KNN-Joins Using a Hybrid Approach: Exploiting CPU/GPU Workload Characteristics

Michael Gowanlock
Northern Arizona University
School of Informatics, Computing, and Cyber Systems
Flagstaff, AZ, U.S.A.
michael.gowanlock@nau.edu

ABSTRACT

K Nearest Neighbor (*KNN*) joins are used in many 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. However, *KNN* searches are computationally expensive, and many GPU *KNN* algorithms focus on the high-dimensional case that plainly gives a performance advantage to the GPU rather than the CPU. Consequently, in this work, we focus on a hybrid CPU/GPU approach for the low-dimensional *KNN*-join problem. In particular, 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, is shown to effectively augment a state-of-the-art multi-core CPU algorithm. We propose optimizations that (*i*) maximize GPU query throughput by assigning the GPU larger batches of work than the CPU; (*ii*) increase workload granularity to optimize GPU resource utilization; and, (*iii*) limit load imbalance between CPU and GPU architectures. Furthermore, the work queue utilized in our approach shows promise for the general purpose division of work for other hybrid CPU/GPU algorithms.

KEYWORDS

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

ACM Reference Format:
Michael Gowanlock. 2019. KNN-Joins Using a Hybrid Approach: Exploiting CPU/GPU Workload Characteristics. In *General Purpose Processing Using GPU (GPGPU-12), April 13, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3300053.3319417

1 INTRODUCTION

The performance of data-intensive computations such as K nearest neighbor (*KNN*) searches are often limited by the memory bottleneck. The high aggregate memory bandwidth of graphics processing units (GPUs) (e.g., 900 GiB/s on the Nvidia Volta [1]) results in roughly an order-of-magnitude increase in memory bandwidth over the CPU. Therefore, GPUs are well-suited to data-intensive workloads. However, it is well-known that data transfers to and from the GPU are a bottleneck, which can decrease the performance advantages afforded by the GPU. Additionally, many data-dependent workloads, such as the *KNN*-join studied in this work, can have irregular execution patterns that make the GPU potentially unsuitable for the algorithm due to thread divergence and serialization that degrades performance [15]. Thus, it is not clear that the GPU will lead to performance gains over multi-core CPU approaches.

We study 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 (e.g., within an astronomy catalog, find the closest five objects of all objects within a feature space [35]). *KNN* searches are used in many applications, such as the k-means [16], and Chameleon [19] clustering algorithms. Consequently, *KNN* searches have been well studied [5, 27, 31], including GPU [28] algorithms. However, many GPU approaches only minimally involve the host, which underutilize CPU resources.

Many GPU-accelerated *KNN* algorithms focus on optimizing brute force approaches, which highlight performance in high dimensional feature spaces and often compute a distance matrix [4, 11, 17, 23]. 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. The algorithms are often intractable for large datasets because the brute force approach has a quadratic complexity. An index data structure can be used to reduce the quadratic complexity of brute force searches on the CPU or GPU, by reducing the number of point comparisons [5, 27, 28, 31].

Given the context above, we outline the major goals of this work. **Addressing Low-Dimensionality on the GPU:** The abovementioned brute force *KNN* searches in high-dimensional spaces are clearly well-suited to the GPU compared to the CPU due to the large number of distance calculations that need to be computed. 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.
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 [12]. 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.

GPGPU-12, April 13, 2019, Providence, RI, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6255-9/19/04…$15.00
https://doi.org/10.1145/3300053.3319417

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. We make the following contributions:

- 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 range queries. 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 queries 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 a method to mitigate load imbalance between architectures.

Paper organization: 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.

2 BACKGROUND

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: $dist(p_a, p_b) = \sqrt{\sum_{j=1}^{n}(p_a(x_j) - p_b(x_j))^2}$, where x_j denotes the point's coordinate in dimension j. We assume an in-memory scenario where the entire database fits within the global memory of a GPU, and the entire result set (the K nearest neighbors of each point) fits within main memory on the host; however, the entire result set may exceed GPU global memory capacity. 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, $R \ltimes_{KNN} S$.

2.1 Related Work

We present an overview of several categories of related work below.
Hybrid Algorithms – Using both the CPU and GPU is needed to achieve peak performance in heterogeneous systems (see [26] for a survey of hybrid algorithms). Several works split the work between the CPU and GPU at runtime. For instance, Li et al. [24] parallelize Cryo-EM 3D reconstruction, and assign tasks to the CPU or GPU depending on the workload. Deshpande et al. [8] 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 these works, HybridKNN-Join dynamically schedules the query points onto the architecture most suitable for the workload.
***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 [3, 5, 27, 28, 31–34]. We describe a sample of the literature below.

An R-tree is used to find the *KNN* in [31] that uses a branch-and-bound recursive algorithm that first gets an estimate of the *KNN* and then performs backtracking on subtrees to find the exact neighbors. Backtracking in tree-based solutions [28, 31] is used to ensure that at least K nearest neighbors are found.

While the E^2LSH [3] algorithm performs range queries, and is not designed for *KNN*, it can be used to find nearest neighbors by constructing several data structures corresponding to increasing search radii, and querying them in ascending order by distance until K neighbors are found [2]. We employ a similar increasing search radius strategy for the GPU component of our work.

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) [27], which achieves good performance using a parallel search over a randomized kd-forest. While FLANN outperforms ANN for one scenario in [27], the comparison was between a parallel (FLANN) and sequential algorithm (ANN). Since ANN is considered state-of-the-art, we parallelize and incorporate it into HybridKNN-Join.
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, such as index-trees (e.g., kd-trees [6], quad-trees [9], and R-trees [14]), and they are constructed as a function of the data distribution. In contrast, there are data-oblivious methods, such as statically partitioned grids [12].

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 they contain many branch instructions, which can reduce the parallel efficiency of the GPU due to the SIMT architecture. A GPU R-tree [21] was 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 [22]. This shows that the GPU should be leveraged through the use of regularized instructions, yielding low thread divergence. Consequently, we use a non-hierarchical indexing technique with low thread divergence for our GPU join operation.
Range Queries and Joins – Our hybrid approach uses range queries 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 [18] 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 [3], and LSS [25] algorithms. A GPU self-join was presented in [12] that was shown to be efficient on low-dimensional data. We leverage some of the optimizations in the GPU self-join work [12] as they are effective for executing range queries that can be used to solve *KNN* searches on the GPU.

3 RECAP OF PREVIOUS SELF-JOIN WORK

HybridKNN-Join leverages the distance similarity self-join work of Gowanlock & Karsin [12], which was evaluated on up to $n = 6$ dimensions. The authors used an efficient indexing scheme and batching scheme from [13], and proposed a technique to reduce the number of duplicate computations. 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 [12], 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 [12, 13] for more detail) with cells of length ϵ. The index 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 point is carried out by performing distance calculations between points in each adjacent cell of the point (and the cell containing the point). The number of adjacent cells is 3^n (e.g., in 2-D there are 9 total grid cells). 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.

3.2 Batching Scheme

We give a brief overview of the GPU batching scheme in [12]. 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 steams (a minimum of $n_b = 3$), which overlaps the execution of the kernel and data transfers to exploit bidirectional PCIe bandwidth, and concurrent host and GPU tasks. We use $b_s = 10^8$ for each stream.

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 range query 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, there is no guarantee that all (or any) of the 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.

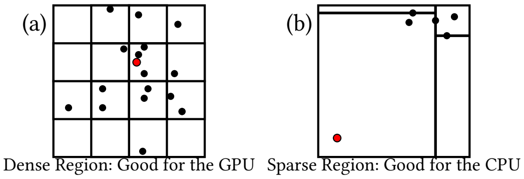

(a)
(b)
Dense Region: Good for the GPU
Sparse Region: Good for the CPU

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).

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; thus, there are a significant number of distance calculations and filtering needed 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 [6] 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

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

to the *KNN* problem, and we execute the algorithm such that we obtain the exact nearest neighbors. As noted in other work [30], 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 communication between 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 [31], backtracking is used to ensure that K neighbors are found for each point searched. Likewise, the E^{2}LSH [3] 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.

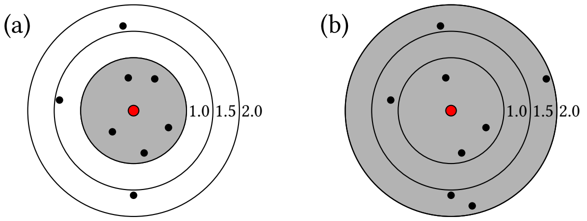


Figure 2: A *KNN* search around two query points (larger red points at the centers) where $K = 5$. Shaded region denotes the range required to find $K = 5$ points. (a) $K = 5$ neighbors are found with $\epsilon = 1$. (b) $K = 5$ neighbors are found when the search distance is expanded to $\epsilon =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

To transform range queries with a distance ϵ into a *KNN* search that considers the throughput-oriented nature of the 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 4.4 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 Hybrid-GPU: Selecting the Search Distance

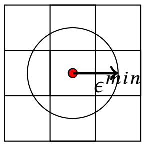
The input parameter to a *KNN* search is K; but Hybrid-GPU needs an ϵ-distance which is expected to find at least K neighbors for each point. Analytically deriving ϵ is feasible when the input data distribution is known. However, real-world datasets have data distributions that make an analytical approach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

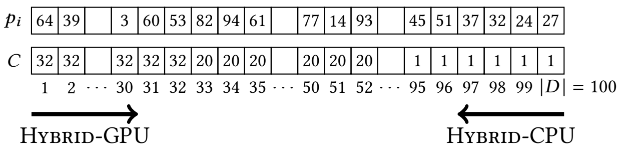
Figure 3: A 2-D example of the search radius ϵ^{min}, which probabilistically contains K neighbors per $p_i \in D$.



4.4 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 4.3) 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 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 the 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 that are to be searched to compute the total number of distance calculations; 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.



p_i	64	39		3	60	53	82	94	61		77	14	93		45	51	37	32	24	27
C	32	32		32	32	32	20	20	20		20	20	20		1	1	1	1	1	1
	1	2	···	30	31	32	33	34	35	···	50	51	52	···	95	96	97	98	99	$\|D\| = 100$

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.

We outline several work queue performance considerations.

- *Load Imbalance* – Performance degrades while one architecture 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.

This is similar to the classical trade-off between load imbalance and work queue overhead (e.g., static vs. dynamic scheduling of for loops in OpenMP [7]). 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, Hybrid-GPU may not find the *KNN* for each point assigned to it (Section 4.3). Each $p_i \in D$ that fails to find *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 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:

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

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 using threads 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 solved and some will have failed to find the *KNN* (shown as partially complete). 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}.

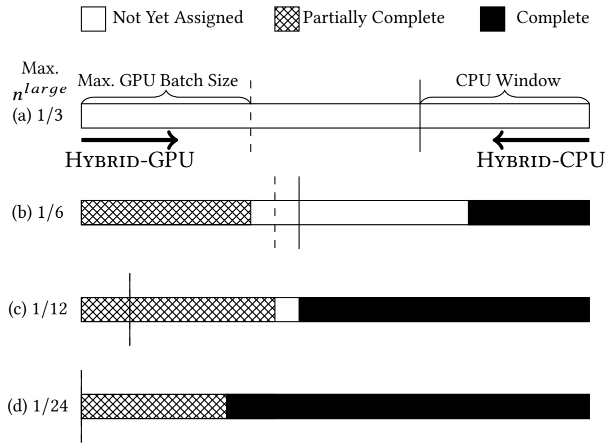


Figure 5: Assigning monolithic batches of queries from the work queue to Hybrid-GPU and queries to Hybrid-CPU (small Hybrid-GPU batch rounds not shown). (a) Initial work queue with $n^{large} = n^{Cwin} = 1/3$. (b) After processing a monolithic batch, some queries have been computed by Hybrid-GPU and 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. 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 [7]).

4.5 GPU: Optimizing Task Granularity

In the self-join work that we leverage [12], 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.

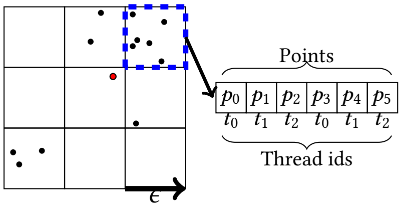

Points
p_0 p_1 p_2 p_3 p_4 p_5
t_0 t_1 t_2 t_0 t_1 t_2
Thread ids
ϵ

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

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 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. Drawbacks include: (*i*) too many threads per point 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. 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.

4.6 Algorithm Overview

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 21, 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). Next, the value of ϵ^{min} is selected (Section 4.3) on line 6, and then ϵ is set using this value on line 7 (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 8. Then, the algorithm gets a number of queries from the work queue rank on line 9 and stores them in Q^{GPU}. A while loop is entered on line 10 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 11 (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 12). At each iteration, the GPUJoinKernel is executed (line 13), which computes the result set for a single batch. On line 14 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 15, after all of the batches have been computed, those query points executed on the GPU that have $< K$ neighbors are assigned to the Q^{Fail} set, and these queries are added back to the work queue on line 16.

On lines 17–19, 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 4.4). And finally, on line 20, the rank retrieves work for the next batch from the work queue.

Regarding Hybrid-CPU, on line 22, queries are obtained from the work queue rank. Assuming there are queries to process, a while loop is entered on line 23, which computes the result of the *KNN* search for its batch of queries on line 24. The next batch of work is obtained from the work queue rank on line 25, and the loop continues until their are no additional queries to compute.

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:         $\epsilon^{min}$ ← selectEpsilon($D$)
 7:         $\epsilon \leftarrow \epsilon^{min}$
 8:         $G$ ← constructIndex($D$, $\epsilon$)
 9:         $Q^{GPU}$ ← getWork()
10:         while $|Q^{GPU}| > 0$ do
11:             $n_b$ ← computeNumGPUBatches($b_s$, $Q^{GPU}$, $\epsilon$)
12:             for $i \in 1, 2, \ldots, n_b$ do
13:                 kernResult[i] ← GPUJoinKernel($D$, $Q^{GPU}$, $G$, $\epsilon$, $i$)
14:                 KNNresult ← KNNresult ∪ filterKeys(kernResult[i])
15:             $Q^{Fail}$ ← findFailedPnts(KNNresult, $Q^{GPU}$)
16:             addFailuresToWorkQueue($Q^{Fail}$)
17:             if $|Q^{Fail}|/|Q^{GPU}| > 0.25$ then
18:                 $\epsilon \leftarrow \epsilon + 0.5\epsilon^{min}$
19:                 $G$ ← constructIndex($D$, $\epsilon$)
20:             $Q^{GPU}$ ← getWork()
21:     else                                                          ▹ CPU Ranks
22:         $Q^{CPU}$ ← getWork()
23:         while $|Q^{CPU}| > 0$ do
24:             KNNresult ← KNNresult ∪ Hybrid-CPU ($Q^{CPU}$, myRank)
25:             $Q^{CPU}$ ← getWork()
26:     return

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

We describe the Hybrid-GPU join kernel, but refer the reader to [12] 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 4.5). In the GPU join kernel shown in Algorithm 1, the result set is initialized (line 28), and then the global thread id is computed (line 29). Next, the query point assigned to the thread is stored (line 30), and a loop iterates over all adjacent cells (lines 31–32). 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 33–34). 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 33 (see Figure 6).

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 [11, 20], 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. 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 [10], and Osm which contains $|D| = 2.5 \times 10^7$ positions from Open Street Map data [29] after point de-duplication.

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. Also, we use OpenMP for parallelizing index construction when re-indexing Hybrid-GPU.

Our platform consists of an NVIDIA GP100 GPU with 16 GiB of global memory, and has 2× 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, ordering the workload for the work queue). Time measurements are averaged over 3 trials.

Table 1: Summary of notation, and parameters. Default values shown in parentheses where applicable.

	Descriptive Notation
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 dynamically expand.
Hybrid-CPU	Parallel CPU component of HybridKNN-Join.
Hybrid-GPU	GPU component of HybridKNN-Join.
HybridKNN-Join	The proposed CPU/GPU approach.
CPU-Only	Parallel CPU-only reference implementation.
	Parameter Notation
n^{large} (0.4)	Initial monolithic batch size for Hybrid-GPU as a fraction of $\|D\|$.
n^{small} (0.005)	Small batch size for Hybrid-GPU as a fraction of $\|D\|$.
n^{Cwin} (0.4)	Window size of reserved queries for Hybrid-CPU during monolithic batch rounds as a fraction of $\|D\|$.
n^{CPU} (0.005)	Batch size for Hybrid-CPU as a fraction of $\|D\|$.
t (8)	Number of threads assigned to each query point for Hybrid-GPU.

Table 1 summarizes notation, parameters, and their default values. 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 work queue overhead.

5.3 Implementations

We compare HybridKNN-Join to a CPU-only implementation, as described below. We do not use a GPU-only implementation because it would be designed differently (e.g., removal of the work queue, pipelining index construction, and other designs).

•**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. We execute CPU-Only 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. 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.

5.4 Results

5.4.1 Overheads. We quantify the percentage of the total response time of major overheads when $K = 32$ on the Unif- and Expo- classes of datasets. The time to find ϵ (Section 4.3) ranges from 4.9% (Unif2D) to 1.1% (Unif6D), and these percentages for ordering the work queue workload (Section 4.4) range from 5.5% (Unif2D) to 0.8% (Unif6D). Thus, these overheads are amortized on larger workloads; however, they are non-negligible on the smaller workloads.

5.4.2 GPU Kernel Task Granularity. Hybrid-GPU uses a number of threads (t) to process each point (Section 4.5). 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.

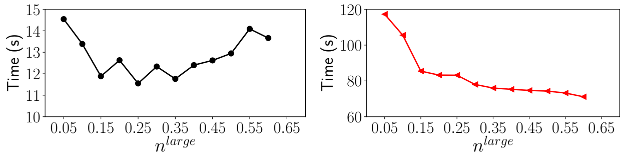
Table 2: Response time (s) when varying t for $t = 1, 8, 32$ and $K = 8, 32, 128$ on Gaia, Osm, Unif4D, and Unif6D datasets. Excepting t, the default parameter values in Table 1 are used. The lowest response time is shown in bold face for each K.

t	1	8	32
		$K = 8$	
Gaia	**25.258**	25.548	25.893
Osm	37.340	**31.812**	32.017
Unif4D	14.960	**14.101**	15.655
Unif6D	43.850	**43.779**	52.362
		$K = 32$	
Gaia	27.123	**23.669**	25.377
Osm	36.509	32.437	**31.707**
Unif4D	21.733	**20.942**	21.698
Unif6D	91.988	**75.275**	85.136
		$K = 128$	
Gaia	43.703	42.251	**41.193**
Osm	36.067	33.182	**30.844**
Unif4D	43.092	**39.300**	41.342
Unif6D	188.299	**167.552**	175.122

Table 2 shows the total response time for a selection of datasets, and values of K (8, 32, 128) and t threads (1, 8, and 32) assigned to perform the distance calculations for each query point. From Table 2 we observe that on $K = 8$ and $K = 32$, assigning t=8 leads to the best response time on three of four of the datasets. At $K = 128$ on Gaia and Osm, t=32 threads yields the best response time, whereas on Unif4D and Unif6D, this value is t=8. Because the real world Gaia and Osm datasets contain more points than Unif4D and Unif6D, we attribute this to the increased workload in the former two datasets that benefit from additional threads. The larger the workload (larger datasets and K), the greater the potential for load imbalance when searching for the neighbors of each point. Thus, more threads can decrease the time to compute individual query points, which reduces load imbalance.

5.4.3 Work Queue Performance Characteristics. We examine performance as a function of the selection of the monolithic batch size and load imbalance between Hybrid-CPU and Hybrid-GPU. The selection of the monolithic batch 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 4.3). A small value of n^{large} will decrease GPU throughput, where GPU resources may not be saturated.

Figure 7 (a) and (b) plot the response time vs. n^{large} for Unif2D and Unif6D, respectively. We set $n^{Cwin} = 0.4$, thus during the monolithic batch phase, 40% of $p_i \in D$ are reserved for Hybrid-CPU, thus $n^{large} \leq 0.6$. Without setting a window size, performance will degrade with large n^{large}, as Hybrid-GPU will starve Hybrid-CPU of queries to process. In Figure 7 (a)–(b), 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 0.15 in Figure 7 (a)). However, on Unif2D, we see that too large a value of n^{large} will decrease



(a) Unif2D

(b) Unif6D

Figure 7: Response time vs. monolithic batch size n^{large} where $K = 32$. Excepting n^{large}, the default parameter values in Table 1 are used.

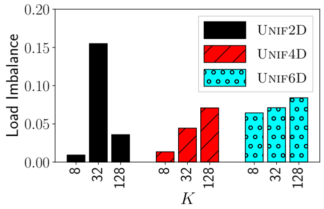


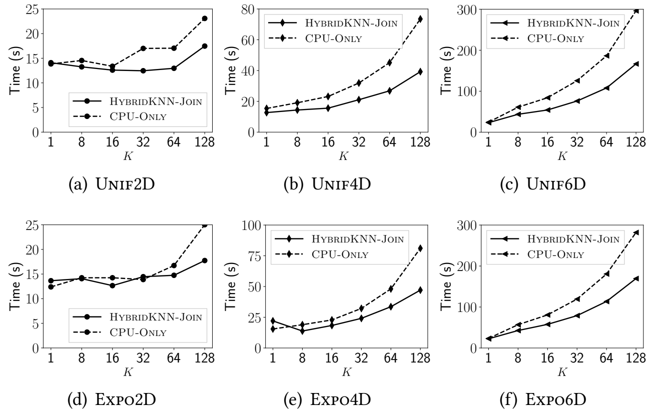
Figure 8: Load imbalance on the uniformly distributed datasets for $K = 8, 32, 128$. Default parameter values in Table 1 are used.

performance. On this 2-D dataset, the CPU is very efficient and Hybrid-GPU can reduce query points available for Hybrid-CPU to process. On the Unif6D dataset, we find that the best value of $n^{large} = 0.6$, which shows that Hybrid-GPU should be configured with large monolithic batches when computing larger workloads. However, we find that in Figure 7 (b), n^{large} can be selected in a large range to achieve good performance (the times are similar between $n^{large} = 0.35 - 0.6$). Hence, we select $n^{large} = 0.4$ in Table 1 to achieve a compromise between small and large workloads.

We determine whether the configuration of the work queue is able to mitigate load imbalance between CPU and GPU components. Figure 8 shows the load imbalance for $K = 8, 32, 128$ on the uniformly distributed datasets[2]. We observe that the load imbalance is high on Unif2D at $K = 32$, which can be caused by the GPU being assigned queries right before the last CPU rank finishes its batch. This occurrence is more likely when the workload/dimensionality is low and Hybrid-CPU is very efficient at processing *KNN* searches. On the 4-D and 6-D datasets, we find that the load imbalance is $< 10\%$ and increases with K. With the exception of the smallest workload, we find that HybridKNN-Join achieves reasonably good load balancing despite several confounding issues. To further mitigate load imbalance, additional approaches may include: (i) further decreasing the batch size, which may not be ideal for Hybrid-GPU, or (ii) forcing Hybrid-GPU to stop retrieving queries when only a small number are left to compute.

5.4.4 *Comparison of HybridKNN-Join to CPU-Only.*

Figure 9 (a)–(c) and (d)–(e) plots the response time vs. K on the uniform and exponentially distributed synthetic datasets, respectively. We find that on the 2-D datasets, HybridKNN-Join does not yield a substantial performance gain over CPU-Only with the exception of larger values of K. The overhead of finding a good ϵ value adds additional time that degrades performance when the workload is relatively low (e.g., low dimensionality and K). However, on the other datasets ($\geq$ 4 dimensions), we observe that HybridKNN-Join outperforms CPU-Only (excepting Figure 9 (e) at $K = 1$).



(a) Unif2D

(b) Unif4D

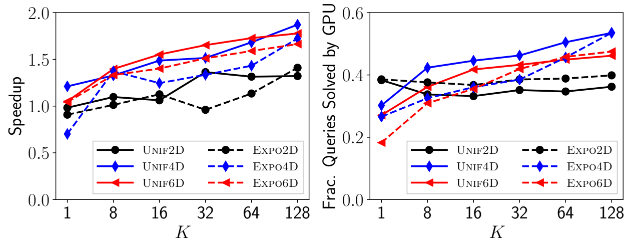
(c) Unif6D

(d) Expo2D

(e) Expo4D

(f) Expo6D

Figure 9: Response time vs. K comparing HybridKNN-Join and CPU-Only on (a)–(c) uniformly, and (d)–(f) exponentially distributed datasets. Default parameter values in Table 1 are used.



(a)

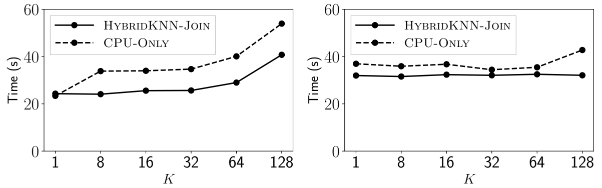
(b)

Figure 10: (a) Speedup of HybridKNN-Join over CPU-Only vs. K on synthetic datasets in Figure 9. Speedup 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 10 (a) plots the speedup of HybridKNN-Join over CPU-Only for all datasets in Figure 9. The figure demonstrates that the performance advantage of HybridKNN-Join is beneficial when K is large or when the dimensionality increases. Figure 10 (b) shows that as K increases, the fraction of queries solved by Hybrid-GPU also increases. 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.

Figure 11 (a) and (b) plot the response time vs. K on the real-world Gaia and Osm datasets, respectively. While these datasets are 2-D with $|D| = 2.5 \times 10^7$ points, they show that across all of K (except $K = 1$ on Gaia), HybridKNN-Join outperforms CPU-Only. These datasets are larger than the synthetic ones in Figure 9, demonstrating that even 2-D datasets benefit from a hybrid approach.

[2] 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.



(a) Gaia (2-D) (b) Osm (2-D)

Figure 11: Response time vs. K on 2-D real-world datasets. Default parameter values in Table 1 are used.

6 DISCUSSION & CONCLUSIONS

Many of the GPU *KNN* works address high-dimensionality [4, 11, 17, 23]. Here, we advance a hybrid approach for low-dimensionality that exploits the relative strengths of the CPU and GPU. 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 find that the speedup over the parallel CPU approach is < 2×; however, from Figure 10 (a), we clearly observe that the speedup of HybridKNN-Join is expected to be greater at higher dimensionality and K than the scenarios examined in this paper.

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 by reducing the batch size assigned to the GPU depending on the number of completed queries, and reserving queries for the CPU to prevent starvation. The work queue allows new advances in GPU- and CPU-only algorithms to be substituted into the 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. Future work directions include application to other algorithms and further work queue optimizations.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant OAC-1849559.

REFERENCES

[1] [n. d.]. Nvidia Volta. http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. Accessed: 31-01-2019.

[2] A Andoni and P Indyk. 2005. E^{2}LSH 0.1 User Manual. http://www.mit.edu/~andoni/LSH/manual.pdf.

[3] Alexandr Andoni and Piotr Indyk. 2006. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *IEEE Symposium on Foundations of Computer Science.* 459–468.

[4] Ahmed Shamsul Arefin, Carlos Riveros, Regina Berretta, and Pablo Moscato. 2012. Gpu-fs-knn: A software tool for fast and scalable knn computation using GPUs. *PLOS ONE* 7, 8 (2012), e44000.

[5] Sunil Arya, David M Mount, Nathan S Netanyahu, Ruth Silverman, and Angela Y Wu. 1998. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *J. ACM* 45, 6 (1998), 891–923.

[6] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *CACM* 18, 9 (1975), 509–517.

[7] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. 2001. *Parallel programming in OpenMP.* Morgan Kaufmann.

[8] A. Deshpande, I. Misra, and P. J. Narayanan. 2011. Hybrid implementation of error diffusion dithering. In *18th Intl. Conf. on High Performance Computing.* 1–10.

[9] Raphael A. Finkel and Jon Louis Bentley. 1974. Quad trees a data structure for retrieval on composite keys. *Acta informatica* 4, 1 (1974), 1–9.

[10] Gaia Collaboration, Brown, A. G. A., Vallenari, A., Prusti, T., de Bruijne, J. H. J., Babusiaux, C., et al. 2018. Gaia Data Release 2 - Summary of the contents and survey properties. *Astronomy & Astrophysics* 616 (2018), A1.

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

[12] M. Gowanlock and B. Karsin. 2018. GPU Accelerated Self-Join for the Distance Similarity Metric. In *Proc. of the 2018 IEEE Intl. Parallel and Distributed Processing Symposium Workshops.* 477–486.

[13] Michael Gowanlock, Cody M Rude, David M Blair, Justin D Li, and Victor Pankratius. 2017. Clustering Throughput Optimization on the GPU. In *Proc. of the IEEE Intl. Parallel and Distributed Processing Symposium.* 832–841.

[14] Antonin Guttman. 1984. R-trees: a dynamic index structure for spatial searching. In *Proc. of ACM Intl. Conf. on Management of Data.* 47–57.

[15] Tianyi David Han and Tarek S. Abdelrahman. 2011. Reducing branch divergence in GPU programs. In *Proc. of the 4th Workshop on General Purpose Processing on Graphics Processing Units.* 3:1–3:8.

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

[17] Liheng Jian, Cheng Wang, Ying Liu, Shenshen Liang, Weidong Yi, and Yong Shi. 2013. Parallel data mining techniques on graphics processing unit with compute unified device architecture (CUDA). *The Journal of Supercomputing* 64 (2013), 942–967.

[18] Dmitri V Kalashnikov. 2013. Super-EGO: fast multi-dimensional similarity join. *The VLDB Journal* 22, 4 (2013), 561–585.

[19] George Karypis, Eui-Hong Han, and Vipin Kumar. 1999. Chameleon: Hierarchical clustering using dynamic modeling. *Computer* 32 (1999), 68–75.

[20] Kimikazu Kato and Tikara Hosino. 2012. Multi-GPU algorithm for k-nearest neighbor problem. *Concurrency and Computation: Practice and Experience* 24, 1 (2012), 45–53.

[21] Jinwoong Kim, Won-Ki Jeong, and Beomseok Nam. 2015. Exploiting Massive Parallelism for Indexing Multi-Dimensional Datasets on the GPU. *IEEE Transactions on Parallel and Distributed Systems* 26, 8 (2015), 2258–2271.

[22] Jinwoong Kim and Beomseok Nam. 2018. Co-processing heterogeneous parallel index for multi-dimensional datasets. *J. Parallel and Distrib. Comput.* 113 (2018), 195–203.

[23] Ivan Komarov, Ali Dashti, and Roshan M D'Souza. 2014. Fast k-NNG construction with GPU-based quick multi-select. *PLOS ONE* 9, 5 (2014), e92409.

[24] Linchuan Li, Xingjian Li, Guangming Tan, Mingyu Chen, and Peiheng Zhang. 2011. 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, 195–204.

[25] Michael D Lieberman, Jagan Sankaranarayanan, and Hanan Samet. 2008. A fast similarity join algorithm using graphics processing units. In *IEEE 24th Intl. Conf. on Data Engineering.* 1111–1120.

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

[27] Marius Muja and David G Lowe. 2014. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis & Machine Intelligence* 11 (2014), 2227–2240.

[28] Moohyeon Nam, Jinwoong Kim, and Beomseok Nam. 2016. Parallel Tree Traversal for Nearest Neighbor Query on the GPU. In *45th Intl. Conf. on Parallel Processing.* 113–122.

[29] OpenStreetMap. [n. d.]. https://blog.openstreetmap.org/2012/04/01/bulk-gps-point-data/. Accessed 31-01-2019.

[30] Md Mostofa Ali Patwary, Nadathur Rajagopalan Satish, Narayanan Sundaram, Jialin Liu, Peter Sadowski, Evan Racah, Suren Byna, Craig Tull, Wahid Bhimji, Pradeep Dubey, et al. 2016. PANDA: Extreme Scale Parallel K-Nearest Neighbor on Distributed Architectures. In *Proc. of the 2016 Intl. Parallel and Distributed Processing Symposium.* 494–503.

[31] Nick Roussopoulos, Stephen Kelley, and Frédéric Vincent. 1995. Nearest Neighbor Queries. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data.* 71–79.

[32] Chenyi Xia, Hongjun Lu, Beng Chin Ooi, and Jing Hu. 2004. Gorder: an efficient method for KNN join processing. In *Proc. of the Intl. Conf. on Very Large Data Bases.* 756–767.

[33] Bin Yao, Feifei Li, and Piyush Kumar. 2010. K nearest neighbor queries and knn-joins in large relational databases (almost) for free. In *IEEE 26th Intl. Conf. on Data Engineering.* 4–15.

[34] Cui Yu, Bin Cui, Shuguang Wang, and Jianwen Su. 2007. Efficient index-based KNN join processing for high-dimensional data. *Information and Software Technology* 49, 4 (2007), 332–344.

[35] Y. Zhang, H. Ma, N. Peng, Y. Zhao, and X.-b. Wu. 2013. Estimating Photometric Redshifts of Quasars via the k-nearest Neighbor Approach Based on Large Survey Databases. *The Astronomical Journal* 146, Article 22 (2013).