and GPU, denoted as n_{Cproc} and n_{Gproc}, respectively, if we let n^{l}_{large} be the size of the monolithic batch at round l, then the size at round $l + 1$ is as follows:

$$n^{l+1}_{large}=\min[0.5n^{l}_{large}, \max(0, 1 - n_{Gproc} - n_{Cproc} - n_{Cwin})].$$

Therefore, the monolithic batch size at round $l + 1$ is either half the size of the monolithic batch at l, or a smaller size, as a function of the fraction of queries already computed and the window of reserved queries, until $n_{large} = 0$.

Once the monolithic batch size decreases to $n_{large} = 0$, HYBRID-GPU reverts to smaller batches and no queries are reserved for HYBRID-CPU ($n_{Cwin} = 0$), such that: (*i*) the GPU is still utilized; and, (*ii*) the GPU and CPU finish their computation at similar times. However, there may be a substantial number of queries to compute despite (potentially) executing several monolithic batches, as the CPU window will have reserved queries from being added to monolithic batches. We denote n_{small} as the size of each smaller HYBRID-GPU (non-monolithic) batch, and n_{CPU} as the size of each HYBRID-CPU batch, both given as a fraction of $|D|$.

HYBRID-GPU may fail to find the *KNN* for many points if ϵ is not increased. As C stores points from most to least work, with each processed GPU batch, there are more query points that fail to find their *KNN*. Thus, when using the small or monolithic batches, we dynamically re-index HYBRID-GPU by increasing ϵ by a distance of $\epsilon_{min}/2$, when on the previous batch, HYBRID-GPU failed to find the *KNN* of at least 25% of its assigned points. This dynamic approach attempts to reach a trade-off between (*i*) not increasing ϵ too much which is expensive; and, (*ii*) not failing to find too many query points in the batch. Re-indexing occurs in parallel on the GPU to reduce the time where the GPU is idle due to expanding ϵ. Finally, when 95% of the query points have found their *KNN*, we then decrease the batch sizes assigned to the CPU and GPU to $n_{CPU}/2$ and $n_{small}/2$, respectively. These smaller batches (half of the initial size) mitigates load imbalance at the end of the computation.

Figure 5 illustrates the monolithic batches from the work queue being assigned to HYBRID-GPU and small batches of queries assigned to HYBRID-CPU. Figure 5(a) shows an initial work queue, where $1/3$ of D ($n_{large} = 1/3$) is assigned to HYBRID-GPU, and $1/3$ of the queries must be reserved for the CPU ($n_{Cwin} = 1/3$). In Figure 5(b), after HYBRID-GPU processes its queries from the first batch, some of the queries will be complete (the *KNN* were found for these query points) and some will have failed to find the *KNN*. Hence, because there are a mix of complete and incomplete queries, we show this as partially complete in the figure. The vertical lines denote n_{large} (dashed line) and n_{Cwin} (solid line). The CPU is guaranteed to find the *KNN* of each query point, thus the queries are shown as complete. Comparing Figure 5(a) and (b) we see that the maximum GPU batch size does not increase substantially because n_{large} is halved between rounds. Comparing Figure 5(c) and (d), the window of reserved CPU queries decreases the queries available for HYBRID-GPU to compute using a monolithic batch. After $n_{large} = 0$, HYBRID-GPU reverts to smaller batches of size n_{small}. Table 3 summarizes algorithm reconfiguration operations that occur during execution.

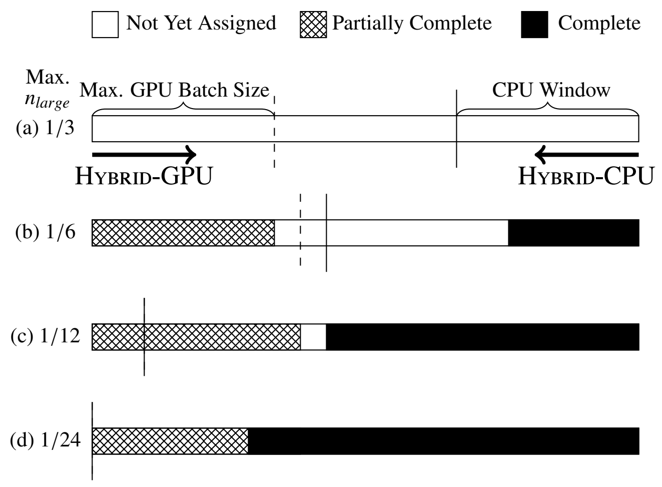
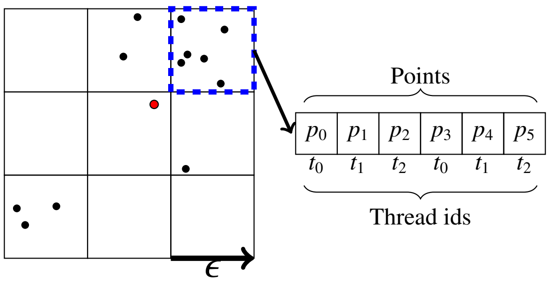


Figure 5: Assigning monolithic batches of queries from the work queue to Hybrid-GPU and queries to Hybrid-CPU (small Hybrid-GPU batch rounds not shown). (a) Initial work queue with $n_{large} = n_{Cwin} = 1/3$. (b) After processing a monolithic batch, some queries have been computed by Hybrid-GPU and Hybrid-CPU, and the monolithic batch size deceases. (c) The CPU window reduces the monolithic batch size. (d) After processing with Hybrid-GPU the monolithic batch rounds are finished as $n_{large} = 0$.

Note that we have made several parameter selection decisions. We dynamically re-index Hybrid-GPU when 25% of queries fail to find at least K neighbors in the previous batch. If we were to use a value >25%, then the number of failed queries would increase, whereas if we use <25% then the algorithm would spend more time re-indexing. Furthermore, we use half of the small GPU batch sizes (n_{small}), and the CPU batch size (n_{CPU}) when 95% of the queries have found their *KNN* in the dataset to obviate load imbalance at the end of the computation. While these parameters are arbitrarily selected, we believe that they are reasonable design decisions (e.g., similarly, OpenMP guided scheduling reduces the chunk size with increasing iteration [41]). In the experimental evaluation, we quantify the load imbalance between the CPU and GPU architectures, which demonstrates that the selection of parameters does indeed yield low load imbalance. The low load imbalance partially justifies the selection of these parameter values.

4.6. GPU: Indexing

As discussed in Section 3.1, we use a grid-based index for the GPU. In the previous work on the similarity self-join that we leverage, the index was constructed on the host/CPU, because we only needed to construct the index once. However, since finding the *KNN* on the GPU may require dynamically expanding ϵ several times, constructing the index on the host may become a bottleneck and degrade performance. Additionally, if the algorithm constructs the index on the host, then this reduces the resources available for the CPU component of the algorithm. Consequently, while we use the index described in previous work in Section 3.1, we construct the index on the GPU. We find that index construction on the GPU is much faster than constructing the index on the host. Our preliminary work on *KNN*-joins [22], that this paper extends, constructed the index on the host, and we found that it reduced performance particularly on lower values of K (smaller workloads), where the ratio of re-indexing overhead to computation is high. On the larger workloads, re-indexing overhead on the host is amortized across the entire computation and only has a minor impact on performance.



Points
p_0 p_1 p_2 p_3 p_4 p_5
t_0 t_1 t_2 t_0 t_1 t_2
Thread ids
ϵ

Figure 6: Using multiple threads to compute the distances between points in 2-D.

4.7. GPU: Optimizing Task Granularity

To improve resource utilization for parallel algorithms executed on the CPU or GPU, a common strategy is to reduce the size of each task to a smaller unit of work, and then redistribute these smaller tasks to threads or processes [42]. We describe a similar optimization as follows.

In the self-join work that we leverage [18], a single thread is assigned to each point in the dataset, where the thread finds all points within ϵ of its assigned point. This approach was tenable because the total number of threads is large ($|D|$). Since Hybrid-GPU may only process a small fraction of D in a batch, then the GPU's resources may be underutilized if we use one thread per point. Also, the GPU hides high memory latency by performing fast context switching between resident threads. Thus, oversubscribing the GPU by using more threads than cores is needed to saturate resources.

We divide the work of the distance calculations for a single point between multiple threads to increase task granularity. Figure 6 shows an example of using multiple threads per query point. The query point (red) is shown in the middle cell. The distances between the query point and the six points are computed in an adjacent cell (dashed blue outline). This example shows three threads each computing the distances between two points.

We assign a static number of threads per query point for performing the distance calculations, where the number of threads are referred to as t (e.g., $t = 32$ denotes using 32 threads per point). An advantage of this approach is that the number of threads per point can be selected to reduce intra-warp thread divergence. For example, if $t = 32$ threads per point are used, then a full warp will compute the distance between a given point and the candidate points. There should be low divergence because each thread in the warp executes similar execution pathways. Recently, Gallet and Gowanlock [43] showed that on the similarity self-join, computing the distances between a query point and candidate points using more than one thread improves warp execution efficiency. However, as discussed above, the primary motivation for using $t > 1$ threads is that the *KNN*-join computes batches of query points which may be much smaller than the number of points in the dataset. Consequently, we need to use more threads per query point to saturate GPU resources.

Two drawbacks of using $t > 1$ include: (*i*) using too many threads per point may increase overhead; and (*ii*) query points in lower density regions may not need a large number of threads, and such threads will have minimal work to compute. There is a trade off between assigning too few or too many threads per point. We assume that the number of threads selected to compute the distance calculations for each point should evenly divide the size of a warp (32 threads). This eliminates the possibility of the threads assigned to a point spanning multiple warps and increasing divergence.

5. Experimental Evaluation

5.1. Datasets

We focus on low-dimensional *KNN*-joins due to their utility in many applications. Additionally, related work has consistently shown that GPU-accelerated *KNN* searches outperform CPU approaches at high dimensionality [14, 44], due to the increased cost of distance calculations. Thus, the GPU may be unsuitable to low-dimensional *KNN*-joins, and we target this low-dimensionality scenario.

We employ two classes of synthetic datasets with different workload characteristics. The Unif- class of datasets contains uniformly distributed data points. The Expo- class of datasets contains exponentially distributed data points with $\lambda = 40$. Datasets are generated in 2, 4, and 6 dimensions for both classes, and contain $|D| = 10^7$ points. Since HybridKNN-Join splits the low and high density regions between the CPU and GPU, respectively, the Unif- datasets represent the case where there is very low variation in density across the data space, whereas the Expo- datasets represent the case where there is a large data density gradient. Comparing the performance between these two classes of datasets allows us to examine how performance may vary as a function of data distribution and workload assignment between the CPU and GPU.

We also employ two 2-D real-world datasets: Gaia which contains $|D| = 2.5 \times 10^7$ positions of astronomical objects from the *Gaia* catalog [45], and Osm which contains $|D| = 2.5 \times 10^7$ positions from Open Street Map data [46]. Since the Open Street Map data contains GPS positions, we removed duplicate point coordinates, otherwise the *KNN* of many points may consist of a single GPS trajectory with identical (or nearly identical) point coordinates.

5.2. Experimental Methodology

All HybridKNN-Join CPU code is written in C/C++, compiled using the GNU compiler (v. 5.4.0) with the O3 flag. The GPU code is written in CUDA v. 9. We use OpenMPI v. 3.1.1 for parallelizing host-side tasks (discussed in Section 4.2.1). The work queue performs minimal work; however, we parallelize it using two OpenMP threads for assigning queries to Hybrid-CPU and Hybrid-GPU, as we need to wait on Hybrid-GPU without blocking Hybrid-CPU from obtaining new work.

Table 4: Default parameter values used in the experimental evaluation.

Parameter	Value
n_{large}	$0.4\lvert D\rvert$
n_{small}	$0.005\lvert D\rvert$
n_{Cwin}	$0.4\lvert D\rvert$
n_{CPU}	$0.005\lvert D\rvert$
t	8

Our platform consists of an NVIDIA GP100 GPU with 16 GiB of global memory, and has 2× Intel E5-2620 v4 2.1 GHz CPUs, with 16 total physical cores. The Hybrid-GPU kernel uses 256 threads per block. In the experiments, we exclude the time needed to load the dataset or construct the Hybrid-CPU indexes or the initial Hybrid-GPU index (see Section 5.3). The response time of performing the *KNN* search on the CPU and GPU is measured after the indexes have been constructed by Hybrid-CPU. All other components of the algorithm are included in the response time (e.g., finding the search distance for Hybrid-GPU, and ordering the workload for the work queue). All time measurements are averaged over 3 trials.

Table 4 outlines default parameter values in the experimental evaluation. Note that the initial monolithic batch size (n_{large}) and the window size of reserved CPU queries (n_{Cwin}) are both 40% of $|D|$. Increasing n_{large} beyond 40% is unlikely to greatly improve performance as the larger the value of n_{large}, the more queries that fail to find $\geq K$ neighbors. n_{small} and n_{CPU} are 0.5% of $|D|$, which is selected to minimize load imbalance, while not assigning too few queries per batch, which can increase the overhead of accessing the work queue.

5.3. Implementations

The implementations we use to carry out our performance evaluation are described below.

- CPU-Only– We compare to a multi-core CPU ANN [5] implementation that obtains the exact neighbors, as described in Section 4.2.1. We compare HybridKNN-Join to CPU-Only to demonstrate the performance gains yielded by the GPU. CPU-Only uses the CPU component of the hybrid algorithm, Hybrid-CPU, as executed with 16 processes that perform *KNN* searches, and 1 process for the work queue. There is no communication between ANN process ranks, as they find the *KNN* independently and write results directly to shared memory. Recall that we needed to parallelize ANN using MPI. We have each process rank independently construct its own kd-tree which is queried in batches obtained from the work queue. Since ANN does not perform parallel index construction and we cannot share the index between processes, we exclude this index construction time.

- HybridKNN-Join– Our hybrid approach uses: Hybrid-CPU with 15 processes, and Hybrid-GPU and the work queue each use 1 process. Hybrid-CPU uses one less process than CPU-Only. Due to the exclusion of the index construction time for CPU-Only (described above), we exclude the Hybrid-CPU index construction time, and the initial index constructed by Hybrid-GPU. However, we include all index construction time in the measurements when ϵ must be dynamically expanded by Hybrid-GPU.

- GPU-Only– We compare to a GPU-only implementation that uses Hybrid-GPU (the GPU component of HybridKNN-Join). Hybrid-GPU and the work queue each use 1 process. We note that because this implementation may fail to find at least K neighbors by design, depending on the data distribution, it may take significant time to find the *KNN* of all points, as those points in sparse regions were intended to be found by the Hybrid-CPU component of HybridKNN-Join. Thus, we show this implementation for comparison purposes, but note that it is not designed for standalone *KNN* searches. We configure GPU-Only to use only monolithic batches, i.e., $n_{large} = 1$ and $n_{Cwin} = 0$ for all batches computed by the algorithm. We ensure that each query batch size is equal to the number of points that have not yet found their respective *KNN*.

- CPU-Only-RR– We evaluate the potential negative performance impact of the work queue. We compare CPU-Only (described above) to another CPU implementation that has the work queue removed. Instead of using the work queue, we simply assign points in a round robin fashion to each process rank. Each $p_i \in D$ is assigned to rank r when $[(i-1) \bmod (P)] + 1 = r$, where P is the number of processes (MPI ranks), and $r = 1, 2, \ldots, P$. The implementation with the work queue removed uses $P = 16$ processes; therefore, the number of ranks computing the *KNN* is equivalent to the CPU-Only implementation. We elect to use a round robin distribution of points to process ranks to achieve good load balancing.

- BufferKDTree– We compare to the GPU buffer kd-tree algorithm by Gieseke et al. [28] discussed in Section 2. In all experiments where we compare to HybridKNN-Join we set the tree depth parameter to 12, as it achieves good performance across several datasets (this will be demonstrated in Section 5.4.3). The algorithm is designed for the same low/moderate dimensional searches that we examine in this paper. To maintain consistency with the methodology in Gieseke et al. [28], when reporting the response time, we only include the time to compute the query/test phase which finds the nearest neighbors. BufferKDTree allows for searching up to $K = 99$ neighbors; in our evaluation, the maximum value of K tested on this algorithm is $K = 96$. The BufferKDTree code is publicly available [47].

- KDTree– We compare to the multi-core CPU implementation in Gieseke et al. [28]. The algorithm is configured to use 16 threads (the number of physical cores on our platform), and similarly to BufferKDTree, we only report the query response time. The code is publicly available [47].

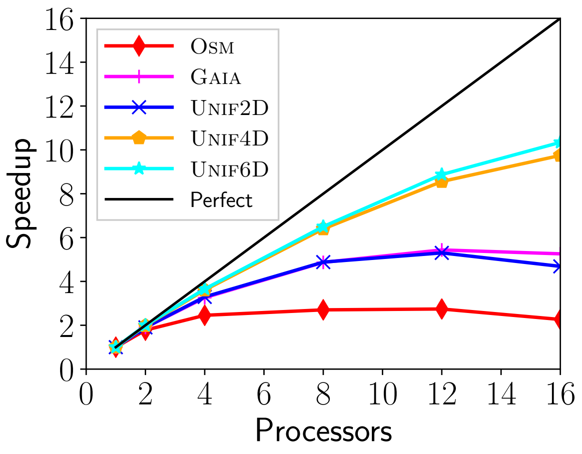


Figure 7: Speedup of the CPU-Only reference implementation on the Osm, Gaia, Unif2D, Unif4D, and Unif6D datasets where $K = 32$.

5.4. Results

5.4.1. Scalability of the CPU-Only Implementation

The CPU-Only reference implementation is a parallelized version of ANN [5] and is used for the CPU component of the hybrid algorithm, Hybrid-CPU (Section 4.2.1). CPU-Only, uses 16 process ranks to independently find the *KNN* of batches of query points. Figure 7 plots the scalability of CPU-Only across several datasets where $K = 32$. We find that scalability improves with data dimensionality. For example, Gaia is a 2-D dataset and achieves a speedup of 5.26× with 16 processes, whereas the 6-D Unif6D dataset achieves a speedup of 10.34× with 16 processes. The cost of the Euclidean distance calculation scales with dimensionality. Therefore, finding the *KNN* on lower dimensional datasets is memory-bound (the algorithm spends most of the time performing tree traversals), which transitions to becoming more compute-bound as the dimensionality increases. On Unif2D, the speedup slightly decreases from 12 to 16 processes. This is indicative of memory bandwidth saturation, where adding more processors does not improve performance (the same trend is observed on Gaia and Osm). However, on Unif4D and Unif6D, the greatest speedup is achieved when 16 processes are used, which indicates that memory bandwidth may not be saturated on those datasets when 16 processes are used. The poor scalability of the 2-D Osm dataset is surprising given that the other 2-D datasets (Gaia and Unif2D) achieve larger speedups.

In summary, CPU-Only (and Hybrid-CPU) achieves good scalability on the higher dimensional datasets, but the speedup is limited on the low dimensional datasets.

5.4.2. Scalability of the KDTree Implementation

The multi-threaded CPU KDTree implementation provides another baseline for comparison. In contrast, CPU-Only is parallelized using MPI. We execute the same experiment in Section 5.4.1 using KDTree. Figure 8 plots the speedup vs. the number of threads. We find very similar scalability using KDTree as we find for CPU-Only in Section 5.4.1. Since 16 threads achieves the best performance, we configure KDTree

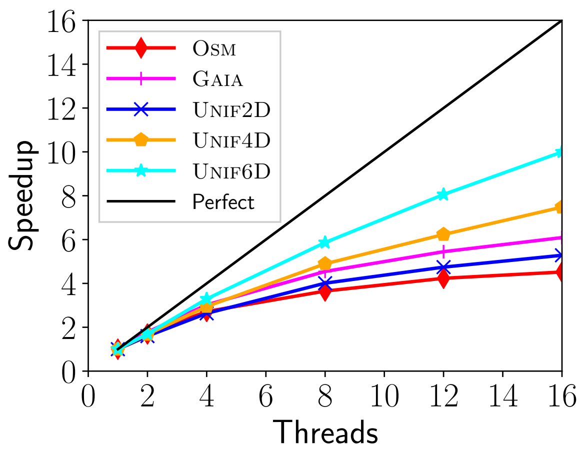


Figure 8: Speedup of the KDTree reference implementation on the Osm, Gaia, Unif2D, Unif4D, and Unif6D datasets where $K = 32$.

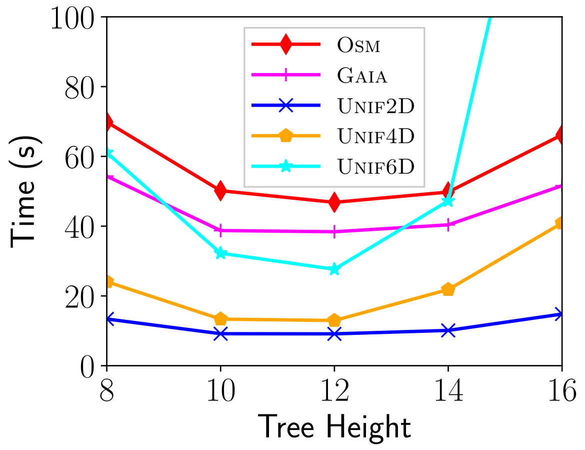


Figure 9: Impact of the BufferKDTree tree height parameter on performance. The response time vs. tree height is plotted for the Osm, Gaia, Unif2D, Unif4D, and Unif6D datasets where $K = 16$. The best performance is achieved across all datasets when the tree height is 12.

to use 16 threads when we compare it to the other implementations.

5.4.3. *Impact of the* BufferKDTree *Height Parameter*

The BufferKDTree implementation uses a height parameter that achieves a trade-off between examining too many leaves and pruning overhead caused by tree traversals. To ensure that we configure BufferKDTree with a height parameter that achieves good performance, we execute BufferKDTree on several datasets across different values of the height parameter. From Figure 9, we find that a height value of 12 achieves the best performance across all datasets, and this performance behavior is consistent with the experimental evaluation in Gieseke et al. [28]. In all future experiments, we use this tree height.

5.4.4. *Potential Impact of Work Queue Overhead on Performance*

Utilizing work queues can add overhead to parallel algorithms (e.g., accesses must be serialized to avoid race conditions). We compare two parallel CPU-only algorithms: CPU-Only that uses the work queue, and CPU-Only-RR that has the work queue removed, where each point is assigned to a rank in a round robin fashion. By comparing the implementation with the work queue to the same implementation without the work queue, we are able to assess whether using the work queue adds considerable overhead.

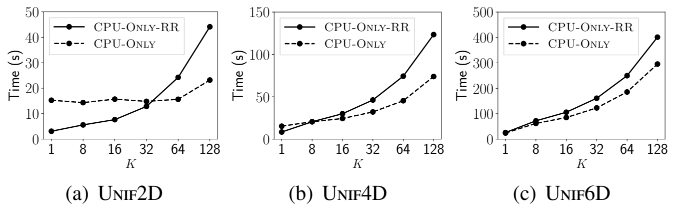


Figure 10: Response time vs. K comparing CPU-Only and CPU-Only-RR on the uniformly distributed datasets.

Figure 10 plots the response time vs. K on the uniformly distributed datasets. We observe from the figure that the work queue improves performance over the round robin assignment of points on the larger workloads. For instance, when $K \geq 8$ on the Unif4D and Unif6D datasets, the implementation with the work queue (CPU-Only) outperforms the round robin distribution of points to ranks (CPU-Only-RR). However, on the Unif2D dataset, CPU-Only-RR outperforms CPU-Only when $K \leq 32$. Because the workload is relatively low in 2-D, the initial work queue overheads have a non-negligible performance impact (we elaborate on these overheads in Section 5.4.5), and accessing the work queue has a non-negligible impact, where several processes contend for a new batch of work to compute.

Overall, we observe that the work queue generally has a positive impact on performance. In fact, the performance gain from the work queue over the round robin implementation is substantial in the cases described above (e.g., Figure 10(b) and (c)). This is an unintended benefit of the work queue. We attribute this to two factors, described as follows:

1. The work queue helps reduce load imbalance between process ranks, as queries are retrieved from the queue on-demand.

2. The work queue first sorts the points by total work based on the number of points found in each cell. This means that spatially co-located points found in the same cell are likely to be assigned to the same process rank by the work queue. Because the points are spatially co-located, it is likely that the tree traversals are benefiting from good locality when performing *KNN* searches. In contrast, the round robin assignment of points to each process rank cannot benefit as much from the spatial co-location of points.

We test the notion that cache effects are improving the performance of CPU-Only over CPU-Only-RR described above, by simply using the Performance analysis tools for Linux (perf) to count the total number of cache references and misses using the `cache-references` and `cache-misses` flags. We note that perf yields a coarse grained measurement of the cache references and misses in the program, and is not limited to *KNN*

Table 5: Comparison of the total number of cache references and percentage of cache misses on the CPU-Only and CPU-Only-RR implementations on the Unif4D dataset. The lowest values of the percentage of cache misses as a function of K are highlighted in bold face.

K	CPU-Only Cache References	CPU-Only Cache Misses (%)	CPU-Only-RR Cache References	CPU-Only-RR Cache Misses (%)
1	29208826489	**52.18**	26790397165	57.80
8	32379870022	**54.42**	31912590674	64.11
16	34398747794	**54.72**	35841307333	67.81
32	37541677665	**53.76**	41829138108	72.19
64	44754599329	**48.18**	51312595208	76.67
128	62648741911	**38.39**	68117485096	81.73

Table 6: HybridKNN-Join work queue construction overheads on the Unif- and Expo- classes of datasets where $K = 32$. The total response time and the fraction of the total response time is shown for computing ϵ and ordering the work queue workload. The default parameter values in Table 4 are used.

Dataset	Total response time (s)	Fraction computing ϵ	Fraction ordering work queue workload
Unif2D	9.82	0.063	0.068
Unif4D	19.54	0.037	0.035
Unif6D	74.50	0.012	0.011
Expo2D	10.82	0.057	0.063
Expo4D	22.97	0.038	0.035
Expo6D	77.21	0.013	0.011

searches. However, since the vast majority of the computation is performing *KNN* searches (tree traversals and filtering) in these algorithms, perf is a reasonable tool for understanding whether good locality is responsible for the performance difference between the CPU-Only and CPU-Only-RR implementations.

Table 5 shows the number of cache references and percentage of cache misses when executing the CPU-Only and CPU-Only-RR implementations on the Unif4D dataset. We find that the percentage of cache misses increases with K in the CPU-Only-RR implementation, whereas this percentage decreases with K in the CPU-Only implementation. We expect that the impact on locality of CPU-Only favors larger values of K. The larger the value of K, the larger the search space, which is likely to increase the probability of two searches for differing points to be able to reuse data in cache. In contrast, the CPU-Only-RR implementation processes queries in a round robin fashion and cannot exploit locality between consecutive searches. Therefore, as K increases, the total percentage of cache misses increases.

By comparing CPU-Only and CPU-Only-RR, we find that the work queue does not add significant overhead to the CPU-Only algorithm (with the exception of the Unif2D dataset when $K \leq 32$), and therefore, does not hinder HybridKNN-Join. Furthermore, we find that assigning batches of query points to processes through the work queue has a positive effect on locality. The performance gain due to positive cache effects outweighs the performance loss due to work queue access overhead.

5.4.5. Overheads of Work Queue Construction and Selection of the Search Distance

Table 6 quantifies the fraction of the total response time of major overheads when $K = 32$ on the Unif- and Expo- classes of datasets when executing HybridKNN-Join with the default parameter values in Table 4. From Table 6, we observe that the fraction of time computing ϵ (Section 4.4) and ordering the work queue workload (Section 4.5) is dependent on the data dimensionality rather than the data distribution, as the uniform and exponentially distributed datasets require similar fractions of the total response time for each of these overheads.

The percentage of the total response time to compute ϵ ranges from 6.3% (Unif2D) to 1.2% (Unif6D), and these percentages for ordering the work queue workload range from 6.8% (Unif2D) to 1.1% (Unif6D). Thus, the overheads associated with constructing the work queue are mostly amortized on larger workloads; however, they are non-negligible on the smaller workloads. Despite these work queue construction overheads, from Section 5.4.4 we find that the work queue is generally advantageous due to positive cache effects.

5.4.6. GPU Kernel Task Granularity

Hybrid-GPU uses a number of threads (t) to process each point (Section 4.7). Since the size of each batch assigned from the work queue to Hybrid-GPU, $|Q_{GPU}|$, can vary (e.g., due to decreasing monolithic batch sizes), it is important that sufficient threads are executed, such that GPU resources remain saturated, which is achieved through oversubscription. Additionally, using more than one thread per query point allows for less divergence in each warp, as fewer query points (with differing execution pathways) are assigned to a single warp.

Table 7 shows the total response time of HybridKNN-Join for a selection of datasets, and values of K (8, 32, 128) and t threads (1, 8, 16, and 32) assigned to perform the distance calculations for each query point. We select values of t such that the threads divide evenly into the size of a warp (32 threads). This ensures that a query point does not span two warps. From Table 7 we observe that many of the datasets have consistent response times when varying t. For instance, on Gaia with $K = 8$, the response times range from 19.90 s to 21.00 s. In contrast, on the Unif6D dataset with $K = 32$, the response times range from 73.99 s to 90.01 s. Using $t = 1$ may lead to inter-warp load imbalance (waiting for the last warp to finish execution) and increases divergence in the kernel. At the other extreme, using a single warp ($t = 32$) to compute each query point may underutilize resources. For instance, if there are fewer than 32 candidates within an adjacent cell of a query point, then several threads will not have any work to execute. Across all datasets and values of t, we find that $t = 8$ yields the best performance. Additionally, in many cases where $t = 8$ does not yield the best performance, we find that it achieves similar performance to the best performing value of t. Consequently, we configure HybridKNN-Join with a default value of $t = 8$.

5.4.7. Work Queue Performance: GPU Monolithic Batch Size

We examine performance as a function of the selection of the monolithic batch size. The selection of the monolithic batch

Table 7: Response time (s) of HybridKNN-Join when varying t for $t = 1, 8, 16, 32$ and $K = 8, 32, 128$ on Gaia, Osm, Unif4D, and Unif6D datasets. Excepting t, the default parameter values in Table 4 are used. The lowest response time is shown in bold face for each K on each dataset.

Dataset	K	$t = 1$	$t = 8$	$t = 16$	$t = 32$
Gaia	8	19.96	**19.90**	19.98	21.00
Osm	8	36.31	32.45	**31.97**	32.34
Unif4D	8	**11.75**	12.23	12.50	14.00
Unif6D	8	**41.33**	42.62	44.85	52.15
Gaia	32	25.85	25.34	**23.78**	24.40
Osm	32	36.73	32.13	31.92	**31.19**
Unif4D	32	20.35	**19.99**	20.15	21.74
Unif6D	32	90.01	**73.99**	78.15	84.43
Gaia	128	41.45	**39.42**	40.32	40.38
Osm	128	35.60	32.51	**31.57**	32.24
Unif4D	128	41.65	**37.27**	38.27	40.23
Unif6D	128	184.45	**168.01**	172.23	172.67

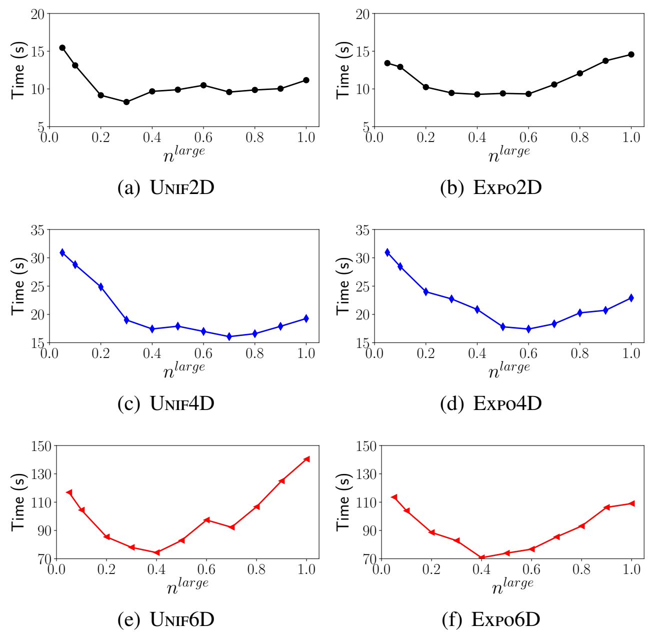


Figure 11: HybridKNN-Join response time vs. monolithic batch size n_{large} where $K = 32$ and $n_{Cwin} = 0$. Excepting n_{large} and n_{Cwin}, the default parameter values in Table 4 are used.

size, n_{large}, has several performance implications. A large n_{large} will decrease the fraction of queries that Hybrid-GPU is able to successfully compute in the first batch round, as ϵ is selected to find on average K neighbors per $p_i \in D$ (Section 4.4). A small value of n_{large} will decrease GPU throughput, where GPU resources may not be sufficiently saturated. We examine the performance impact of the monolithic batch size when we do not reserve any queries for the CPU ($n_{Cwin} = 0$). This allows us to observe how performance changes across values of n_{large} in the range [0.05, 1.0]. Note that when $n_{large} = 1.0$, the GPU can be assigned the entire dataset to process during its first batch round and this will have the effect of starving the CPU of work.

Figure 11(a), (c), and (e) plot the response time vs. n_{large} on the Unif2D, Unif4D, and Unif6D datasets, respectively, and the exponential datasets are shown in Figure 11(b), (d), and (f). For clarity, the uniform and exponential datasets of the same dimensionality are positioned adjacent to each other. From Figure 11(a), (c), and (e), we find that n_{large} should not be too small, otherwise GPU resources will not be fully utilized, which is shown by the initial decrease in response time (e.g., comparing $n_{large} = 0.05$ and 0.2 in Figure 11(a)). However, on Unif2D, we observe that too large a value of n_{large} will decrease performance. This effect is more pronounced on the Unif6D dataset, where we find that the best value of $n_{large} = 0.4$, which shows that Hybrid-GPU should be configured with large monolithic batches when computing larger workloads. However, the value should not be too large, otherwise, the CPU will be starved of work, which explains the performance degradation when $n_{large} > 0.4$ on Unif6D. Similar results are shown on the exponential datasets in Figure 11(b), (d), and (f), so we omit a similar discussion.

Comparing the results in Figure 11, n_{large} can be selected in a large range to achieve good performance (the response times are similar between $n_{large} \approx 0.3-0.6$). Hence, we select $n_{large} = 0.4$ in Table 4 to achieve a compromise between small (low dimensionality) and large (high dimensionality) workloads.

5.4.8. *Work Queue Performance: Load Balancing*

We determine whether the configuration of the work queue is able to mitigate load imbalance between CPU and GPU components. Load imbalance is defined as: $|T_{CPU} - T_{GPU}|/T$, where T_{CPU} is the time that the last executing Hybrid-CPU rank finishes computation, T_{GPU} is the time that the Hybrid-GPU rank finishes computing its last batch, and T is the total response time.

Figure 12 shows the load imbalance for $K = 8, 32, 128$ on the Unif- and Expo- classes of datasets. Across all datasets and values of K, we find that the load imbalance is $< 10\%$ and generally increases with K. As K increases, the total work computed by each batch increases. This enhances the chances of either the CPU or GPU components of HybridKNN-Join finishing their work at disparate times. We find that HybridKNN-Join achieves reasonably good load balancing despite several confounding issues. To further mitigate load imbalance, the batch sizes could be adjusted based on the value of K. For instance, when K is large, smaller batches can be employed. We do not consider this optimization, as it would require an additional parameter that scales as a function of K, and it is unlikely to lead to substantial performance gains, as the load imbalance is already within an acceptable range ($< 10\%$).

5.4.9. *Quantifying the Number of Failed Queries*

As described in Section 4.2.2 and 4.5, Hybrid-GPU will fail to find some of the query points in each batch. This design decision was made to avoid dynamically increasing the search radius and number of searched cells inside the kernel.[3] In this section, we examine the number of failed queries on all datasets for selected values of K.

[3]In an early implementation, we dynamically increased the search radius inside the kernel, but found in preliminary experiments that it led to poor performance due to low warp execution efficiency.

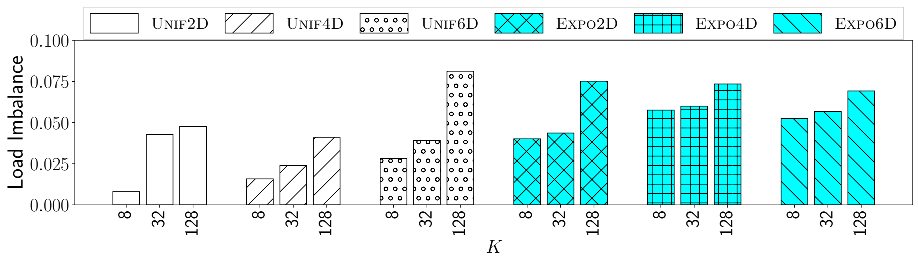


Figure 12: Load imbalance on the uniformly and exponentially distributed datasets for $K = 8, 32, 128$. Default parameter values in Table 4 are used.

Table 8: The fraction of failed Hybrid-GPU queries when executing HybridKNN-Join, calculated as the total number of failed queries divided by the total number of attempted queries. The default parameter values in Table 4 are used.

Dataset	$K = 8$	$K = 32$	$K = 96$
Gaia	0.073	0.164	0.240
Osm	0.000	0.000	0.000
Unif2D	0.163	0.135	0.162
Unif4D	0.217	0.198	0.273
Unif6D	0.270	0.236	0.211
Expo2D	0.087	0.058	0.041
Expo4D	0.257	0.280	0.296
Expo6D	0.342	0.274	0.250

Table 8 shows the fraction of failed queries computed as the total number of failed queries divided by the total number of attempted queries. The fraction of failed queries ranges from 0–0.342. There are two key observations regarding failed queries described as follows:

1. Clearly, failed queries are wasted work computed by the GPU. From Table 8, up to 34.2% of queries are wasted.
2. If ϵ is sufficiently large you can always find the *KNN* of each query point, and eliminate the wasted work in (1) above. However, this is at the expense of wasted work in another context: many of the query points will find significantly more neighbors than needed, and this increases the overhead of the refinement step, thereby increasing the number of distance calculations.

Since the fraction of failed queries is not too large (e.g., a fraction > 0.5 is likely too large), it indicates that we are reaching a trade-off between not finding too few or too many points. In other words, if the fraction of failed queries was too small, we would expect that we are wasting work refining too many candidate points for the average query point. From Table 8 we find that on the Osm dataset, we (nearly) always find at least K neighbors for each query point. This indicates on this dataset that a lower value of ϵ would likely lead to better performance, as the algorithm is refining many more candidate points than needed. We will elaborate on this observation in Section 5.4.10.

All implementations that use indexes for *KNN* searches such as the CPU-Only and KDTree reference implementations will suffer from the refinement overhead described in (2) above. This is because an index cannot guarantee that only K neighbors will be found for a given search. Therefore, this is not a shortcoming of Hybrid-GPU, rather it is a consequence of using an index.

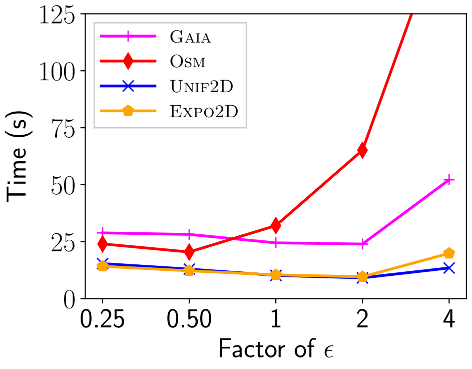


Figure 13: Examining the sensitivity of HybridKNN-Join to the initial selection of ϵ on the 2-D datasets, where $K = 32$. We execute HybridKNN-Join with a factor 0.25, 0.5, 2, and 4 of the initial search radius, ϵ, where a value of 1 corresponds to the default ϵ value. Default parameter values in Table 4 are used.

5.4.10. Performance Impact of the Initial Selection of the Search Distance

As described in Section 4.4, we select an initial search distance, ϵ, that is expected to find at least K neighbors on average for each query point. In this section, we examine how sensitive the performance of HybridKNN-Join is to variations in this parameter. Figure 13 plots the response time of HybridKNN-Join vs. the factor of the initial value of ϵ. In the plot the value of 1 corresponds to the initial/default value of ϵ. We vary the value of ϵ to be 0.25×–4× the initial search distance. From Figure 13, we find that at a factor 2ϵ, the Gaia, Unif2D, and Expo2D datasets achieve slightly better performance than at the default value of ϵ; however, the performance of Osm degrades significantly at 2ϵ. Additionally, we find that on Osm, the best performance is achieved at 0.5ϵ. Overall, our method for selecting an initial value of ϵ achieves good performance across all datasets in Figure 13. This supports the semi-analytic geometric justification for the selection of ϵ outlined in Section 4.4.

In Section 5.4.9 we observed that the Osm dataset has nearly 0 failed queries (Table 8). From Figure 13, we observe that at a lower value of ϵ, such as 0.5ϵ, Osm achieves better performance than at the default ϵ value. This demonstrates that having nearly no failed queries is an indicator that too many neighbors are being found, which yields additional distance calculations and as-

Table 9: The fraction of the total response time spent re-indexing Hybrid-GPU when ϵ is expanded for various values of K. Values in parentheses show the number of times Hybrid-GPU needed to re-index. The default parameter values in Table 4 are used.

Dataset	$K = 8$	$K = 32$	$K = 96$
Gaia	0.039 (1)	0.026 (1)	0.029 (1)
Osm	0.000 (0)	0.000 (0)	0.000 (0)
Unif2D	0.030 (1)	0.024 (1)	0.021 (1)
Unif4D	0.028 (1)	0.018 (1)	0.017 (1)
Unif6D	0.015 (1)	0.009 (1)	0.005 (1)
Expo2D	0.061 (1)	0.027 (1)	0.020 (1)
Expo4D	0.083 (3)	0.034 (2)	0.021 (2)
Expo6D	0.036 (2)	0.017 (2)	0.011 (2)

sociated overhead. Also, it demonstrates that having a moderate fraction of query failures is beneficial for performance.

5.4.11. *Temporal Evolution: Re-indexing and Batch Sizes*

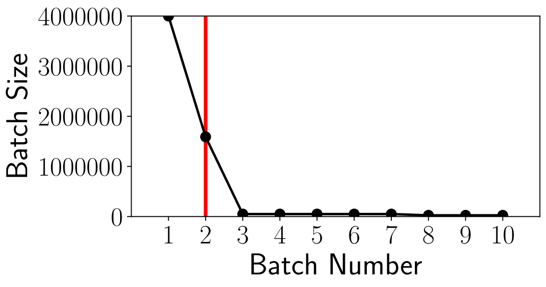
At each batch, the Hybrid-GPU component of HybridKNN-Join counts the number of failed queries. If the number of failed queries on the previous batch exceeds 25% of the number of query points in the batch, the algorithm will re-index with a larger value of ϵ (Sections 4.2.2 and 4.5). During this time no queries are computed on the GPU. Table 9 shows the fraction of the total response time spent re-indexing across all datasets for various values of K.[4] Values in parentheses indicate the number of times Hybrid-GPU re-indexed. From Table 9 we find that Hybrid-GPU will sometimes not re-index at all (Osm for all values of K), and will re-index up to 3 times (Expo4D, $K = 8$). The median number of times the algorithm will re-index is one. Furthermore, re-indexing accounts for 0%–8.3% of the total response time. In the majority of cases, less than 3% of the total response time is spent re-indexing. This demonstrates that re-indexing overhead does not significantly degrade the GPU's query throughput.

Using the same experiments shown in Table 9, Figure 14 plots the total number of queries assigned to Hybrid-GPU at each batch on Unif4D and Expo4D using $K = 8$. The vertical red lines correspond to the batch number that triggered recomputing the index. The figure shows the temporal evolution of HybridKNN-Join indicating that large workloads are initially assigned to the GPU at batches 1 and 2, which decrease to ensure that low load imbalance occurs between CPU and GPU components of the algorithm.

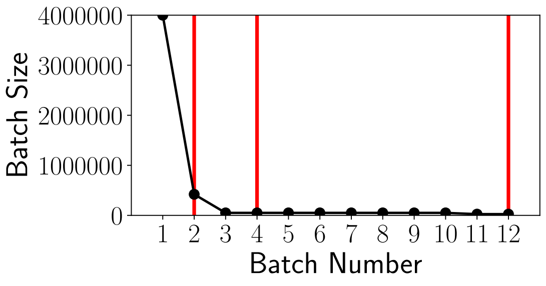
5.4.12. *Characterization of the* Hybrid-GPU *Search Distance Expansion*

Recall from Section 4.5 that Hybrid-GPU expands ϵ while computing batches of query points. We examine the performance of the Hybrid-GPU expansion of ϵ through an experiment using GPU-Only. As discussed in Section 5.3, the GPU-Only implementation is used for comparison purposes, but is not designed for standalone *KNN* searches. In this section, we

[4] A non-integral average number of times HybridKNN-Join needs to re-index is counterintuitive. Therefore, for clarity, we only used a single time trial in this experiment.



(a) Unif4D, $K = 8$



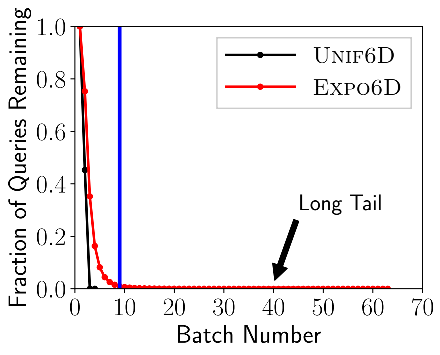
(b) Expo4D, $K = 8$

Figure 14: Hybrid-GPU batch size vs. batch number for (a) Unif4D and (b) Expo4D, where $K = 8$. The vertical red lines correspond to the batch number that triggered re-indexing the dataset.

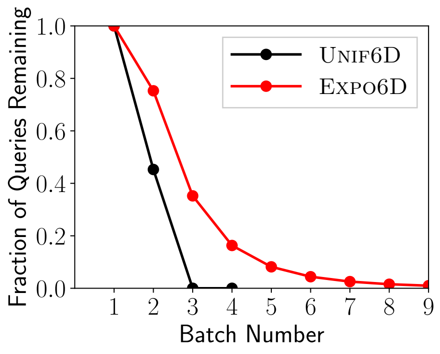
also elaborate on the reasoning for the unsuitability of GPU-Only (that uses Hybrid-GPU) as a standalone *KNN* search algorithm.

We execute GPU-Only on the Unif6D and Expo6D datasets. Figure 15(a) plots the fraction of queries remaining to be processed vs. the batch number. With each successive batch, the number of queries left to be computed decreases, but the number of successfully solved queries is dependent on the search distance, ϵ, and the data distribution. For example, when the batch number is 1 (the first batch), then 100% of the queries still need to be computed. From Figure 15(a) we observe that on the Unif6D dataset only 4 batches are required to find the *KNN* of all query points in the dataset. However, on the Expo6D dataset, 64 batches are needed to compute the *KNN* of all points in the dataset. However, on the Expo6D dataset, the *KNN* of 99% of the points in the dataset are found after only computing 9 batches (denoted by the vertical blue line). The remaining 1% of the data points require computing 64-9=55 additional batches. Figure 15(b) shows Figure 15(a) but constrained to the first 9 batches. From Figure 15(b), we find that if we exclude the remaining 1% of data points that require executing an additional 55 batches (the long tail in Figure 15(a)), then the fraction of queries remaining to be computed by Expo6D is reasonable. For comparison, when 4 batches have been computed, the *KNN* have been found for all points in the Unif6D dataset, and only 16.3% of the points are left to compute on the Expo6D dataset.

In general, we find that when the data are uniformly distributed, the selection of the initial value of ϵ is able to find the *KNN* of a large fraction of the data points with few expansions of ϵ. In contrast, when the data are exponentially dis-



(a)



(b)

Figure 15: The fraction of queries in D that remain to be computed vs. the batch number when executing GPU-ONLY on the UNIF6D and EXPO6D datasets. (a) The UNIF6D dataset is computed in 4 batches, whereas the EXPO6D dataset requires 64 batches to find the *KNN* of all points. The vertical line at 9 batches denotes where 99% of the points in D have been computed on the EXPO6D dataset. (b) The first 9 batches in (a).

tributed, the mean value of ϵ needed to find K neighbors on average (ϵ_{min}) is largely unsuitable for those data points in sparse regions. Thus, the GPU-ONLY algorithm needs to continually expand ϵ to find the points in these sparse regions, and this may require a prohibitive number of ϵ expansions that degrades performance.

To remedy this problem in a GPU-only implementation, a non-linear expansion of ϵ could be employed to avoid the long tail exhibited in Figure 15(a) (recall that ϵ is expanded by $\epsilon_{min}/2$ when HYBRID-GPU fails to find the *KNN* of at least 25% of the query points in the previous batch). As an alternative, a simple brute force search could be employed when a small fraction of points in the dataset have not found their *KNN* (e.g., similarly to EXPO6D in Figure 15, where 99% of the data points find their respective *KNN* in 9 batches, then only 1% of the data are left to find their respective *KNN*). In this case, a brute force search will not be prohibitive to performance. However, we reiterate that HYBRID-GPU expects those points in the low density regions to be found by HYBRID-CPU; therefore, we do not use these solutions, as their implementation would not be executed when using HYBRIDKNN-JOIN.

Figure 16(a)–(c) plots the response time vs. K on the UNIF- datasets, and the EXPO- datasets are shown in Figure 16(d)–(f). Despite the performance drawback of the GPU-ONLY implementation on the EXPO- datasets described above, we find that on the uniformly distributed datasets, the GPU-ONLY implementation is very efficient. In fact, we find that GPU-ONLY slightly outperforms HYBRIDKNN-JOIN across all values of K on the UNIF2D dataset. Because the dataset contains 2-D points (the smallest workload), the slight overheads and load imbalance between architectures are observable on this dataset. In contrast, on the UNIF6D dataset (a larger workload), we find that HYBRIDKNN-JOIN outperforms GPU-ONLY.

5.4.13. *Comparison of* CPU-ONLY *and* KDTREE

We compare the performance of the standalone multi-core CPU implementations, CPU-ONLY and KDTREE. Recall that CPU-ONLY is parallelized using MPI processes and KDTREE is parallelized using threads. From Figure 16, we observe that the on the synthetic datasets, the performance of CPU-ONLY degrades gracefully with increasing K. In contrast, the performance of KDTREE degrades much more significantly with K on some datasets, such as UNIF2D and EXPO2D.

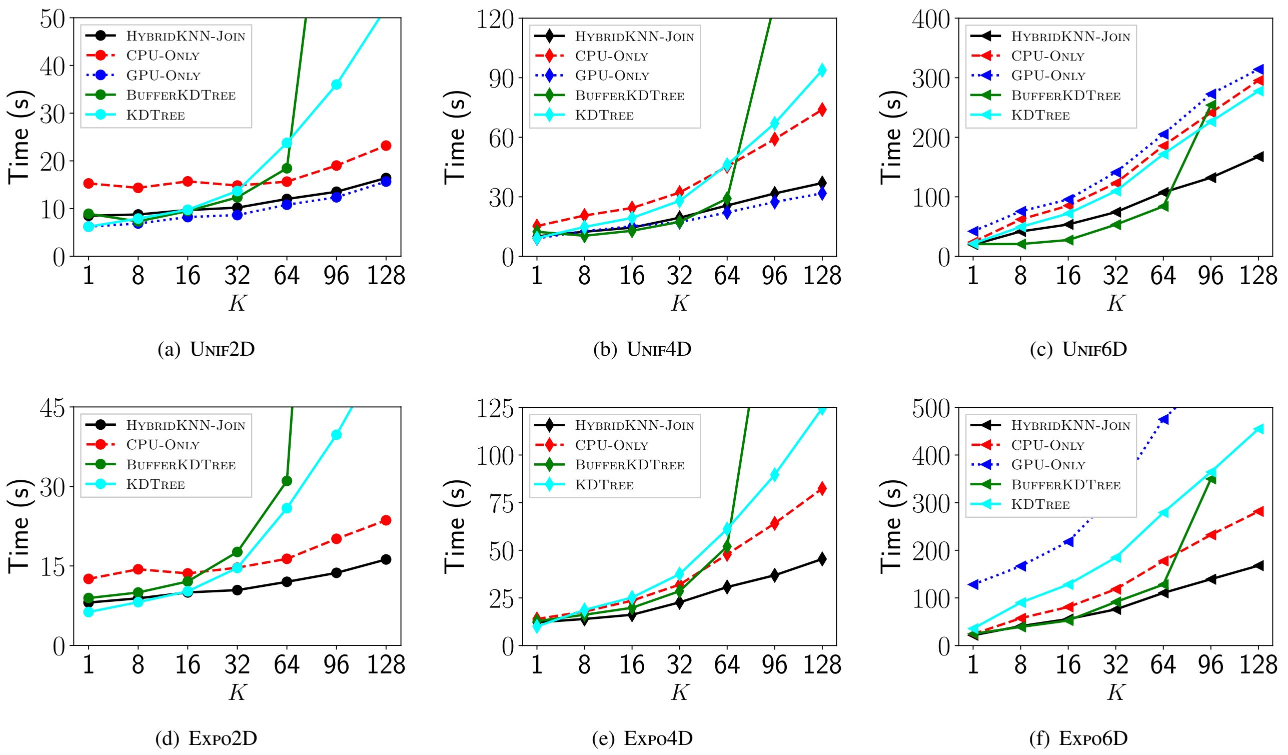
Figure 17 plots the response time vs. K on the real-world datasets. These datasets are 2-D with $|D| = 2.5 \times 10^7$ points. We note on this experiment that KDTREE failed to execute on $K \geq 96$ on both GAIA and OSM. We find that KDTREE outperforms CPU-ONLY when $K \lesssim 32$, whereas CPU-ONLY outperforms KDTREE when $K > 32$. Based on the performance of KDTREE in Figure 16, we expect the response time of KDTREE to increase significantly at $K \geq 96$.

To more clearly observe the performance differences between CPU-ONLY and KDTREE, Figure 18 plots the speedup of CPU-ONLY over KDTREE on all datasets. We find that CPU-ONLY is much more efficient on the larger values of K on the UNIF2D, EXPO2D, and EXPO4D datasets. Additionally, since these two multi-core CPU reference implementations have varying performance characteristics, they are able to provide a more comprehensive comparison to HYBRIDKNN-JOIN (see Section 5.4.14).

In general, we find that CPU-ONLY is mostly competitive with or outperforms KDTREE (Figure 18). This indicates that the use of MPI to parallelize HYBRID-CPU and CPU-ONLY is not prohibitive to the performance of the algorithms. If MPI were prohibitive, we would expect that KDTREE would achieve better performance relative to CPU-ONLY.

5.4.14. *Comparison of* HYBRIDKNN-JOIN *to* CPU-ONLY *and* KDTREE

We compare the performance of HYBRIDKNN-JOIN to the two multi-core CPU implementations, CPU-ONLY and KDTREE. From Figure 16, across all datasets and values of K, HYBRIDKNN-JOIN outperforms CPU-ONLY. This demonstrates that the use of the CPU and GPU in the hybrid algorithm does not degrade performance for any value of K. Comparing HYBRIDKNN-JOIN to KDTREE, we find that there are a few cases where KDTREE outperforms HYBRIDKNN-JOIN on lower values of K. For example, on UNIF2D and EXPO2D, KDTREE slightly outperforms HYBRIDKNN-JOIN when $K \leq 8$. Overall,



(a) Unif2D (b) Unif4D (c) Unif6D

(d) Expo2D (e) Expo4D (f) Expo6D

Figure 16: Response time vs. K comparing HybridKNN-Join, CPU-Only, GPU-Only, BufferKDTree, and KDTree on (a)–(c) uniformly, and (d)–(f) exponentially distributed datasets. HybridKNN-Join is configured with the default parameter values in Table 4. In (d) and (e), we do not plot GPU-Only, as the total response times are much larger than the other implementations due to the explanation given in Section 5.4.12.

HybridKNN-Join yields a reasonable performance gains over KDTree.

On the Gaia dataset (Figure 17(a)), HybridKNN-Join outperforms CPU-Only across all values of K. On the Osm dataset (Figure 17(b)), HybridKNN-Join and CPU-Only have nearly equal performance from $1 < K \leq 32$; however, when $K > 32$, HybridKNN-Join outperforms CPU-Only. We find that KDTree outperforms HybridKNN-Join on Gaia and Osm when $K \leq 16$.

It is interesting to note that on Gaia and Osm, KDTree outperforms CPU-Only on low values of K, whereas CPU-Only performs better on the higher values of K (we observed similar behavior on some of the synthetic datasets in Figure 16). This suggests that HybridKNN-Join could be equipped with different algorithms, such as using KDTree instead of the ANN-based CPU-Only implementation for lower values of K. While this is beyond the scope of this work, algorithm selection as a function of input parameters is an interesting research direction for hybrid CPU/GPU algorithms.

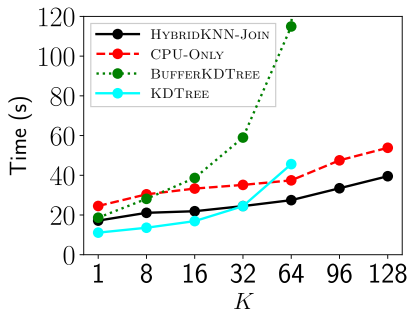
Figure 19(a) plots the speedup of HybridKNN-Join over CPU-Only for all datasets in Figures 16 and 17. The plot demonstrates that the performance advantage of HybridKNN-Join is greater when K is large or when the dimensionality increases (excepting the 2-D datasets). Figure 19(b) shows that as K increases, the fraction of queries solved by Hybrid-GPU also increases on most datasets. Note that while the fraction of D computed by Hybrid-GPU is generally <50%, Hybrid-GPU is computing the queries with the greatest amount of work in the denser data regions of the Gaia, Osm, and Expo- datasets (by definition, the Unif- datasets have constant density across the data space).

5.4.15. *Comparison of* HybridKNN-Join *to* BufferKDTree

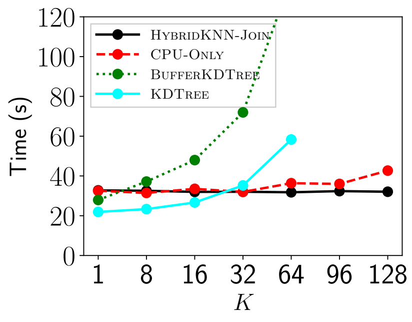
Figure 20 plots the speedup of HybridKNN-Join over the GPU algorithm BufferKDTree vs. K on all datasets shown in Figures 16 and 17, corresponding to the synthetic and real-world datasets, respectively.

We find that on most synthetic datasets, HybridKNN-Join achieves a speedup over BufferKDTree. However, BufferKDTree outperforms HybridKNN-Join on some smaller values of K (e.g., on the Unif2D and Unif4D datasets), and outperforms HybridKNN-Join on Unif6D when $K \leq 64$. Overall, we find that HybridKNN-Join significantly outperforms BufferKDTree on the larger values of K.

On the real-world datasets, Gaia and Osm, the performance gains over BufferKDTree are more pronounced than on the synthetic datasets. On the Gaia dataset, the speedup ranges from 1.08–16.59×, and on the Osm dataset the speedup ranges from 0.85–19.24×. Therefore, there is only one case at $K = 1$ on Osm where HybridKNN-Join achieves a slowdown relative to BufferKDTree.



(a) Gaia (2-D)



(b) Osm (2-D)

Figure 17: Response time vs. K on 2-D real-world datasets. Default parameter values in Table 4 are used.

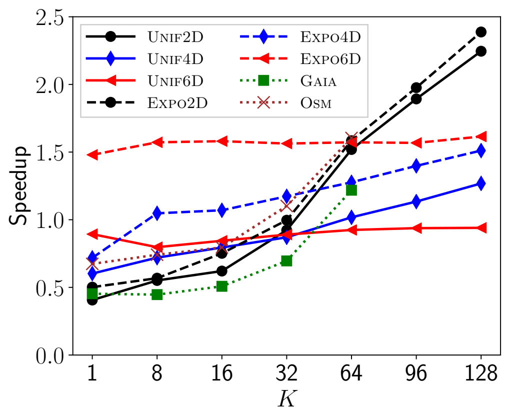
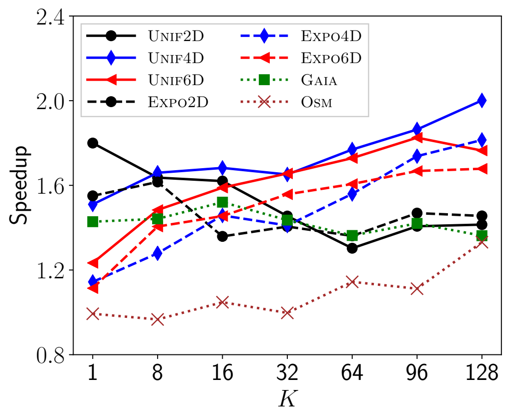


Figure 18: Speedup of CPU-Only over KDTree vs. K on all datasets shown in Figures 16 and 17.

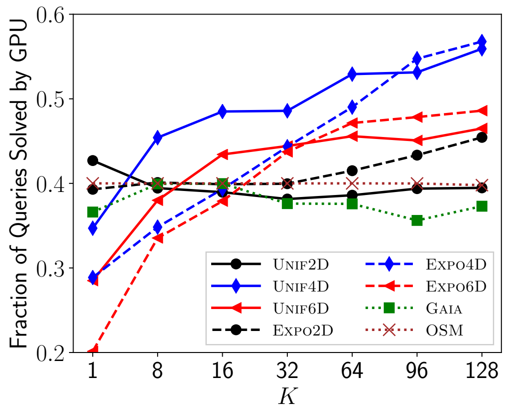
6. Discussion & Conclusions

Many of the GPU *KNN* works address high-dimensionality [14, 15, 16, 17]. Here, we advance a hybrid approach for low-dimensionality that exploits the relative strengths of the CPU and GPU architectures. GPU *KNN* algorithms are less likely to achieve significant performance gains in low dimensionality due to highly efficient CPU algorithms, such as ANN [5].

We consider the throughput-oriented GPU vs. the low-latency CPU. Our strategy assigns large batches to the GPU to maintain high throughput, while the CPU ranks are assigned smaller chunks of work. We largely mitigate load imbalance and starvation by reducing the batch size assigned to the GPU depending on the number of completed queries, and reserving queries for the CPU. The work queue allows new advances in GPU- and CPU-only algorithms to be substituted into the



(a)



(b)

Figure 19: (a) Speedup of HybridKNN-Join over CPU-Only vs. K on all datasets in Figures 16 and 17. With the exception of the 2-D datasets, the speedup roughly increases with dimensionality and K. (b) Fraction of query points solved by Hybrid-GPU. Excepting 2-D datasets, the GPU computes a larger fraction of D with increasing K.

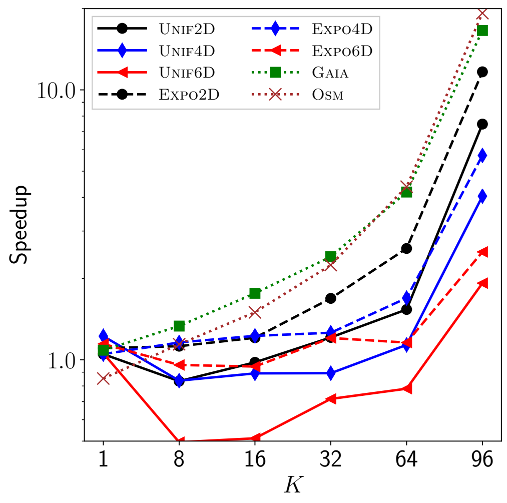


Figure 20: Speedup of HybridKNN-Join over BufferKDTree vs. K on all datasets shown in Figures 16 and 17.

framework to further improve performance. More broadly, the work queue could be used as a general technique to address other CPU/GPU algorithms with data-dependent performance characteristics.

HybridKNN-Join yields reasonable performance gains over the reference implementations. We find that the speedup over the parallel CPU approach (CPU-Only) is $< 2\times$; however, from Figure 19(a), we clearly observe that the speedup of HybridKNN-Join is expected to be greater at higher values of K (and potentially dimensionality) than the scenarios examined in this paper. Similarly, we find that HybridKNN-Join outperforms the GPU reference implementation, BufferKDTree, on most scenarios (Figure 20).

An overall observation from this exercise is that hybrid algorithms are difficult to design. Since the performance of each CPU and GPU algorithm largely varies due to input parameters (K) and data properties, it is challenging to design an algorithm that will outperform or achieve comparable performance to all other CPU-only or GPU-only reference implementations.

We found that there are some experimental scenarios where HybridKNN-Join yields a slowdown compared to some of the reference implementations (e.g., the KDTree implementation on low values of K). Since all *KNN* algorithms have particular performance niches, hybrid algorithms could be developed to include algorithm selection as a function of several parameters, such as K, data dimensionality, and data distribution. This would allow hybrid algorithms to achieve better performance over a wider range of scenarios. We leave this research direction for future work.

A recent trend in computer architecture is the use of GPUs in clusters. For example, each compute node in the Summit supercomputer at Oak Ridge National Laboratory contains six Nvidia Volta GPUs [48]. The work queue proposed in this paper could be used to distribute work to multiple GPUs within a single node. Furthermore, the work queue is a good design for workloads that vary based on data distribution, and could be applied to other spatial search algorithms used for data analysis, such as similarity searches [35], and DBSCAN clustering [49, 50]. Based on our experiments, the work queue is able to achieve good load balancing between the CPU and GPU, and therefore, we would expect to achieve good load balancing between multiple GPUs. The work queue would only need to be reconfigured to incorporate several GPU consumers. Interestingly, on fat-nodes like those in Summit, the use of the CPU in hybrid algorithms would become less important, since the computational throughput of several GPUs would be much higher than the CPUs in the system. Another interesting design is partitioning the input dataset across the global memory of multiple GPUs to enable larger datasets to be processed. New interconnects such as NVLink [51] enable direct GPU-to-GPU communication, thus obviating slower main-memory accesses orchestrated by the host.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1849559.

[1] Y. Zhang, H. Ma, N. Peng, Y. Zhao, X.-b. Wu, Estimating Photometric Redshifts of Quasars via the k-nearest Neighbor Approach Based on Large Survey Databases, The Astronomical Journal 146.

[2] J. A. Hartigan, M. A. Wong, Algorithm AS 136: A k-means clustering algorithm, Journal of the Royal Statistical Society. Series C (Applied Statistics) 28 (1) (1979) 100–108.

[3] G. Karypis, E.-H. Han, V. Kumar, Chameleon: Hierarchical clustering using dynamic modeling, Computer 32 (1999) 68–75.

[4] N. Roussopoulos, S. Kelley, F. Vincent, Nearest neighbor queries, in: Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, 1995, pp. 71–79.

[5] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, A. Y. Wu, An optimal algorithm for approximate nearest neighbor searching in fixed dimensions, Journal of the ACM 45 (6) (1998) 891–923.

[6] M. Muja, D. G. Lowe, Scalable nearest neighbor algorithms for high dimensional data, IEEE Transactions on Pattern Analysis & Machine Intelligence (11) (2014) 2227–2240.

[7] M. Nam, J. Kim, B. Nam, Parallel Tree Traversal for Nearest Neighbor Query on the GPU, in: 45th Intl. Conf. on Parallel Processing, 2016, pp. 113–122.

[8] J. Kim, B. Kim, S. Savarese, Comparing image classification methods: K-nearest-neighbor and support-vector-machines, in: Proceedings of the 6th WSEAS international conference on Computer Engineering and Applications, and Proceedings of the 2012 American conference on Applied Mathematics, Vol. 1001, 2012, pp. 48109–2122.

[9] J. L. Bentley, Multidimensional binary search trees used for associative searching, CACM 18 (9) (1975) 509–517.

[10] A. Guttman, R-trees: a dynamic index structure for spatial searching, in: Proc. of ACM Intl. Conf. on Management of Data, 1984, pp. 47–57.

[11] R. E. Bellman, Adaptive control processes: a guided tour, Princeton university press, 1961.

[12] Nvidia Volta, `http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`, accessed: 31-01-2019.

[13] T. D. Han, T. S. Abdelrahman, Reducing branch divergence in GPU programs, in: Proc. of the 4th Workshop on General Purpose Processing on Graphics Processing Units, 2011, pp. 3:1–3:8.

[14] V. Garcia and É. Debreuve and F. Nielsen and M. Barlaud, K-nearest neighbor search: Fast gpu-based implementations and application to high-dimensional feature matching, in: 2010 IEEE Intl. Conf. on Image Processing, 2010, pp. 3757–3760.

[15] A. S. Arefin, C. Riveros, R. Berretta, P. Moscato, Gpu-fs-knn: A software tool for fast and scalable knn computation using GPUs, PLOS ONE 7 (8) (2012) e44000.

[16] L. Jian, C. Wang, Y. Liu, S. Liang, W. Yi, Y. Shi, Parallel data mining techniques on graphics processing unit with compute unified device architecture (cuda), The Journal of Supercomputing 64 (2013) 942–967.

[17] I. Komarov, A. Dashti, R. M. D'Souza, Fast k-nng construction with gpu-based quick multi-select, PLOS ONE 9 (5) (2014) e92409.

[18] M. Gowanlock, B. Karsin, Accelerating the similarity self-join using the GPU, Journal of Parallel and Distributed Computing 133 (2019) 107–123.

[19] S. Mittal, J. S. Vetter, A Survey of CPU-GPU Heterogeneous Computing Techniques, ACM Comput. Surv. 47 (4) (2015) 69:1–69:35.

[20] L. Li, X. Li, G. Tan, M. Chen, P. Zhang, Experience of Parallelizing cryo-EM 3D Reconstruction on a CPU-GPU Heterogeneous System, in: Proc. of the 20th Intl. Symposium on High Performance Distributed Computing, ACM, 2011, pp. 195–204.

[21] A. Deshpande, I. Misra, P. J. Narayanan, Hybrid implementation of error diffusion dithering, in: 18th Intl. Conf. on High Performance Computing, 2011, pp. 1–10.

[22] M. Gowanlock, KNN-Joins Using a Hybrid Approach: Exploiting CPU/GPU Workload Characteristics, in: Proceedings of the 12th Workshop on General Purpose Processing Using GPUs, ACM, 2019, pp. 33–42.

[23] C. Xia, H. Lu, B. C. Ooi, J. Hu, Gorder: an efficient method for KNN join processing, in: Proc. of the Intl. Conf. on Very Large Data Bases, 2004, pp. 756–767.

[24] A. Andoni, P. Indyk, Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions, in: IEEE Symposium on Foundations of Computer Science, 2006, pp. 459–468.

[25] C. Yu, B. Cui, S. Wang, J. Su, Efficient index-based knn join processing

for high-dimensional data, Information and Software Technology 49 (4) (2007) 332–344.
[26] B. Yao, F. Li, P. Kumar, K nearest neighbor queries and knn-joins in large relational databases (almost) for free, in: IEEE 26th Intl. Conf. on Data Engineering, 2010, pp. 4–15.
[27] P. Leite, J. M. Teixeira, T. Farias, B. Reis, V. Teichrieb, J. Kelner, Nearest neighbor searches on the gpu, International Journal of Parallel Programming 40 (3) (2012) 313–330.
[28] F. Gieseke, J. Heinermann, C. Oancea, C. Igel, Buffer kd trees: processing massive nearest neighbor queries on GPUs, in: International Conference on Machine Learning, 2014, pp. 172–180.
[29] M. M. A. Patwary, N. R. Satish, N. Sundaram, J. Liu, P. Sadowski, E. Racah, S. Byna, C. Tull, W. Bhimji, P. Dubey, et al., Panda: Extreme scale parallel k-nearest neighbor on distributed architectures, in: Proc. of the 2016 Intl. Parallel and Distributed Processing Symposium, 2016, pp. 494–503.
[30] J. Ryoo, M. Arunachalam, R. Khanna, M. T. Kandemir, Efficient k nearest neighbor algorithm implementations for throughput-oriented architectures, in: 2018 19th international symposium on quality electronic design (isqed), IEEE, 2018, pp. 144–150.
[31] R. A. Finkel, J. L. Bentley, Quad trees a data structure for retrieval on composite keys, Acta informatica 4 (1) (1974) 1–9.
[32] S. K. Prasad, M. McDermott, X. He, S. Puri, Gpu-based parallel r-tree construction and querying, in: 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, IEEE, 2015, pp. 618–627.
[33] J. Kim, W.-K. Jeong, B. Nam, Exploiting massive parallelism for indexing multi-dimensional datasets on the gpu, IEEE Transactions on Parallel and Distributed Systems 26 (8) (2015) 2258–2271.
[34] J. Kim, B. Nam, Co-processing heterogeneous parallel index for multi-dimensional datasets, Journal of Parallel and Distributed Computing 113 (2018) 195–203.
[35] D. V. Kalashnikov, Super-EGO: fast multi-dimensional similarity join, The VLDB Journal 22 (4) (2013) 561–585.
[36] M. D. Lieberman, J. Sankaranarayanan, H. Samet, A fast similarity join algorithm using graphics processing units, in: IEEE 24th Intl. Conf. on Data Engineering, 2008, pp. 1111–1120.
[37] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, CACM 51 (1) (2008) 107–113.
[38] W. Lu, Y. Shen, S. Chen, B. C. Ooi, Efficient Processing of K Nearest Neighbor Joins Using MapReduce, Proc. VLDB Endow. 5 (10) (2012) 1016–1027.
[39] C. Zhang, F. Li, J. Jestes, Efficient parallel kNN joins for large data in MapReduce, in: Proc. of the 15th Intl. Conf. on Extending Database Technology, 2012, pp. 38–49.
[40] M. Gowanlock, C. M. Rude, D. M. Blair, J. D. Li, V. Pankratius, Clustering Throughput Optimization on the GPU, in: Proc. of the IEEE Intl. Parallel and Distributed Processing Symposium, 2017, pp. 832–841.
[41] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, J. McDonald, Parallel programming in OpenMP, Morgan Kaufmann, 2001.
[42] D. Merrill, M. Garland, A. Grimshaw, Scalable GPU Graph Traversal, in: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, 2012, pp. 117–128.
[43] B. Gallet, M. Gowanlock, Load Imbalance Mitigation Optimizations for GPU-Accelerated Similarity Joins, in: Proc. of the 2018 IEEE Intl. Parallel and Distributed Processing Symposium Workshops, to appear, 2019.
[44] K. Kato, T. Hosino, Multi-GPU algorithm for k-nearest neighbor problem, Concurrency and Computation: Practice and Experience 24 (1) (2012) 45–53.
[45] Gaia Collaboration, Brown, A. G. A., Vallenari, A., Prusti, T., de Bruijne, J. H. J., Babusiaux, C., et al., Gaia data release 2 - summary of the contents and survey properties, Astronomy & Astrophysics 616 (2018) A1.
[46] OpenStreetMap, `https://blog.openstreetmap.org/2012/04/01/bulk-gps-point-data/`, accessed 31-01-2019.
[47] F. Gieseke, Bufferkdtree, accessed 6-February-2020 (2016). URL `https://bufferkdtree.readthedocs.io/`
[48] Oak Ridge National Laboratory, Summit Supercomputer, `https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/summit-faqs/`, accessed: 12-02-2020.
[49] M. Ester, H. Kriegel, J. Sander, X. Xu, A density-based algorithm for discovering clusters in large spatial databases with noise, in: Proc. of the 2nd KDD, 1996, pp. 226–231.
[50] C. Böhm, B. Braunmüller, M. Breunig, H.-P. Kriegel, High performance clustering based on the similarity join, in: Proc. of the Ninth Intl. Conf. on Information and Knowledge Management, 2000, pp. 298–305.
[51] D. Foley, J. Danskin, Ultra-Performance Pascal GPU and NVLink Interconnect, IEEE Micro 37 (2) (2017) 7–17. `doi:10.1109/MM.2017.37`.

Michael Gowanlock is an assistant professor in the School of Informatics, Computing, & Cyber Systems at Northern Arizona University. He received a Ph.D. in computer science at the University of Hawai‘i at Mānoa. He was a postdoctoral associate at MIT Haystack Observatory. His research interests include parallel data-intensive computing, and astronomy.