P. Velentzas, M. Vassilakopoulos, A. Corral. (2021): GPU-aided edge computing for processing the k nearest-neighbor query on SSD-resident data. Internet of Things. Volume 15, September 2021, 100428 (Elsevier). ISSN: 2542-6605

GPU-aided edge computing for processing the *k* nearest-neighbor query on SSD-resident data

Polychronis Velentzas [a], Michael Vassilakopoulos [a,*], Antonio Corral [b]

[a] *Data Structuring & Eng. Lab., Dept. of Electrical & Computer Eng., Univ. of Thessaly, Volos, Greece*
[b] *Dept. of Informatics, University of Almeria, Almeria, Spain*

ARTICLE INFO

Article history:
Received 22 February 2021
Revised 13 June 2021
Accepted 15 June 2021
Available online 26 June 2021

Keywords:
k Nearest-neighbor query
GPU
SSD
Spatial query
Parallel algorithms
Edge computing

ABSTRACT

Edge computing aims at improving performance by storing and processing data closer to their source. The *k* Nearest-Neighbor (*k*-NN) query is a common spatial query in several applications. For example, this query can be used for distance classification of a group of points against a big reference dataset to derive the dominating feature class. Typically, GPU devices have much larger numbers of processing cores than CPUs and faster device memory than main memory accessed by CPUs, thus, providing higher computing power. However, since device and/or main memory may not be able to host an entire reference dataset, the use of secondary storage is inevitable. Solid State Disks (SSDs) could be used for storing such a dataset. In this paper, we propose an architecture of a distributed edge-computing environment where large-scale processing of the *k*-NN query can be accomplished by executing an efficient algorithm for processing the *k*-NN query on its (GPU and SSD enabled) edge nodes. We also propose a new algorithm for this purpose, a GPU-based partitioning algorithm for processing the *k*-NN query on big reference data stored on SSDs. We implement this algorithm in a GPU-enabled edge-computing device, hosting reference data on an SSD. Using synthetic datasets, we present an extensive experimental performance comparison of the new algorithm against two existing ones (working on memory-resident data) proposed by other researchers and two existing ones (working on SSD-resident data) recently proposed by us. The new algorithm excels in all the conducted experiments and outperforms its competitors.

1. Introduction

Edge computing is a distributed computing paradigm bringing computation and data storage as close to the source of data as possible. Its target is to improve performance by avoiding transferring data over a (possibly long-distance) network. Since a dataset may be rather big, instead of transferring it as a whole from the networks edge (e.g., the data collection point) to a centralized or cloud-based location for processing a demanding query, this rather limited-size query is transferred to a device at the networks edge. Then, this device processes this query on its locally stored data and transfers back only the rather limited-size result to the coordinating machine.

Modern applications utilize big spatial data which are collected from distant network edges. Processing of these data is demanding and the use of parallel processing plays a crucial role. Parallelism based on GPU devices is gaining popularity

https://doi.org/10.1016/j.iot.2021.100428

during last years [1]. A GPU device can host a very large number of threads accessing the same device memory. In most cases, GPU devices have much larger numbers of processing cores than CPUs and faster device memory than main memory accessed by CPUs, thus, providing higher computing power. GPU devices that have general computing capabilities appear in many modern commodity desktop and laptop computers. Therefore, GPU-devices can be widely used to efficiently compute demanding queries.

The k Nearest-Neighbor (k-NN) algorithm is widely used for spatial queries and for spatial distance classification in many problems areas (for example, k-NN classification has been used for economic forecasting [2]). Let a group of query points, for each of which we need to compute the k-NNs within a reference dataset to derive the dominating feature class. When the reference points volume is extremely big, delivering low latency results is possibly challenging and GPU-based techniques could improve efficiency. Furthermore, when the query points are originating fast from streams, the computational overhead is even larger and the need for new parallel methods arises.

In [3], we presented a new in-memory GPU-based algorithm for the k-NN query, using the CUDA run-time API [4]. This algorithm takes advantage of symmetrical partitioning, to efficiently compute the k-NN of all query points. Moreover, it utilizes a k-NN list buffer to avoid distance sorting of big datasets (resulting to expensive computations) and to store only k candidates for each query point (saving device memory). Through extensive experimentation, this algorithm was shown to outperform existing ones, in terms of execution time as well as total volume of in-memory reference points that can be handled.

Since GPU device memory is expensive, it is very important to take advantage of this memory as much as possible and scale-up to larger datasets. However, since device and/or main memory may not be able to host an entire, rather big, reference dataset, storing this dataset in a fast secondary device, like a Solid State Disk (SSD) is, in many practical cases, a feasible solution.

Considering the above scenery of large amounts of data being collected from distant locations, the need for storing these data in fast secondary devices and, at the same time, for processing demanding queries from them and the processing benefits GPU-enabled devices can offer, in this paper:

- We propose an architecture of a distributed edge-computing environment where large-scale processing of the k-NN query can be accomplished by executing an efficient algorithm for processing the k-NN query on the GPU and SSD enabled edge nodes of this environment.
- We propose a new algorithm for this purpose, a GPU-based partitioning algorithm for processing the k-NN query on big reference data stored on SSDs. This algorithm extends the algorithm of [3], which utilizes memory resident data only. Moreover, it stores the candidate neighbors of each query point in an array-based distance list buffer, while the algorithm proposed in this paper has two variations, one which uses the same buffer as [3] and another which uses a max-Heap distance list buffer (as proposed in [5,6]).
- We implement this algorithm in a GPU-enabled edge-computing device, hosting reference data on an SSD. We have chosen a popular such device, NVIDIA Jetson Nano (https://developer.nvidia.com/embedded/jetson-nano-developer-kit). It is an IoT device specialized for edge computing. It is vastly used by professional developers to create breakthrough AI products across all industries. It features CPU-GPU heterogeneous architecture where CPU can boot the OS and the CUDA-capable GPU can be quickly programmed to accelerate complex machine-learning tasks [7].
- Using synthetic datasets, we present an extensive experimental performance comparison of the new algorithm against two existing ones proposed by other researchers for GPU-based processing of the query under study on memory-resident data and two existing ones recently proposed by us in [6] for GPU-based processing of the same query on disk resident data. Our algorithms in [6] are the first GPU-based ones for processing the k-NN query for big reference data stored on SSDs and each of them has a variation which stores the candidate neighbors of each query point in an array-based distance list buffer and another one which uses a max-Heap distance list buffer for the same purpose. The results show that our new algorithm outperforms all its competitors.

The rest of this paper is organized as follows. In Section 2, we review related material and present the motivation for our work, while in Section 3 we present our proposal of the distributed edge-computing environment architecture for processing queries like the k-NN one. Next, in Section 4, we briefly present the algorithms of [6], present the new algorithm that we developed for the k-NN GPU-based processing and the two buffering methods utilized by all three algorithms. In Section 5, we present the experimental study that we performed for analyzing the performance of our algorithm and for comparing it to algorithms by other researchers and the algorithms presented in [6]. Finally, in Section 6, we present the conclusions arising from our work and discuss our future plans.

2. Related work and motivation

A recent trend in the research for parallelization of nearest neighbor search is to use GPUs. Parallel k-NN algorithms on GPUs can be usually implemented by employing a *Brute-Force* methods or by using *Space Subdivision* techniques.

2.1. Brute-Force techniques

k-NN on GPUs using a Brute-Force method applies a two-stage scheme: (1) the computation of distances and (2) the selection of the nearest neighbors by using sorting algorithms [8]. For the first stage, a distance matrix is built grouping the distance array to each query point. In the second stage, several selections are performed in parallel on the different rows of the matrix. In the last decade, many Brute-Force approaches have been proposed in the literature, and the most representative ones are briefly reviewed in the following.

Garcia et al. [9] was one of the first approaches to implement a Brute-Force *k*-NN algorithm on GPUs, highlighting two important characteristics: (1) each thread computes the distance between a given query point and a reference point, and (2) each thread sorts, by using an *insertion sort* algorithm, all the distances computed for a given query point.

In [10], the distance matrix is split into blocks of rows and each matrix row is sorted using *radix sort* method, obtaining a performance more than 10x faster than the sequential counterpart. Moreover, the authors used a segmentation method for pair-wise distance computations.

Liang et al. [11] proposed the *CUKNN* algorithm, a CUDA based parallel implementation of *k*-NN. It used the same approach as [9] and [10] to compute the distance matrix. But for the selection phase, a local *k*-NN for each block of threads is computed, then merging and sorting them in order to obtain a global *k*-NN.

In [12], an improved GPU-based approach by using the CUBLAS (CUDA Basic Linear Algebra Subroutines) API is proposed for a faster Brute-Force *k*-NN parallelization to efficiently calculate a distance matrix. A modified version of the *insertion sort* algorithm proposed in [9] is applied when each column of the distance matrix is sorted.

In [13], a GPU heap-based algorithm (called *Batch Heap-Reduction*) is presented, which achieves a better performance than the sorting-based GPU-Quicksort algorithms. The Batch Heap-Reduction algorithm uses a heap for each thread of a CUDA Block, by means of a three-step algorithm to obtain the final *k*-NN. First, the distance vector is evenly distributed across the CUDA block threads. Each thread determines its own partial *k*-NN by the *heap sort* algorithm. The other two stages implement the reduction of the partial heaps.

In [14], the *truncated sort* algorithm is introduced in the selection phase for *k*-NN search. For this sorting algorithm, elements are discarded from the sorting when it is clear that they cannot belong to the smallest *k*. That is, the algorithm first locates the *k*th element as a threshold, then searches all the elements smaller than that threshold, followed by another search to finish the *k*-list with elements equal to the threshold.

In [15], the *GPU-FS-kNN* algorithm is presented. It divides the computation of the distance matrix into smaller submatrices (squared chunks) in all dimensions in order to parallelize distance calculations and *k*-NN search over these chunks. Each chunk is computed using a different kernel call, reusing the allocated GPU-memory. In selection phase, each chunk is processed with a modified version of the *insertion sort* algorithm.

Kato and Hosino [16] proposed a Brute-Force *k*-NN algorithm that is also suitable for several GPU devices. The distance matrix is split into blocks of rows where each thread computes the distances for a matrix row. Then a max-heap is built for each query and parallel threads push new candidates to the max-heap using atomic operations. Therefore, only smallest *k* distances are computed by *heap sort* algorithm in parallel on each thread in the thread block.

In [17], a hybrid parallelization approach for Brute-Force computation of multiple *k*-NN queries on GPUs is proposed. For the matrix computation uses the [10] and [12] scheme, modifying the selection phase with a *quicksort*-based selection. An additional optimization was implemented, using voting functions available on the latest GPUs along with user-controlled cache.

In [18], a Brute-Force *k*-NN implementation is proposed by using a modified inner loop of the SGEMM kernel in MAGMA library, a well-optimized open-source matrix multiplication kernel. Besides, they search only *k* smallest squared distances for each query by using the merge-path function from the Modern GPU library and a *truncated merge sort*.

In [19], an incremental neighborhood computation scheme that eliminates the dependencies between the dataset size and memory is presented. As a result, a new scalable and memory efficient design for a GPU-based *k*-NN rule, called *GPU-SME-kNN*, is proposed. It takes advantage of asynchronous memory transfers, making the data structures fit into the available memory while delivering high run-time performance independently of the data size.

In [20], Brute-Force approaches to solving *k*-NN queries in GPUs on the *selection sort, quick sort* and state-of-the-art *heaps-based* algorithms are proposed. Due to the fact that the best approach depends on the *k* value of the *k*-NN query, the authors also proposed a multi-core algorithm to be used as reference for the experiments and a hybrid algorithm which combines the proposed sorting algorithms.

In [21], a Brute-Force parallel algorithm to solve *k*-NN queries on a multi-GPU platform is presented. The proposed method is comprised of two stages, which first is based on pivots using the value of *k* to reduce the search space, and the second one uses a set of heaps to return the final results.

Some of these Brute-Force algorithms (like the ones of [12] and their improved implementations [22]) consume a lot of device memory, since a Cartesian product matrix, containing the distances of reference points to the query points, is stored. In [23], two new algorithms based on GPUs to process *k*-NN queries on spatial data are proposed, using the Thrust library [1], that maximize device memory utilization. The first algorithm is based on Brute Force scheme and the second one uses heuristics to minimize the reference points near a query point.

https://doi.org/10.1016/j.iot.2021.100428

2.2. Spatial subdivision techniques

Spatial subdivision is a well-known technique for improving the query performance used in a variety of applications. There are many data structures that handle spatial subdivision efficiently and, they can be used as GPU index-based data structures to find effective k-NN. The most representative approaches of this category are briefly reviewed in the following.

kd-trees [24] have been successfully used for nearest neighbor searching for long time. For this reason, several variations of kd-trees have been implemented on GPUs. In [25], an algorithm for constructing kd-trees on GPUs is developed. The building process adopts a top-down, breadth-first search order, starting from the root bounding box. The k-NN implementation is based on a range search on the tree (with a given radius), and it continues to increase the size of the radius until k elements are retrieved. In [26], a *buffer kd-tree* for GPUs is presented. The buffer kd-tree algorithm avoids several drawbacks of the GPU's architecture. In particular, the buffer refers to a query buffer located in every node of kd-tree, which is used to delay the execution of queries by waiting for sufficient work to be accumulated into a buffer before accessing leaf nodes. Each node in the buffer kd-tree corresponds to a set of reference patterns. Therefore, a lazy nearest neighbor search schema is applied. The algorithm also focuses on improving the fraction of coalesced memory accesses by having threads within a warp access either consecutive or nearby memory addresses.

Locality Sensitive Hashing (LSH) [27] was introduced as a solution to approximate nearest neighbor problem. It is a hashing based indexing structure that clusters the data points to the closer hash buckets by using multiple probabilistic hash functions and storing them in different hash tables. Several contributions have been proposed for LSH-based similarity searching algorithms using GPUs, and the most representative ones are [28,29], where variants of LSH are built to develop an efficient GPU-based parallel LSH algorithm to perform approximate k-NN computation in high-dimensional spaces. The running times of these approximate methods are competitive with existing Brute-Force implementations, but they return approximate results. Another approximate k-NN approach is proposed in [30], where the idea of product quantization is extended. The algorithm also includes a parallelizable re-ranking method for candidate vectors by efficiently reusing already computed intermediate values that can be used in a parallel GPU implementation.

Efficient spatial indexing structures such as R-trees [31] are promising in speeding up such computing on GPUs; therefore, several papers have been proposed for this purpose. The most significant one is [32], where parallel designs of bulk loading R-trees and several parallel query processing techniques (range query) on GPUs using R-trees are implemented. Moreover, in [33], a parallel bottom-up construction of SS-tree [34] on GPUs is proposed. And the develop a data parallel tree traversal algorithm, called *Parallel Scan and Backtrack* (PSB), for k-NN query processing on the GPU. This algorithm traverses a SS-tree index while avoiding warp divergence problems. In order to take advantage of accessing contiguous memory blocks, the proposed PSB algorithm performs linear scanning of sibling leaf nodes, which increases the chance to optimize the parallel algorithm.

In the context of a spatial index, a grid structure is a regular tessellation of a manifold that divides the space into a series of contiguous cells, which can then be assigned unique identifiers and used for spatial indexing purposes. According to this subdivision of the space, a GPU grid-based data structure is appropriate for massively parallel nearest neighbor searches over dynamic point datasets. A key contribution is [35], where a grid-based indexing solution for 3-dimensional k-NN searches on the GPU is proposed. The k-NN algorithm works as follows: for a given query point, the algorithm expands the number of grid cells searched to ensure that at least k neighbors are found. That is, the algorithm uses a query-centric approach that expands the search radius when the number of found neighbors is less than k. The proposed k-NN algorithm minimizes the memory transfer between device and system memories, improving overall performance. More recently, in [36], the Adaptive Inverse Distance Weighting (AIDW) interpolation algorithm on GPU is presented, where a fast kNN search approach based on an even grid is used.

Effective spatial data partitioning [37] is critical for task parallelization, load balancing, and directly affects system performance. A proper spatial partitioning schema is essential for optimal query performance and system efficiency for parallel spatial query processing. Keeping this in mind, in [3], a new algorithm for the k-NN query processing in GPUs is presented. It implements a new GPU-based partitioning algorithm based on a sort-tile partitioning method for the k-NN query (called *Symmetric Progression Partitioning*, SPP), using the CUDA runtime API, avoiding the calculation of distances for the whole dataset. Moreover, this k-NN query algorithm maximizes the utilization of device memory (using *KNN-DLB buffer*) and therefore permits larger reference datasets to take part in the processing of the query. Thus, by processing only the necessary parts of the reference dataset and by executing the whole process in the GPU device only, it minimizes execution speed. A thorough experimental evaluation proved that the proposed algorithm, not only works faster than existing methods, but also scales-up to much larger reference datasets.

2.3. Motivation

Flash-based Solid State Drives (SSDs) have been widely used as secondary storage in database servers because of their improved characteristics compared to Hard Disk Drives (HDDs) to manage large-scale datasets [38]. These characteristics include smaller size, lighter weight, lower power consumption, better shock resistance, and faster reads and writes. In fact, in the recent years, SSD storage devices, based on NAND flash technology, started replacing magnetic disks due to their appeal-

https://doi.org/10.1016/j.iot.2021.100428

ing characteristics: high throughput/low latency, shock resistance, absence of mechanical parts and low power consumption. In these secondary devices, read operations are faster than writes, while difference exist among the speeds of sequential and random I/Os as well. Moreover, the high degree of internal parallelism of latest SSDs substantially contributes to the improvement of I/O performance [39].

In this paper, we significantly extend our work in [3] adapting it to reference data stored on SSD storage. We present a new algorithm, called "Disk Symmetric Progression Partitioning" (Section 4.3), which has two variations, depending on the list-buffer structure used for storing nearest neighbors of each query point: either an array based list buffer (Section 4.4), as in [3], or a max-Heap based buffer (Section 4.5), as in [5]. To the best of our knowledge, our recent paper [6] and this paper are the first ones to deal with GPU processing of the *k*NN query on SSD-resident data.

Edge computing devices are based on ultra-low-power solutions, and they are designed with a higher computational power relying on heterogeneous processors. As we can see above, many articles in the literature have discussed parallel versions of *k*-NN on GPUs of typical (non-edge) computing devices.

In this paper, we present a distributed architecture embedding nodes performing edge-based query processing. We also apply the variations of our new algorithm to an edge computing device (NVIDIA Jetson Nano) which can be used within such an architecture and compare it against existing algorithms of other researchers [22,36] and algorithms developed by us [6]. None of the methods by previous researchers used disk-resident data, so we chose two rather recent ones which could be adapted to loading their dataset from disk.

To the best of our knowledge, our work is the first effort investigating the potentials and the possibilities of the modern edge computing devices (like NVIDIA Jetson Nano) in the context of performing *k*-NN algorithms with big spatial datasets stored in SSD storage devices. The most closely related works, using the NVIDIA Jetson Nano as edge device to study its performance in the solution of problems different to the one we study and without utilizing a fast secondary device for big data storage, are: [40], where an evaluation of clustering algorithms on GPU-based modern edge computing platforms is presented; [41,42], where the computational capability of up-to-date accelerator-based edge devices in the context of scientific computing is evaluated; and [43], where a deep learning benchmark on the latest GPU-accelerated edge devices to measure its performance is proposed.

3. Edge computing with IoT distributed architecture

If the data to be processed is too large to be handled in a centralized system, a distributed computing environment could be utilized. In most cases, a cluster of computers interconnected through a fast LAN could handle data which are larger by orders of magnitude. When dealing with IoT a fast network connection is not a feasible option. IoTs are remotely deployed and the interconnection network is slow. For example, in agricultural field deployments the IoTs are equipped with a multitude of sensors and deliver their raw data to a gateway, through wireless networks like Lora, Teensy, XBee, Beaglebone and others. The data is stored in the gateway device and then transmitted to a remote database through an API or a MQTT technology. In this context, the gateways are responsible for delivering all the data occasionally with missing values (network unavailability or power downs). Furthermore, most of the transmitted data will be poorly or maybe never processed by an analysis service. It will not be processed because the density of the data is excessive (usually an aggregate or an approximation of the data is needed) or it will never be queried. It makes sense, for the aforementioned reasons and when the data is not so critical, the raw sensor data not to be transmitted to the remote database. Due to the nature of the queries studied, the related algorithms can be easily applied in such a distributed setting. If data is distributed among edge computing nodes (the gateways), each node can compute the query result for its local data, stored on SSD. The query batch would be transmitted to every remote node, which would compute the answer based on its own data, and the results from all nodes would be sent to and merged by an Analytics Server acting as a coordinator of query processing. This is possible, since the answer of queries like the one we study for a node is independent to the answers for other nodes. An architecture that makes such processing possible is presented in Fig. 1. The application of this architecture can be done in agricultural fields deployments, where we collect agricultural and meteorological data.

Furthermore, this approach will work efficiency because the data are not blindly distributed among nodes. Each node keeps data that are spatially close, even in case partial intersection between the areas covered by nodes is allowed.

As shown in Fig. 1, on the client side, a request of a batch *k*-NN query is transferred to the Web Server. The Web Server parses the request and requests a *k*-NN query result from the Analytics Server. The Analytics Server requests the query results from every edge computing node, in parallel way. The query results are accumulated and merged in the Analytics Server. The Analytics Server is equipped with a GPU and computes in the GPU device the final result. The query results are transferred to the Web Server and the client request is served.

To achieve this interoperability, appropriate APIs should be developed. The Edge Computing nodes can be integrated with a Node.js instance which can serve such requests. Another possible integration for the back-end can be Ballerina.io, which is a cloud native development language.

In the rest of the paper, we present our proposal for an efficient *k*-NN algorithm to be executed in the edge computing nodes of such a distributed architecture, taking advantage both of SSD storage and multiple GPU cores, to accelerate spatial query processing.

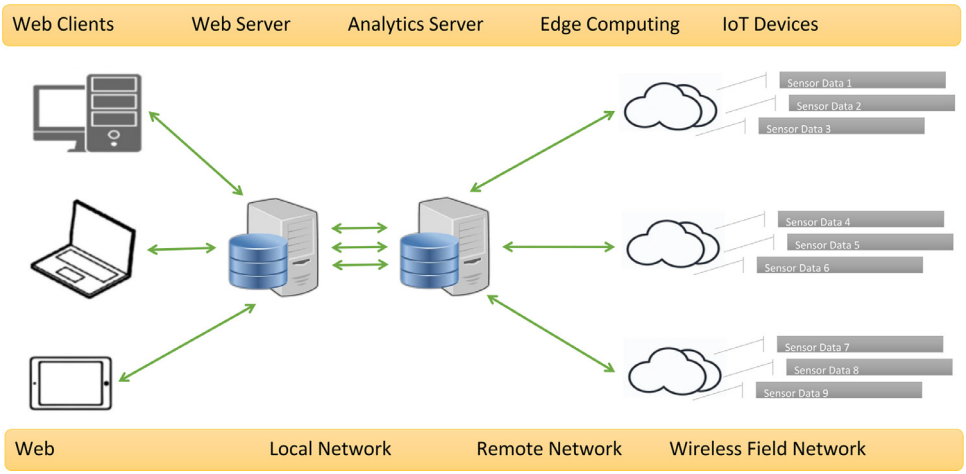


Fig. 1. Edge Computing Architecture.

4. *k*NN Disk Algorithms

A common practice to handle big data is data partitioning. In order to describe our new algorithms, we should firstly present the mechanism of data partition transfers to device memory. This step is identical in all our methods. Each reference dataset is partitioned in N partitions containing an equal number of reference points. If the total number of reference points is not divided exactly with N, the Nth partition contains the remainder of the division. Initially the host (the computing machine hosting the GPU device) reads a partition from SSD[1] and loads it into the host memory. The host copies the in-memory partition data into the GPU device memory (the whole process is outlined in Fig. 2).

Another common approach in all our four methods is the GPU thread dispatching. Every query point is assigned to a GPU thread. The GPU device starts the k-NN calculation simultaneously for all threads. If the number of query points is bigger than the total available GPU threads, then the execution progresses whenever a block of threads finishes the previously assigned query points calculation. The thread dispatching consists of the following four main steps (also depicted in Fig. 3):

1. The kernel function requests N threads.
2. The requested N threads are assigned to N query points.
3. Every thread carries out the calculation of reference point distances to its query point and updates the k-NN buffer holding the current (and eventually the final) nearest neighbors of this point.
4. The final k-NN list produced by each algorithm is populated with the results of all the query points.

In the next sections, we will briefly describe two of our existing methods and in detail a new one we developed. The two existing methods are based on “Disk Brute-force” and “Disk Plane-sweep” [6]. The new one is based on the “Symmetric Progression Partitioning Algorithm” [3]. For all methods we have two implementations of k-NN buffer variations resulting to a total of four existing methods and two new ones (algorithmic variations).

4.1. Disk Brute-Force algorithm

The Disk Brute-force algorithm (denoted by DBF) [6] is a Brute-force algorithm enhanced with capability to read SSD-resident data. Brute-force algorithms are highly efficient when executed in parallel. The algorithm accepts as inputs a reference dataset R consisting of m reference points $R = \{r_1, r_2, r_3 \dots r_m\}$ in 3d space and a dataset Q of n query points $Q = \{q_1, q_2, q_3 \dots q_n\}$ also in 3d space. The host reads the query dataset and transfers it in the device memory. The reference dataset is partitioned in equally sized bins and each bin is transferred to the device memory. For each partition, we apply the k-NN Brute-force computations for each of the threads.

For every reference point within the loaded partition, we calculate the Euclidean distance to the query point of the current thread. Every calculated distance is compared to the current thread maximum distance and if it is smaller we add it to the k-NN list buffer. We will use and compare two alternative k-NN buffer implementations, presented in Sections 4.4 and 4.5.

[1] Reading from SSD is accomplished by read operations of large sequences of consecutive pages, exploiting the internal parallelism of SSDs, although our experiments showed that reading from SSD does not contribute significantly to the performance cost of our algorithms.

https://doi.org/10.1016/j.iot.2021.100428

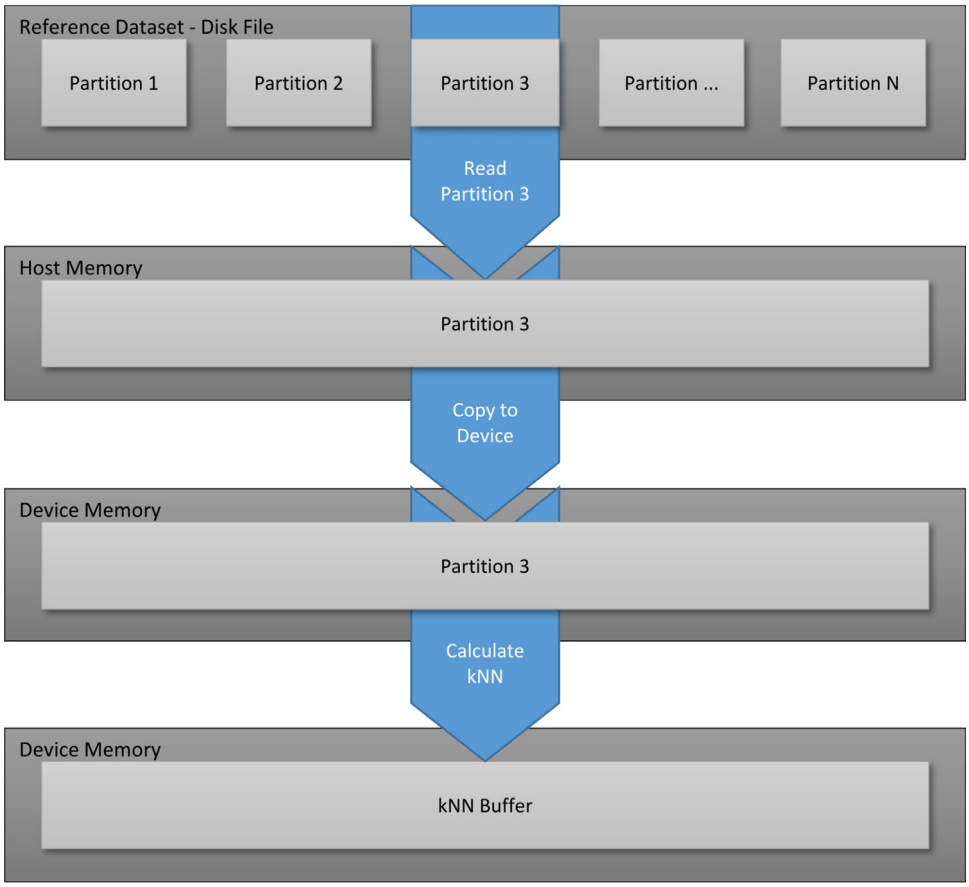


Fig. 2. Datafile partitioning and loading of partitions into device's memory.

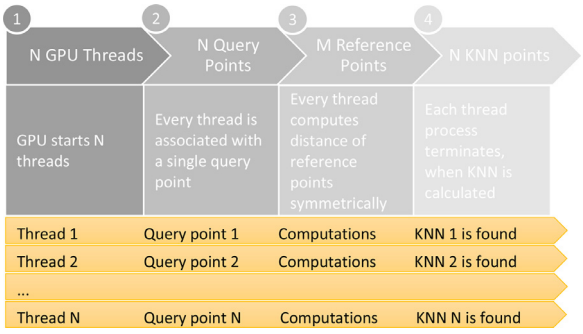


Fig. 3. GPU thread dispatching. Every thread computes one query point [3].

4.2. Disk Plane-sweep algorithm

An important improvement for join queries is the use of the Plane-sweep technique, which is commonly used for computing intersections [44]. The Plane-sweep technique is applied in [45] to find the closest pair in a set of points which resides in main memory. The basic idea, in the context of spatial databases, is to move a line, the so-called sweep-line, perpendicular to one of the axes, e.g., X-axis, from left to right, and process objects (points, in the context of this paper) as they are reached by this sweep-line. We can apply this technique for restricting all possible combinations of pairs of objects from the two datasets. The Disk Plane-sweep algorithm (denoted as DSP) [6] incorporates this technique which is further enhanced with capability to read SSD-resident data.

Like DBF, DSP accepts as inputs a reference dataset R consisting of m reference points $R = \{r_1, r_2, r_3 \dots r_m\}$ in 3d space and a dataset Q of n query points $Q = \{q_1, q_2, q_3 \dots q_n\}$ also in 3d space. The host reads the query dataset and transfers it in

https://doi.org/10.1016/j.iot.2021.100428

the device memory. The reference dataset is partitioned in equally sized bins, each bin is transferred to the device memory and sorted by the x-values of its reference points. For each partition we apply the k-NN Plane-sweep technique.

4.3. Disk symmetric progression partitioning

We designed and implemented the novel method "Disk Symmetric Progression Partitioning", denoted as DSPP. DSPP is enhanced with the capability to read SSD-resident big data. The DSPP algorithm is using partitioning in both the host and the GPU device. The first level partitioning is taking place in the host, when reading data from the SSD-resident datasets, as we discussed in Section 4. The second level partitioning is taking place in the device memory and is essential for the DSPP execution. We will document in detail the usage of the two distinct levels of partitioning later in this section.

Like DBF and DSP, DSPP accepts as inputs a reference dataset R consisting of m reference points $R = \{r_1, r_2, r_3 \dots r_m\}$ in 3d space and a dataset Q of n query points $Q = \{q_1, q_2, q_3 \dots q_n\}$ also in 3d space. The host reads the whole query dataset and transfers it in the device memory. The reference dataset is partitioned in equally sized bins (first level partition). Each partition is transferred to the device memory and sorted by the x-values of its reference points (Algorithm 1). The host

Algorithm 1 DSPP Host algorithm.

Input: NN cardinality=K, Reference filename=RF, Query filename=QF,Partition size=S, sub-Partition cardinality=CB
Output: Host k-NN Buffer=HostKNNBufferVector

```
1: HostQueryVector ← readFile(QF);
2: queryPoints ← HostQueryVector.size();
3: DeviceQueryVector ← HostQueryVector;
4: createEmptyHostVector(HostKNNBufferVector,queryPoints*K);
5: DeviceKNNBufferVector ← HostKNNBufferVector;
6: subPartitionPoints ← S/CB;
7: while not end-of-file RF do
8:     HostReferencePartition ← readPartition(RF,S);
9:     DeviceReferencePartition ← HostReferencePartition;
10:    cudaSort(DeviceReferencePartition);
11:    HostReferencePartition ← DeviceReferencePartition;
12:    HostReferencePartitionIndex.clear();
13:    HostReferencePartitionIndex.add(0,HostReferencePartition[0].x);
14:    for i=0 to CB-1 do
15:        HostReferencePartitionIndex.add(HostReferencePartition[i*subPartitionPoints].x,
               HostReferencePartition[(i+1)*subPartitionPoints].x);
16:    DeviceReferencePartitionIndex ← HostReferencePartitionIndex;
17:    runKNN<<<(queryPoints-1)/256 +1, 256>>>(DeviceReferencePartition,
           DeviceQueryVector,DeviceKNNBufferVector,K,DeviceReferencePartitionIndex);      // 256 cores assumed
18: HostKNNBufferVector ← DeviceKNNBufferVector;
```

fetches back the sorted partition from the device in order to further partition it, into smaller sub-partitions (second level partition), and prepare the sub-partition index data for the SPP execution. This second partitioning will be taking place in the device memory and further accelerates the k-NN process, as we will prove experimentally. The host process, as a last step, executes the device kernel program.

From the device scope, every query point is assigned to a GPU thread. The GPU starts the k-NN calculation simultaneously for all threads. If the number of query points is bigger than the total available GPU threads, then the execution progresses whenever a block of threads finish the previously assigned query points calculation. For every partition the host reads, our algorithm processes 4 main steps (Fig. 3):

1. The kernel function requests N threads
2. The requested N threads are assigned to N query points
3. Every thread carries out the calculation of DSPP
4. The k-NN list is populated with the results of all the query points

The in-device-memory partitioning technique we are using, partitions the dataset in equally sized sub-partitions throughout the X-axis. DSPP searches for k-NN, traversing the partition index (Algorithm 2), that the host provided, and checks if its bounding box contains the query point (sub-partition number 5, in our example, Fig. 4). If k-NN are not found the thread searches for k-NN in the next closest sub-partition (sub-partition 6). Similarly, the process continues until all reference points are processed. In Fig. 4 we search for 20 nearest neighbors. We processed 7 out of 10 partitions and found the k-NN. Sub-partitions 1, 9 and 10 were excluded because the 20 nearest neighbors were already found.

https://doi.org/10.1016/j.iot.2021.100428

Algorithm 2 DSPP Device Kernel algorithm (runKNN).

Input: NN cardinality=K, Partition Reference array=R, Query array=Q, Partition size=S,
Device Partition Index=DeviceReferencePartitionIndex
Output: Device k-NN Buffer array=DKB

```
1: qIdx ← blockIdx.x*blockDim.x+threadIdx.x;
2: knnBufferOffest ← qIdx*K;
3: for currentPartition ← 0 to DeviceReferencePartitionIndex.size()-1 do
4:     if DeviceReferencePartitionIndex[currentPartition].right-X-Limit<Q[qIdx].x thenbreak;
5: if currentPartition < DeviceReferencePartitionIndex.size()-1 then currentPartition--;
6: while maxdistance>Q[qIdx].x-DeviceReferencePartitionIndex[currentPartition].left-X-Limit or
       maxdistance>DeviceReferencePartitionIndex[currentPartition].right-X-Limit-Q[qIdx].x do
7:     idx1 ← currentPartition * R.size();
8:     idx2 ← (currentPartition+1) * R.size();
9:     for i ← idx1 to idx2-1 do
10:        dist ← sqrt[2]((R[i].x − Q[qIdx].x)^2 + (R[i].y − Q[qIdx].y)^2 + (R[i].z − Q[qIdx].z)^2)
11:        insertIntoBuffer(DKB,knnBufferOffest,i,qIdx,dist);
12:    currentPartition ← FindNextClosestPartition;
```

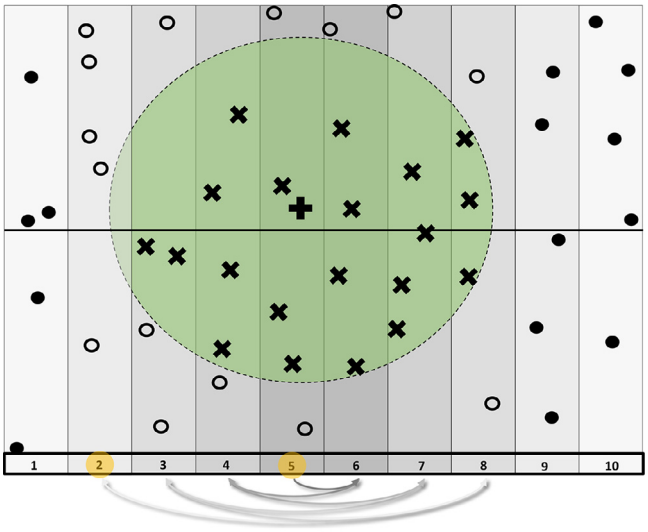


Fig. 4. DSPP sub-partition example, query point represented by + symbol, reference points represented by × symbols (k-NN points), analyzed points represented by empty circles, non-analyzed points represented by filled circles, $k = 20$ [3].

The host algorithm continuous to read partitions from the reference dataset, process them and execute the device kernel, until the reference dataset is fully read. Every kernel execution, merges into the k-NN buffer list the calculated distances that are shorter than the maximum current ones and produces the final k-NNs upon read reference data completion.

4.4. k-*NN distance list buffer*

In our methods we implemented two different k-NN list buffers. The first one is the k-NN Distance List Buffer (denoted by KNN-DLB) [5,6]. KNN-DLB is an array where all calculated distances are stored (Table 1). KNN-DLB array size is k per thread, resulting to a minimum possible device memory utilization. When the buffer is not full, we append the calculated distances. When the buffer is full, we compare every newly calculated distance with the largest one stored in KNN-DLB. If it is smaller, we simply replace the largest distance with the new one. Therefore, we do not utilize sorting. The resulting buffer contains the correct k-NNs, but not in an ascending order. The usage of KNN-DLB buffer is performing better than sorting a large distance array [3].

4.5. k-*NN max-Heap distance list buffer*

The second list buffer that we implemented is based on a max-Heap (a priority queue represented by a complete binary tree which is implemented using an array) [5,6]. max-Heap array size is $k + 1$ per thread, because the first array element is occupied by a sentinel. The sentinel value is the largest value for double numbers (for C++ language, used in this work, it

https://doi.org/10.1016/j.iot.2021.100428

Table 1
KNN Distance List Buffer, k=10.

	New Distance	1	2	3	4	5	6	7	8	9	10
First 10 distances are appended to the list	5.1	**5.1**									
	2.7	5.1	**2.7**								
	4.0	5.1	2.7	**4.0**							
	2.8	5.1	2.7	4.0	**2.8**						
	11.2	5.1	2.7	4.0	2.8	**11.2**					
	1.7	5.1	2.7	4.0	2.8	11.2	**1.7**				
	3.5	5.1	2.7	4.0	2.8	11.2	1.7	**3.5**			
	0.6	5.1	2.7	4.0	2.8	11.2	1.7	3.5	**0.6**		
	0.1	5.1	2.7	4.0	2.8	11.2	1.7	3.5	0.6	**0.1**	
	7.1	5.1	2.7	4.0	2.8	11.2	1.7	3.5	0.6	0.1	**7.1**
Distances smaller than the maximum distance, replace it	8.5	5.1	2.7	4.0	2.8	**8.5**	1.7	3.5	0.6	0.1	7.1
	6.9	5.1	2.7	4.0	2.8	**6.9**	1.7	3.5	0.6	0.1	7.1
	1.6	5.1	2.7	4.0	2.8	6.9	1.7	3.5	0.6	0.1	**1.6**
	5.8	5.1	2.7	4.0	2.8	**5.8**	1.7	3.5	0.6	0.1	1.6

Table 2
SpiderWeb Dataset generator parameters, for the existing algorithms comparison experiment.

Distribution	Cardinality	Seed	Binary File Size	Usage
Bit	10K	1	312KB	Reference Dataset
Bit	20K	2	625KB	Reference Dataset
Bit	30K	3	937KB	Reference Dataset
Bit	40K	4	1250KB	Reference Dataset
Bit	50K	3	1562KB	Reference Dataset
Bit	100K	4	3125KB	Reference Dataset
Gaussian	2K	6	62KB	Query Dataset

is the constant DBL_MAX). KNN-DLB is adequate for smaller k values, but when k value increases performance deteriorates, primarily due to KNN-DLB O(n) insertion complexity. On the other hand, max-Heap insertion complexity is O(log(n)) and for large enough k max-Heap implementations are expected to outperform KNN-DLB ones.

5. Experimental study

We run a large set of experiments to compare the repetitive application of our SSD-resident data algorithms for processing batch k-NN queries. We performed two kinds of experiments. First, we compared to existing methods by other researchers (working on memory-resident data) and to methods recently proposed by us (working on SSD-resident data). Next, we performed scaling experiments of our existing and new methods. The scaling experiments query at least 5M reference points. We did not include less than 5M reference points because we target reference datasets that do not fit in the GPU device memory. We experimentally found that the largest dataset that we could fit in device memory is about 1M reference points. The numeric accuracy for storing point data, is double precision (Algorithm 3). Another parameter that we

Algorithm 3 Point Structure.

```
    // Point record structure, used in reference datasets. Record size 32 bytes
1:  record point_struct begin
2:      id,      // Point ID, 8 bytes
3:      x, y, z      // Coordinates of 3D space, 8 bytes per dimension
4:  end;
```

evaluated is the list buffer performance and we will highlight the pros and cons of using KNN-DLB or max Heap buffer.

All the datasets were created using the SpiderWeb [46,47] generator. This generator allows users to choose from a wide range of widely accepted spatial data distributions and configure the cardinality of the data and the distribution parameters. This generator has been successfully used in existing research to evaluate index construction, query processing, spatial partitioning, and cost model verification, as reported in [47]. While some real spatial datasets are available in repositories and they can be used for testing the performance of spatial algorithms, researchers also need to have full control of the parameters of data and improve the reproducibility of experiments.

Table 2 lists all the generated datasets, that will be used in the existing methods comparison experiments. For the reference dataset we created six datasets using the "Bit" distribution (Fig. 5), with file sizes ranging from 312KB to 3.125KB. The reference points cardinality ranges from 10K points to 100K points. For the query points dataset we created one "Gaussian" dataset of 2K points.

https://doi.org/10.1016/j.iot.2021.100428

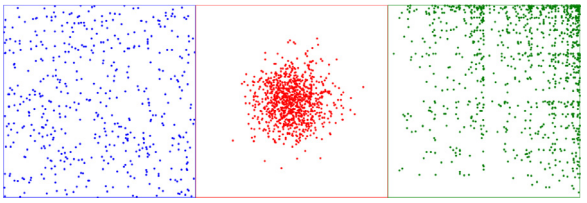


Fig. 5. Experimental data distributions, Blue=Uniform, Red=Gaussian, Green=Bit.

Table 3
SpiderWeb Dataset generator parameters, for the new methods scaling experiments.

Distribution	Cardinality	Seed	Binary File Size	Usage
Bit	5M	1	153MB	Reference Dataset
Bit	10M	2	306MB	Reference Dataset
Bit	15M	3	458MB	Reference Dataset
Bit	20M	4	611MB	Reference Dataset
Uniform	10	5	32B	Query Dataset
Gaussian	10K	6	320KB	Query Dataset
Gaussian	20K	7	640KB	Query Dataset
Gaussian	30K	8	960KB	Query Dataset
Gaussian	40K	9	1,3MB	Query Dataset
Gaussian	50K	10	1,6MB	Query Dataset

Table 3 lists all the generated datasets, that will be used in scaling experiments of our methods. For the reference dataset we created four datasets using the "Bit" distribution (Fig. 5), with file sizes ranging from 153MB to 611MB. The reference points cardinality ranges from 5M points to 20M points. For the query points dataset we created one "Uniform" dataset of 10 points and five "Gaussian" datasets ranging from 10K to 50K points.

All experiments were performed on a NVIDIA Jetson Nano. This device is running Ubuntu 18.04 and is equipped with a Quad-core ARM A57 cpu at 1.43 GHz and 4 GB 64-bit LPDDR4 of main memory. The operating system is installed on a 32GB microSD. The experiment datasets are stored on a USB-3 external 500GB Samsung 860 EVO SSD. Jetson nano's GPU microarchitecture is based on the MAXWELL model, which is the successor to the Kepler microarchitecture, and is armed with 128 CUDA cores.

We run experiments to compare the performance of multiple k-NN queries, regarding execution time as well as memory utilization. We tested our three algorithms (two presented in [6] and a new one, presented in the paper) in two variations each (depending on the list-buffer structure used for storing nearest neighbors of each query point). The list of our algorithms tested is as follows:

1. DBF [6], Disk Brute Force using KNN-DLB buffer.
2. DBF Heap [6], Disk Brute Force using max Heap buffer.
3. DPS [6], Plane Sweep using KNN-DLB buffer.
4. DPS Heap [6], Plane Sweep using max Heap buffer.
5. DSPP, Symmetric Progression Partitioning using KNN-DLB buffer.
6. DSPP Heap, Symmetric Progression Partitioning max Heap buffer.

In order to compare our methods' performance, we will compare them with existing in-memory ones. This is because, apart from the methods of [6], to the best of our knowledge there are not any other methods in the literature to address the k-NN query, using SSD-resident data.

We slightly altered the code of the following existing methods, in order to load data from disk:

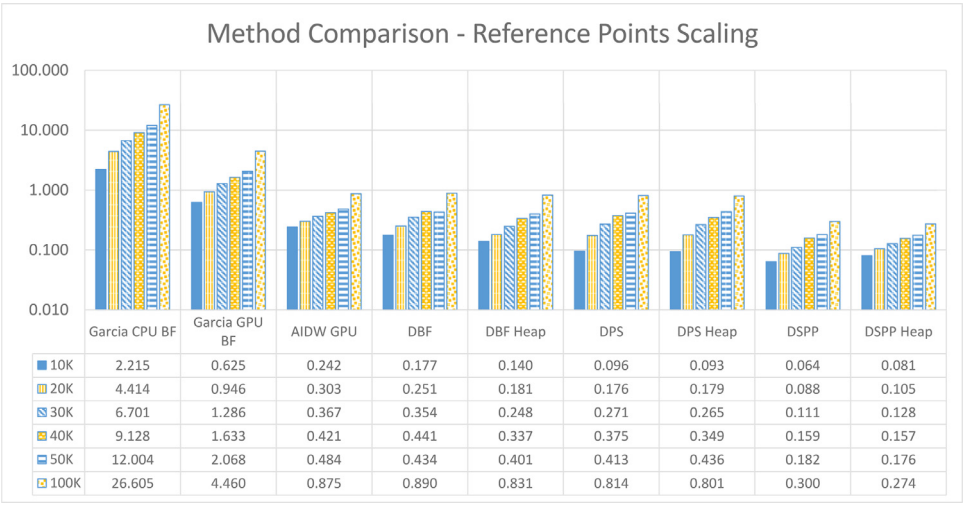
1. Garcia CPU BF [22], Brute Force algorithm using only the CPU.
2. Garcia GPU BF [22], Brute Force algorithm using the GPU.
3. AIDW [36], Fast k-NN search used for Adaptive Inverse Distance Weighting (AIDW) interpolation algorithm.

Garcia CPU BF was included only for showing the advantage of using a GPU against a CPU.

5.1. Comparison to existing methods

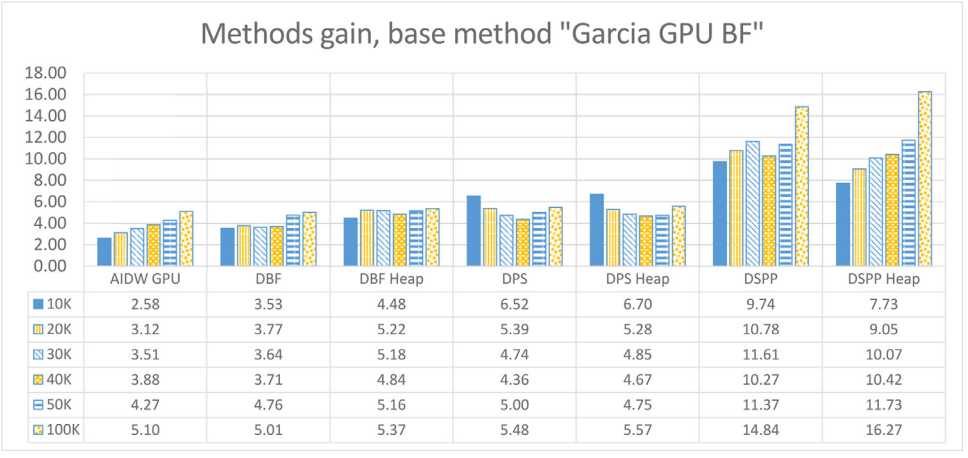
In our first series of experiments, we are comparing existing in-memory methods of other researchers (Garcia CPU BF, Garcia GPU BF and AIDW) against four existing of our own (DBF, DBF Heap, DPS, DPS Heap) and two new ones (DSPP and DSPP Heap). We aim to test all methods under the same conditions. Having this in mind, we altered the code of existing methods by other researchers in the data load part only. The code added to these methods is using exactly the same disk read calls that we are using in our methods. This way we are using the same file structure and exactly the same data

https://doi.org/10.1016/j.iot.2021.100428



	Garcia CPU BF	Garcia GPU BF	AIDW GPU	DBF	DBF Heap	DPS	DPS Heap	DSPP	DSPP Heap
10K	2.215	0.625	0.242	0.177	0.140	0.096	0.093	0.064	0.081
20K	4.414	0.946	0.303	0.251	0.181	0.176	0.179	0.088	0.105
30K	6.701	1.286	0.367	0.354	0.248	0.271	0.265	0.111	0.128
40K	9.128	1.633	0.421	0.441	0.337	0.375	0.349	0.159	0.157
50K	12.004	2.068	0.484	0.434	0.401	0.413	0.436	0.182	0.176
100K	26.605	4.460	0.875	0.890	0.831	0.814	0.801	0.300	0.274

Fig. 6. Experiment Comparison. All values are measured in seconds. The *Y* axis is in logarithmic scale.



	AIDW GPU	DBF	DBF Heap	DPS	DPS Heap	DSPP	DSPP Heap
10K	2.58	3.53	4.48	6.52	6.70	9.74	7.73
20K	3.12	3.77	5.22	5.39	5.28	10.78	9.05
30K	3.51	3.64	5.18	4.74	4.85	11.61	10.07
40K	3.88	3.71	4.84	4.36	4.67	10.27	10.42
50K	4.27	4.76	5.16	5.00	4.75	11.37	11.73
100K	5.10	5.01	5.37	5.48	5.57	14.84	16.27

Fig. 7. Experiment Comparison. All values are measured in seconds.

files for all methods. The data loading overhead of the existing methods of other researchers was measured and was found to be almost negligible when compared to the algorithm execution part. The total execution time for each of the existing methods of other researchers is calculated by adding the data load overhead to the algorithm execution time. Data loading is an integral part of our methods.

The existing methods of other researchers are designed for in-memory execution. This design radically affects the data volume we can experiment on. The maximum reference points volume we could test was 100K points, because volumes larger than 100K resulted in memory allocation errors. Under these restrictions, the comparison against existing methods of other researchers will be conducted for reference point volumes starting from 10K and scaling up to a maximum of 100K points. The maximum attainable query points volume is 2K. The maximum reference and query points volumes were experimentally found so that the same test was successfully executed by all methods. We used the “Bit” distribution synthetic datasets for the reference points and the “Gaussian” distribution for the query dataset.

In Fig. 6, we depict the results of the first series of experiments. We observe that in all reference point volumes the Garcia CPU BF method is the slowest one. This is to be expected, because this method is executed in the host CPU using only one thread. This results to higher execution times and this is also the main reason we are using a logarithmic *Y*-axis scale, just to produce an easily readable chart.

In the rest of this section we will continue by comparing the results of the GPU-based methods only. In Fig. 7 we depict the execution time gain of each such algorithm against one (denoted in each relevant figure) which is considered the comparison baseline (for each reference dataset volume, we divide the execution time of the baseline method (which is the slowest method) by the execution time of each of the other algorithms). Starting from the smallest reference points volume (10K) we observe that the fastest method is DSPP with 0.064 s, achieving the fastest execution time in our comparison experiment. DSPP heap follows with a slightly slower time of 0.081 s, DPS Heap with 0.093 s, DPS with 0.096 s, DBF Heap

https://doi.org/10.1016/j.iot.2021.100428

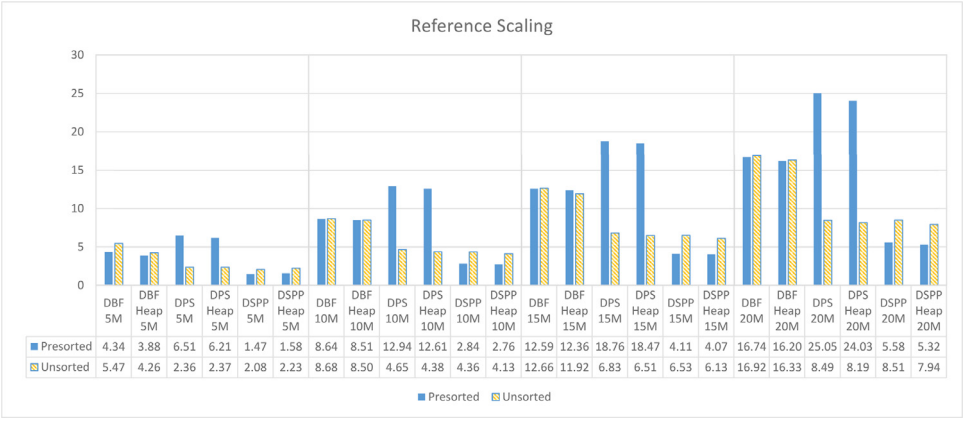


Fig. 8. Reference scaling experiment (*Y*-axis in seconds).

with 0.140 s, DBF with 0.177 s, AIDW with 0.242 and the slowest one is Garcia GPU BF with 0.625 s. As a result DSPP is about 9.7 times faster than the Garcia GPU BF method (baseline one).

When experimenting on larger reference point volumes, we observe analogous execution characteristics. DSPP Heap is the fastest method and DSPP follows closely. For the largest reference points experiment at 100K points, the execution times are 0.274 s for DSPP Heap, 0.300 s for DSPP, 0.801 s for DPS Heap, 0.814 s for DPS, 0.831 s for DBF Heap, 0875 s for AIDW GPU, 0.890 for DBF and 4.460 s for Garcia GPU BF. DSPP Heap is much faster than the other methods, resulting to a 16.27 gain (times faster) when compared to the Garcia GPU BF method.

DSPP is executing faster than the other methods, primarily because of the second level partitioning that is implemented in the device memory. By using this kind of partitioning, the query points are firstly compared to one partition at a time and in case the query point is within the desired range the process continues inside this partition and calculates all the reference point distances. On the contrary, in the other methods the query points distance is calculated directly upon reference points, resulting to slower performance.

5.2. Reference dataset scaling

In our second series of experiments, we used the “Bit” distribution synthetic datasets for the reference points. The size of the reference point dataset ranged from 5M points to 20M points. Furthermore, we used a small query dataset of 10 points, with “Uniform” distribution and a relatively small *k* value, 10, in order to focus only on the reference dataset scaling.

In Fig. 8, we can see the presorted dataset results in blue and the unsorted dataset results in stripped yellow. In the presorted experiment, for the 5M dataset the execution time is 4.34 s for DBF, 3.88 s for DBF Heap, 6.51 for DPS, 6.21 s for DPS Heap, 1.47 s for DSPP and 1.58 for DSPP Heap. The execution times scale analogously to the reference dataset size. As expected, we get the slowest execution times for the 20M dataset, 16.74 s for DBF, 16.20 for DBF Heap, 25.05 s for DPS, 24.03 s for DPS Heap, 5.58 s for DSPP and 5.32 for DSPP Heap.

In the unsorted dataset experiments, we observe that execution times are smaller than the presorted ones, for the Plane-sweep methods. The execution of DBF methods is about the same and for DSPP are slightly slower. For the 5M unsorted dataset, the execution time for DBF is 5.47 s, for DBF Heap is 4.26 s, for DPS 2.36 s, 2.37 s for DPS Heap, 2.08 from DSPP and 2.23 s for DSPP Heap. Once again, the execution times scale analogously to the reference dataset size. For the 20M unsorted dataset, we get 16.92 s for DBF, 16.33 s for DBF Heap, 8.49 s for DPS,8.19 s for DPS Heap, 8.51 s for DSPP and 7.94 for DSPP Heap.

The reference dataset scaling experiments reveal DPS methods performed better for the unsorted dataset. For the unsorted dataset, the Plane-sweep methods performed exceptionally better in unsorted than the presorted dataset. The Heap methods were slightly better, in most cases, than the KNN-DLB ones. The fastest methods overall are DSPP and DSPP Heap and both methods are 1.94 to 3.14 times faster (Fig. 9) than Brute-force one, in both dataset experiments.

5.3. Query dataset scaling

In our third set of experiments, we used 10K up to 50K query points with “Gaussian” distribution. For the reference points we used a 5M “Bit” distribution synthetic dataset. These experiments also used a relatively small *k* value of 10, in order to focus only on query dataset scaling.

In Fig. 10, we can see the presorted dataset results in blue and the unsorted dataset results in stripped yellow. In the presorted experiments, we notice that the execution time of the Brute-force algorithms is always larger than the other ones. Depending on the query dataset size, we observe that the execution time gradually increases analogously for every method. For the 10K dataset, the execution time is 110 s for DBF, 105 s for DBF Heap, 95 for DPS, 87 s for DPS Heap and 19 for DSPP

https://doi.org/10.1016/j.iot.2021.100428

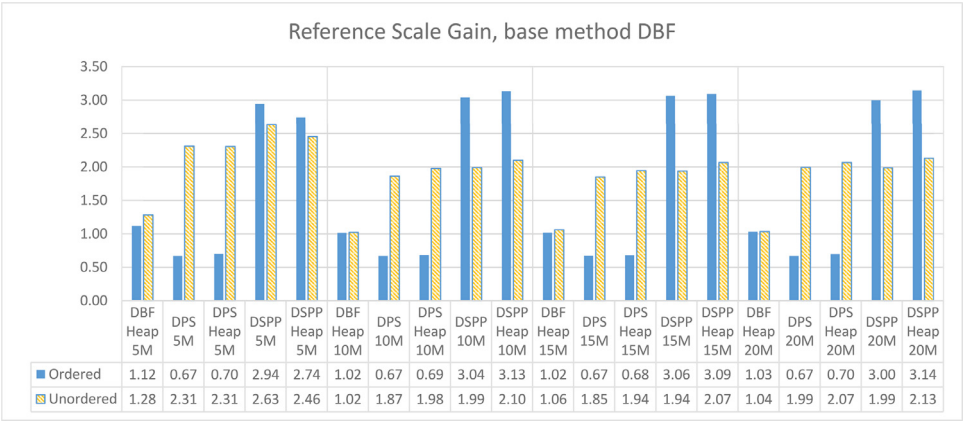


Fig. 9. Reference scaling experiment gain, base method DBF.

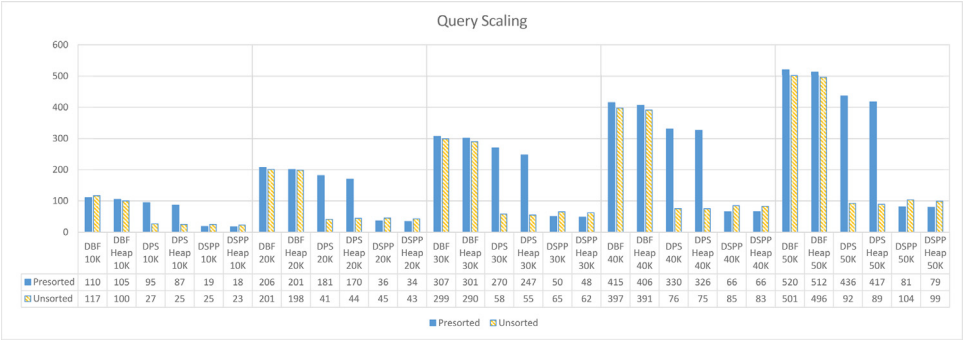


Fig. 10. Query scaling experiment (Y-axis in seconds).

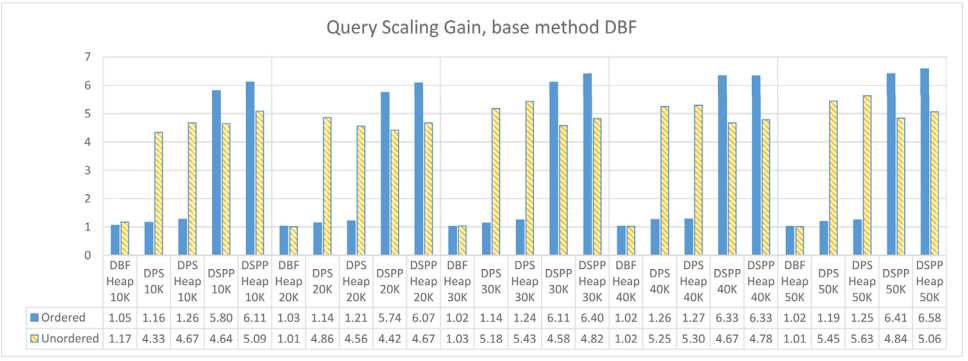


Fig. 11. Query scaling experiment gain, base method DBF.

and 18 for DSPP Heap. The slowest execution times were recorded for the 50K query dataset, 520 s for DBF, 512 s for DBF Heap, 436 s for DPS, 417 s for DPS Heap, and 81 s for both DSS and 79 for DSPP Heap.

In the unsorted experiments, we observe once more that all execution times are about the same of smaller, except for the Plane-sweep methods which was significantly faster. For the 10K unsorted query dataset, the execution time for DBF is 100 s, for DBF Heap is 117 s, for DPS 24 s, for DPS Heap 25 s, for DSPP 25 s and for DSPP Heap 26. Once again, the execution times scale analogously to the query dataset size. For the 50K unsorted query dataset, we get 496 s for DBF, 501 s for DBF Heap, 90 s for DPS, 94 s for DPS Heap, 104 s for DSPP and 106 s for DSPP Heap.

The results of the query dataset scaling experiments conform with the ones of the reference scaling experiments. For the unsorted dataset, the Plane-sweep methods performed exceptionally better in unsorted than the presorted dataset. The Heap methods were slightly better than the KNN-DLB ones. The fastest methods overall is DSPP and DSPP Heap and both methods are 5.5 to 6.6 times faster (Fig. 11) than the Brute-force one, in both dataset experiments

https://doi.org/10.1016/j.iot.2021.100428

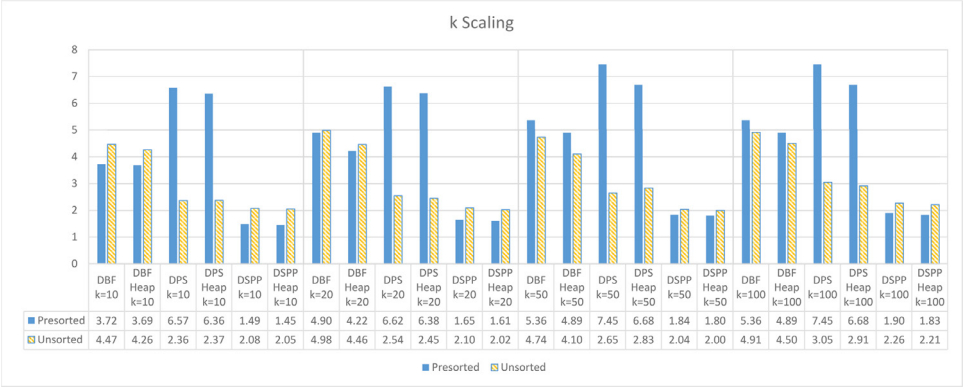


Fig. 12. *k* scaling experiment (*Y*-axis in seconds).

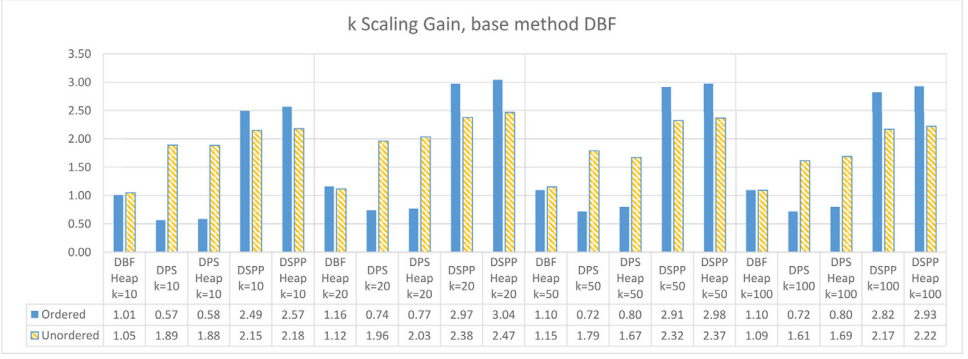


Fig. 13. *k* scaling experiment gain, base method DBF.

5.4. k *scaling*

The *k* scaling is our last set of experiments. In these tests we used *k* values of 10, 20, 50 and 100. For the reference points we used the 5M "Bit" distribution synthetic dataset and a small query group of 10 points, with "Uniform" distribution, in order to focus only on the *k* scaling.

In Fig. 12, we can see the presorted dataset results in blue and the unsorted dataset results in stripped yellow. In the presorted experiments, we notice that the execution time of the Brute-force algorithms is shorter than the Plane-sweep ones. DSPP methods are performing much better than the other two. Depending on the *k* value, we observe that seamlessly the execution time increases. For *k* equal to 10 the execution time is 3.72 s for DBF, 3.69 s for DBF Heap, 6.57 for DPS, 6.36 s for DPS Heap, 1.49 s for DSPP and 1.45 for DSPP Heap. The slowest execution times were recorded for *k* value of 100, 5.36 s for DBF, 4.89 s for DBF Heap, 7.45 s for DPS, 6.68 for DPS Heap, 1.90 s for DSPP and 1.83 for DSPP Heap.

In the unsorted experiments, we observe once more that the execution of DPS is faster than the presorted ones. The Brute-force methods are slightly faster for the unsorted dataset than for the presorted one. The DSPP methods are quicker than the other methods. For *k* equal to 10 for the unsorted query dataset, the execution time for DBF is 4.47 s, for DBF Heap is 4.26 s, for DPS 2.36 s, for DPS Heap 2.08 s, for DSPP 1.49 and for DSPP Heap 1.45 s. Once again, the execution times scale analogously to the *k* value. For $k = 100$, we get 4.91 s for DBF, 4.50 s for DBF Heap, 3.05 s for DPS, 2.91 for DPS Heap, 2.26 for DSPP and 2.21 for DSPP Heap.

The results of the *k* scaling experiments also conform with the results of the previous experiments. The Plane-sweep methods performed once again exceptionally better in the unsorted dataset than in the presorted dataset. Furthermore, the DSPP methods were overall quicker, performing more than 2 times faster than the Brute-force one (Fig. 13), in the all dataset experiments. The Heap methods were slightly better than the KNN-DLB ones.

Although, the two *k*-NN list buffer methods were shown equal, for even larger *k* values than the ones studied in this paper, the *k* max-Heap list buffer is expected to outperform the KNN-DLB one.

5.5. *Interpretation of results*

As noted in [6], exploring why the application of the Plane-sweep algorithms on unsorted reference data is significantly more efficient, we observed that, when the reference dataset is presorted, each partition contains points that fall within a

https://doi.org/10.1016/j.iot.2021.100428

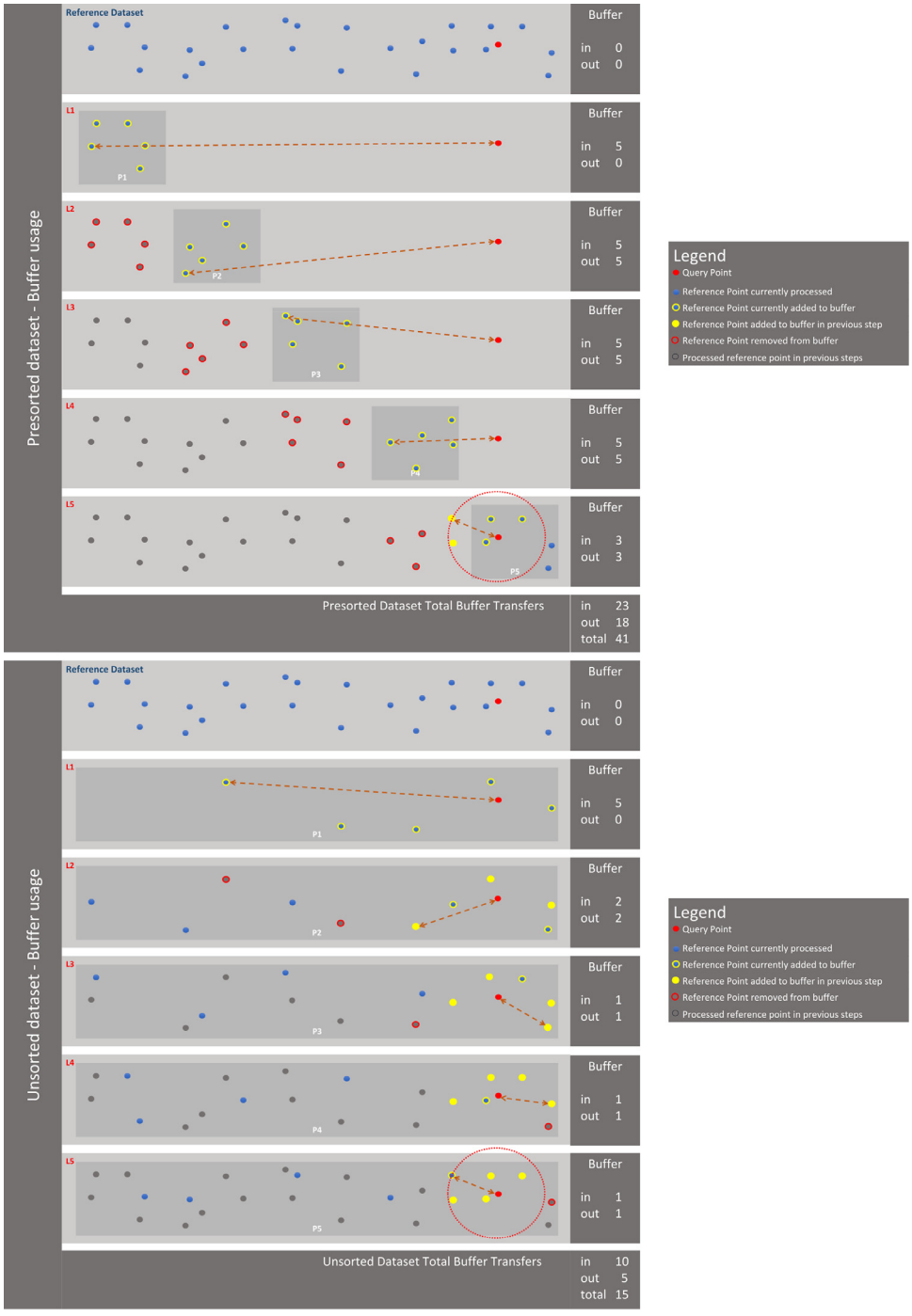


Fig. 14. Presorted versus unsorted reference dataset buffer update.

limited x-range and in case the query point under examination is on the right side of this partition regarding x-dimension, most of the reference points of this partition will likely replace points already included in the current set of k NNs for this query point (Fig. 14 top), since partitions are loaded from left to right and previous partitions examined were less x-close to this query point. However, when the reference dataset is unsorted, each partition contains points that cover a wide x-range and it is likely that many of the reference points of this partition will be rejected by comparing their x-distance to the distance of the kth NN found so far (Fig. 14 bottom).

https://doi.org/10.1016/j.iot.2021.100428

The best performance was achieved with the DSPP methods. The partition reading from the SSD reference data was equivalent for every method. So, the computational acceleration of DSPP was due to the second level partitioning. All the other methods are calculating distances of points of the reference partition. DSPP incorporates a second level partitioning that further reduces the distances of points that need to be calculated. In this context, DSPP firstly selects the closest to the query point partition and then executes the distance calculation. If needed it advances to the next closest partition, having in mind that its distance does not exceed the current maximum k-NN distance. In a way, DSPP actively regulates the reference points search scope.

6. Conclusions and future plans

In this paper, we presented a new partitioning algorithm for processing the k-NN query for big reference data which exploits the parallelism of GPUs and the speed of SSDs, as secondary memory storage. We implemented this algorithm in an edge-computing device, showing that answering this query is feasible and efficient using such a device, which has limited power needs and small size, being versatile for on-site computing applications. Using synthetic datasets, through an extensive experimental performance comparison of the new algorithm against (in-memory) existing ones by other researchers and two algorithms (working on SSD-resident data) recently proposed by us it was shown that the new algorithm excels in all the conducted experiments and outperforms its rivals. This is due to the two-level partitioning employed by the new algorithm, since this approach leads to a reduction of the in-memory reference points distance calculations. We also proposed an architecture of a distributed environment embedding such edge-computing devices where large-scale processing of the k-NN query through the proposed algorithm can be accomplished. This architecture is suitable for processing of a wide range of queries on big data, where most of the processing takes place at the network edges.

Future work plans include:

- Developing algorithms for other demanding queries and/or queries also addressing the temporal dimension of data for the same computing architecture.
- Examining the utilization of indexes for speeding up processing even further.
- Introducing even more advanced partitioning in DSPP algorithm, to further improve its performance.
- Developing the appropriate APIs for implementing the architecture presented in Section 3.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

Work of M. Vassilakopoulos and A. Corral funded by the MINECO research project [TIN2017-83964-R].

References

[1] G. Barlas, Multicore and GPU Programming: An Integrated Approach, first ed., Morgan Kaufmann, 2014.
[2] S.B. Imandoust, M. Bolandraftar, Application of k-nearest neighbor (KNN) approach for predicting economic events theoretical background, Int. J. Eng. Res.Appl. 3 (5) (2013) 605–610.
[3] P. Velentzas, M. Vassilakopoulos, A. Corral, A partitioning GPU-based algorithm for processing the k nearest-neighbor query, in: MEDES Conference, 2020, pp. 2–9.
[4] NVIDIA, Nvidia CUDA runtime API, 2020, https://docs.nvidia.com/cuda/cuda-runtime-api/index.html.
[5] P. Velentzas, P. Moutafis, G. Mavrommatis, An improved GPU-based algorithmfor processing the k nearest neighbor query, in: PCI Conference, 2020, pp. 372–375.
[6] P. Velentzas, M. Vassilakopoulos, A. Corral, GPU-based algorithms for processing the k nearest-neighbor query on disk-resident data, in: MEDI Conference, 2021, pp. 264–278.
[7] S. Mittal, A survey on optimized implementation of deep learning models on the NVIDIA Jetson platform, J. Syst. Archit. 97 (2019) 428–442.
[8] D.P. Singh, I. Joshi, J. Choudhary, Survey of GPU based sorting algorithms, Int. J. Parallel Program. 46 (6) (2018) 1017–1034.
[9] V. Garcia, E. Debreuve, M. Barlaud, Fast k nearest neighbor search using GPU, in: CVPR Workshops, 2008, pp. 1–6.
[10] Q. Kuang, L. Zhao, A practical GPU based KNN algorithm, in: SCSCT Conference, 2009, pp. 151–155.
[11] S. Liang, C. Wang, Y. Liu, L. Jian, CUKNN: a parallel implementation of k-nearest neighbor on CUDA-enabled GPU, in: YC-ICT Conference, 2009, pp. 415–418.
[12] V. Garcia, E. Debreuve, F. Nielsen, M. Barlaud, K-nearest neighbor search: fast GPU-based implementations and application to high-dimensional feature matching, in: ICIP Conference, 2010, pp. 3757–3760.
[13] R.J. Barrientos, J.I. Gómez, C. Tenllado, M. Prieto-Matías, M. Marín, kNN query processing in metric spaces using GPUs, in: Euro-Par Conference, 2011, pp. 380–392.
[14] N. Sismanis, N. Pitsianis, X. Sun, Parallel search of k-nearest neighbors with synchronous operations, in: HPEC Conference, 2012, pp. 1–6.
[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) 1–13.
[16] K. Kato, T. Hosino, Multi-GPU algorithm for k-nearest neighbor problem, Concurrency Comput. Pract.Exp. 24 (1) (2012) 45–53.
[17] I. Komarov, A. Dashti, R.M. D'Souza, Fast k-NNG construction with GPU-based quick multi-select, PLoS ONE 9 (5) (2014) 1–9.
[18] S. Li, N. Amenta, Brute-force k-nearest neighbors search on the GPU, in: SISAP Conference, 2015, pp. 259–270.
[19] P.D. Gutiérrez, M. Lastra, J. Bacardit, J.M. Benítez, F. Herrera, GPU-SME-kNN: scalable and memory efficient kNN and lazy learning using GPUs, Inf. Sci. 373 (2016) 165–182.

https://doi.org/10.1016/j.iot.2021.100428

[20] R.J. Barrientos, F. Millaguir, J.L. Sánchez, E. Arias, GPU-based exhaustive algorithms processing kNN queries, J. Supercomput. 73 (10) (2017) 4611–4634.
[21] J.A. Riquelme, R.J. Barrientos, R. Hernández-García, C.A. Navarro, An exhaustive algorithm based on GPU to process a kNN query, in: SCCC Conference, 2020, pp. 1–8.
[22] V. Garcia,;. Debreuve, M. Barlaud, Fast k nearest neighbor search using GPU, 2018, http://vincentfpgarcia.github.io/kNN-CUDA/.
[23] P. Velentzas, M. Vassilakopoulos, A. Corral, In-memory k nearest neighbor GPU-based query processing, in: GISTAM Conference, 2020, pp. 310–317.
[24] J.L. Bentley, Multidimensional binary search trees used for associative searching, Commun. ACM 18 (9) (1975) 509–517.
[25] K. Zhou, Q. Hou, R. Wang, B. Guo, Real-time KD-tree construction on graphics hardware, ACM Trans. Graph. 27 (5) (2008) 126.
[26] F. Gieseke, J. Heinermann, C.E. Oancea, C. Igel, Buffer k-d trees: processing massive nearest neighbor queries on GPUs, in: ICML Conference, 2014, pp. 172–180.
[27] P. Indyk, R. Motwani, Approximate nearest neighbors: towards removing the curse of dimensionality, in: STOC Conference, 1998, pp. 604–613.
[28] J. Pan, C. Lauterbach, D. Manocha, Efficient nearest-neighbor computation for GPU-based motion planning, in: IROS Conference, 2010, pp. 2243–2248.
[29] J. Pan, D. Manocha, Fast GPU-based locality sensitive hashing for k-nearest neighbor computation, in: ACM-GIS Conference, 2011, pp. 211–220.
[30] P. Wieschollek, O. Wang, A. Sorkine-Hornung, H.P.A. Lensch, Efficient large-scale approximate nearest neighbor search on the GPU, CoRR (2017) abs/1702.05911.
[31] A. Guttman, R-trees: a dynamic index structure for spatial searching, in: SIGMOD Conference, 1984, pp. 47–57.
[32] S. You, J. Zhang, L. Gruenwald, Parallel spatial query processing on GPUs using r-trees, in: BigSpatial@SIGSPATIAL Workshop, 2013, pp. 23–31.
[33] M. Nam, J. Kim, B. Nam, Parallel tree traversal for nearest neighbor query on the GPU, in: ICPP Conference, 2016, pp. 113–122.
[34] D.A. White, R.C. Jain, Similarity indexing with the SS-tree, in: ICDE Conference, 1996, pp. 516–523.
[35] P.J.S. Leite, J.a.M.X.N. Teixeira, T.S.M.C. de Farias, B. Reis, V. Teichrieb, J. Kelner, Nearest neighbor searches on the GPU - a massively parallel approach for dynamic point clouds, Int. J. Parallel Program. 40 (3) (2012) 313–330.
[36] G. Mei, N. Xu, L. Xu, Improving GPU-accelerated adaptive IDW interpolation algorithm using fast kNN search, Springerplus 5 (1) (2016) 1389.
[37] A. Aji, H. Vo, F. Wang, Effective spatial data partitioning for scalable query processing, CoRR (2015) 1–12 abs/1509.00910.
[38] S. Mittal, J.S. Vetter, A survey of software techniques for using non-volatile memories for storage and main memory systems, IEEE Trans. Parallel Distrib. Syst. 27 (5) (2016) 1537–1550.
[39] H. Roh, S. Park, S. Kim, M. Shin, S.-W. Lee, B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives, Proc. VLDB Endow. 5 (4) (2011) 286–297.
[40] J.M. Cecilia, J.-C. Cano, J. Morales-García, A. Llanes, B. Imbernón, Evaluation of clustering algorithms on GPU-based edge computing platforms, Sensors 20 (21) (2020) 6335.
[41] P. Kang, S. Lim, A taste of scientific computing on the GPU-accelerated edge device, IEEE Access 8 (2020) 208337–208347.
[42] S. Lim, P. Kang, Implementing scientific simulations on GPU-accelerated edge devices, in: ICOIN Conference, 2020, pp. 756–760.
[43] J. Jo, S. Jeong, P. Kang, Benchmarking GPU-accelerated edge devices, in: BigComp Conference, 2020, pp. 117–120.
[44] F.P. Preparata, M.I. Shamos, Computational Geometry - An Introduction, Texts and Monographs in Computer Science, Springer, 1985.
[45] K.H. Hinrichs, J. Nievergelt, P. Schorn, Plane-sweep solves the closest pair problem elegantly, Inf. Process. Lett. 26 (5) (1988) 255–261.
[46] P. Katiyar, T. Vu, A. Eldawy, S. Migliorini, A. Belussi, SpiderWeb: a spatial data generator on the web, in: SIGSPATIAL Conference, 2020, pp. 465–468.
[47] T. Vu, S. Migliorini, A. Eldawy, A. Belussi, Spatial data generators, 1st ACM SIGSPATIAL International Workshop on Spatial Gems (SpatialGems), ACM, 2019. https://www.spatialgems.net/spatial-gems-collection