UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Parallel Query Processing on Big Spatial Data

A dissertation submitted in partial fulfillment of the requirements

for the degree of Doctor of Philosophy

Polychronis Velentzas

November 2022

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Parallel Query Processing on Big Spatial Data

A dissertation submitted in partial fulfillment of the requirements

for the degree of Doctor of Philosophy

Polychronis Velentzas

November 2022

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Παράλληλη Επεξεργασία Ερωτημάτων σε Μεγάλου Όγκου Χωρικά Δεδομένα

Διατριβή η οποία υποβλήθηκε για τη μερική εκπλήρωση

των υποχρεώσεων απόκτησης του Διδακτορικού Διπλώματος

Πολυχρόνης Βελέντζας

Νοέμβριος 2022

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Parallel Query Processing on Big Spatial Data

PhD Dissertation

Polychronis Velentzas

Advisory committee

Examination committee

November 2022

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

Being fully aware of the implications of copyright laws, I expressly state that this PH.D. dissertation, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism.

The declarant

Polychronis Velentzas

Acknowledgments

I would like to acknowledge and thank my advisory committee, Professors Michael Vassilakopoulos, Emmanouil Vavalis, and Vassilios Verykios, for their invaluable support and guidance. I would also like to thank Professors Antonio Corral and Christos Antonopoulos for providing feedback and advice. I owe a debt of gratitude to my family for their unwavering support and encouragement.

PhD Dissertation

Parallel Query Processing on Big Spatial Data

Polychronis Velentzas

Abstract

Nowadays we are storing overwhelming volumes of Spatial data. The continuously growing need for processing such big data has led us create distributed or parallel frameworks, to address the emerging computational overhead. Such data (often termed as geospatial data), are used in many geographically enabled applications.

To exploit these data, efficient processing of spatial queries is of great importance due to the wide area of applications that may address such queries. The most common spatial queries where points are involved are point-location, window, distance-range and k nearest-neighbor queries (k-NNQs). At a higher level, such queries have been used as the basis of many complex operations in advanced applications, for example, geographical information systems (GIS), location-based systems (LBS), geometric databases, CAD, etc.

The spatial queries processing performance is usually affected by two major factors. The computational and the storage factor. The first one can be addressed by many architectures, for example single core (sequential execution), multi-core (parallel execution), multi-node or even GPU processing models. The second factor is also of great importance and affects the retrieval or storage of information. The advent of non-volatile memories (NVM) has enabled a brand-new class of storage devices with exciting features that will prevail in the storage market in the near future. Their high read and write speeds, small size, low power consumption and shock resistance are some of the reasons that made them storage medium of choice. NAND flash is undoubtedly the most popular NVM today. Storage devices based on NAND Flash are found both in consumer devices and enterprise data-centers.The increasing needs for efficient storage drove the emergence of Solid-State Drives (SSDs).

Aside from the design and development of the above spatial queries, we furthermore tested their efficiency in Edge Computers. Edge computing is a distributed computing architecture where computation and data storage is as close to the source of data as possible. We also designed an Edge computing and IoT distributed architecture, where we described thoroughly the way we can exploit the dynamics of our techniques.

In this thesis we focus on the design and implementation of spatial queries that take advantage of multi-core CPU, GPU and SSD for storage. We employ techniques for faster and fewer computations, computation reductions as well as data processing reductions, using spatial locality and data distribution, on either synthetic or real data. Based on these techniques

- we develop efficient spatial query algorithms using multi-core CPU
- we develop efficient spatial query algorithms using GPU
- we exploit the massive I/O advantages of SSDs.
- we apply these queries to Edge Computers
- we perform extensive experimental tests to compare our algorithms to other existing ones using synthetic and real spatial data.

Keywords

multi-core; k Nearest-Neighbor Query; GPU; SSD; Spatial-queries algorithms; Plane-sweep; Max-Heap; Parallel computing; Edge computing; IoT

Διδακτορική Διατριβή

Παράλληλη Επεξεργασία Ερωτημάτων

σε Μεγάλου Όγκου Χωρικά Δεδομένα

Πολυχρόνης Βελέντζας

Περίληψη

Σήμερα αποθηκεύουμε πολύ μεγάλο όγκο χωρικών δεδομένων. Η συνεχώς αυξανόμενη ανάγκη για επεξεργασία τέτοιων μεγάλων δεδομένων μας οδήγησε να δημιουργήσουμε κατανεμημένα ή παράλληλα συστήματα, για να αντιμετωπίσουμε τα αυξανόμενα υπολογιστικά κόστη. Τέτοια δεδομένα (συχνά ονομάζονται γεωχωρικά δεδομένα), χρησιμοποιούνται σε πολλές γεωγραφικές εφαρμογές.

Για την εκμετάλλευση αυτών των δεδομένων, η αποτελεσματική επεξεργασία χωρικών ερωτημάτων είναι μεγάλης σημασίας λόγω της ευρείας γκάμας εφαρμογών όπου μπορούν να χρησιμοποιήσουν αυτά τα ερωτήματα. Τα πιο συνηθισμένα χωρικά ερωτήματα στα οποία εμπλέκονται χωρικά σημεία είναι τα ερωτήματα θέσης σημείου, παραθύρου, απόστασης εμβέλειας και k πλησιέστερου γείτονα (k-NNQs). Σε υψηλότερο επίπεδο, τέτοια ερωτήματα έχουν χρησιμοποιηθεί ως βάση πολλών πολύπλοκων λειτουργιών σε προηγμένες εφαρμογές, για παράδειγμα, συστήματα γεωγραφικών πληροφοριών (GIS), συστήματα που βασίζονται σε τοποθεσία (LBS), γεωγραφικές βάσεις δεδομένων, CAD κ.λπ.

Η απόδοση της επεξεργασίας των χωρικών ερωτημάτων συνήθως επηρεάζεται από δύο βασικούς παράγοντες. Ο υπολογιστικός παράγοντας και ο παράγοντας αποθήκευσης. Το πρώτο μπορεί να αντιμετωπιστεί από πολλές αρχιτεκτονικές, για παράδειγμα, μονού πύρηνα (διαδοχική-σειριακή εκτέλεση), πολλαπλών πυρήνων (παράλληλη εκτέλεση), πολλαπλών κόμβων ή ακόμη και μοντέλα επεξεργασίας GPU. Ο δεύτερος παράγοντας έχει επίσης μεγάλη σημασία και επηρεάζει την ανάκτηση ή αποθήκευση πληροφοριών. Η έλευση της non-volative μνήμης (NVM) επέτρεψε μια ολοκαίνουργια κατηγορία συσκευών αποθήκευσης με συναρπαστικά χαρακτηριστικά που θα επικρατήσουν στην αγορά αποθήκευσης στο εγγύς μέλλον. Οι υψηλές ταχύτητες ανάγνωσης και εγγραφής, το μικρό μέγεθος, η χαμηλή κατανάλωση ενέργειας και η αντοχή τους σε κραδασμούς είναι μερικοί από τους λόγους που τα έκαναν δημοφιλή ως μέσο αποθήκευσης. Το NAND flash είναι αναμφίβολα το πιο δημοφιλές NVM σήμερα. Οι συσκευές αποθήκευσης που βασίζονται στο NAND Flash βρίσκονται τόσο σε καταναλωτικές συσκευές όσο και σε εταιρικά κέντρα δεδομένων. Οι αυξανόμενες ανάγκες για αποτελεσματική αποθήκευση οδήγησαν στην εμφάνιση μονάδων δίσκου στερεάς κατάστασης (SSD).

Εκτός από το σχεδιασμό και την ανάπτυξη των παραπάνω χωρικών ερωτημάτων, δοκιμάσαμε επιπλέον την αποτελεσματικότητά τους σε Edge Computers. Το Edge computing είναι μια κατανεμημένη υπολογιστική αρχιτεκτονική, όπου ο υπολογισμός και η αποθήκευση δεδομένων είναι όσο το δυνατόν πιο κοντά στην πηγή των δεδομένων. Σχεδιάσαμε επίσης μια κατανεμημένη αρχιτεκτονική υπολογιστών Edge και IoT, όπου περιγράψαμε διεξοδικά τον τρόπο με τον οποίο μπορούμε να εκμεταλλευτούμε τη δυναμική των τεχνικών μας.

Σε αυτή τη διατριβή εστιάζουμε στον σχεδιασμό και την υλοποίηση χωρικών ερωτημάτων που εκμεταλλεύονται επεξεργαστές πολλαπλών πυρήνων, GPU και SSD για αποθήκευση. Χρησιμοποιούμε τεχνικές για ταχύτερους και λιγότερους υπολογισμούς, μειώσεις υπολογισμών καθώς και μειώσεις επεξεργασίας δεδομένων, χρησιμοποιώντας χωρική εγγύτητα, διανομή δεδομένων, σε συνθετικά και σε πραγματικά δεδομένα. Με βάση αυτές τις τεχνικές

- αναπτύσσουμε αποτελεσματικούς αλγόριθμους χωρικών ερωτημάτων χρησιμοποιώντας CPU πολλαπλών πυρήνων
- αναπτύσσουμε αποτελεσματικούς αλγόριθμους χωρικών ερωτημάτων χρησιμοποιώντας GPU
- εκμεταλλευόμαστε τα τεράστια πλεονεκτήματα I/O των SSD.
- εφαρμόζουμε αυτά τα ερωτήματα στους Edge Computers
- διενεργούμε εκτενείς πειραματικές δοκιμές για να συγκρίνουμε τους αλγόριθμούς μας με άλλους υπάρχοντες χρησιμοποιώντας συνθετικά και πραγματικά χωρικά δεδομένα.

Λέξεις Κλειδιά

πολλαπλοί πυρήνες; k Κοντινότεροι-Γείτονες; GPU; SSD; Αλγόριθμοι χωρικών ερωτημάτων; Plane-sweep; Max-Heap; Παράλληλος προγραμματισμός; Edge computing; IoT

Table of contents

List of figures

List of tables

Abbreviations

GPU	Graphics processing unit
CUDA	Compute Unified Device Architecture
MPI	Message Passing Interface
SIMD	Single Instructions Multiple Data
SSD	Solid State Drive
GIS	Geographical Information Systems
LBS	Location-Based Systems
CAD	Computer-aided design
NVM	Non-Volative Memory
k-NN	k Nearest Neighbor
TBF	Thrust Brute Force
TDS	Thrust Distance Refinement
PS	Plane Sweep
SPP	Symmetric Progression Partitioning

Chapter 1

Introduction

Every year, the volume of the produced Spatial and Temporal data is continuously growing. In order to process and analyze such big data we need more powerfull devices, smarter and more efficient algorithms. This emerging computational overhead can be addressed by architectures based on distributed or parallel frameworks. Such data (often termed as geospatial or spatio-temporal), are used in many geographically enabled applications.

There are many spatial queries that we need in our every day life applications. The most common spatial queries, where points are involved, are point-location, window, distance-range and k nearest-neighbor queries (PLQs, WQs, DRQs and k-NNQs, respectively, in the sequel). At a higher level, such queries have been used as the basis of many complex operations in advanced applications, for example, geographical information systems (GIS), location-based systems (LBS), geometric databases, CAD, etc.

Another emerging technology that can benefit from such queries is Edge Computing. Edge computing is reshaping IT and business computing. Employing spatial queries in the Edge is a crucial success factor for fast and efficient query results, because Edge computers are closely located to data sources.

The performance of these geographic applications is usually affected by two major factors. The computational and the storage factor. The first one can be addressed by many architectures, for example single core (sequential execution), multi-core (parallel execution), multi-node or even GPU processing models. The second factor is also of great importance and affects the retrieval or storage of information. The arrival of non-volatile memories (NVM) has enabled a brand-new class of storage devices with exciting features that will prevail in the storage market in the near future. Their superior features, such as high read and write speeds, small size, low power consumption and shock resistance are some of the reasons that made them the preferable storage medium of choice. NAND flash is undoubtedly the most popular NVM today. Storage devices based on NAND Flash are found both in consumer devices and enterprise data-centers.The increasing needs for efficient storage drove the emergence of Solid-State Drives (SSDs).

In order to accelerate the operational speed of demanding spatial and spatio-temporal applications we can exploit many techniques and follow best practices. First and foremost would be to write better code, use better libraries or use better algorithms. Next would be the use of compiler level optimizations. Then the computations could be executed on faster hardware but there are physical limits to how fast a single processor core can be manufactured. With multicore processors, parallelization techniques could be implemented to speedup computation. This can usually be done with concurrency and threads. Again there are physical limits to the number of cores that can be added into a single processor. So, multiple processor and something like the Message Passing Interface (MPI) to manage communication among different processors could be the next step.

Furthermore, accelerator hardware like many cores GPU can be used to offload some of the computation. GPUs are extremely useful and efficient in doing Single Instructions Multiple Data (SIMD) computations but have some data transfer overheads. GPU coding also requires rewriting the kernels in CUDA so they have coding effort overheads too. In the CPUs, we can further use vectorization using intrinsics to vectorize the code so that the operation run concurrently. Also, focus on the memory hierarchy and cache organisation can lead to developing algorithms that are either cache aware or oblivious and reduce the memory movement overheads. Also, in a distributed setup, communication reducing or communication avoiding can reduce the communication costs which are usually a big part of distributed computing. Proper load balancing among the nodes can also lead to more efficient computation. Targeting memory IO patterns based on the existing hardware or file systems and avoiding writes to disk as much as possible because writes are far more costlier than reads, can also lead to better total computation time on a distributed system.

1.1 Thesis contribution

We are motivated by the vast amount of spatial data produced, their exploitation in numerous modern applications and the need to process these data in multi-core parallel environments with efficient algorithms, in this thesis, we develop such algorithms and study their performance.

Since, the possible research field directions within this discipline is immense, we focus on the above four demanding queries, namely point-location, window, distance-range and k nearest-neighbor queries (PLQs, WQs, DRQs and k-NNQs, respectively, in the sequel). Moreover, since an algorithm for a parallel system relies on the characteristics of the system and the possible system choices are numerous, we develop algorithms for some of the most popular such systems. In this thesis, we focus on point data and employing of techniques for faster and fewer computations, pruning of unnecessary computations, taking advantage of spatial locality and distribution of data, parallelism using OpenMP or CUDA library, optimizing the amount of computational load among threads, and using indexes such as xBR+tree for even higher execution time gain

- we develop the first Batch Point-Location Query algorithm (*BPLQ*) using xBR$^+$-trees in SSDs, taking advantage of multi-core CPUs .
- we develop the first Batch Window Query algorithm (*BWQ*) using xBR$^+$-trees in SSDs, taking advantage of multi-core CPUs .
- we develop the first Batch Distance-Range Queries (*BDRQ*) using xBR$^+$-trees in SSDs, taking advantage of multi-core CPUs .
- we develop novel in-memory GPU based algorithms for the k-NN query
- we develop novel data partitioning GPU based algorithms for the k-NN query using GPU and SSD
- we apply these queries to Edge Computers
- for each of the above queries, we perform extensive experimental tests to derive the best parameter settings for each algorithm and to compare the efficiency of the several alternative algorithms we developed and ones appearing in the literature (for the cases where such algorithms already existed).

Our first multi-threaded implementations successfully exploited the dynamics derived from the use of xBR$^+$-trees in SSDs. All of our queries run faster than previous implementations. OpenMP was a key "ingredient" to accomplish such task. We seamlessly integrated our existing single-thread algorithms to out new multi-threaded ones.

The next step was to develop spatial queries on CUDA devices. Although, CUDA devices are not designed specifically for spatial data, the algorithms we developed make these systems suitable for processing the spatial queries we study. Initially we used the well known CUDA Thrust library for developing k-NN query. Using Thrust we created our two first in-memory algorithms the Trust Brute Force (*TBF*) and the Thrust Distance Refinement (*TDS*) k-NN query. *TDS* is excellent for small volumes of query datasets and greatly outperforms existing methods.

Furthermore, we extended our algorithms by developing an in-memory Plane-Sweep (*PS*) k-NN query and the new Symmetric Progression Partitioning (*SPP*) k-NN query, which partitions reference data in the device memory.

Our next step was to develop a k-NN query for big data reference datasets. While keeping the query dataset in-memory we partitioned the reference dataset. This resulted to the development of the algorithms of our disk based algorithm implementations Disk Plane Sweep (*DPS*) and Disk Symmetric Progression Partitioning (*DSPP*) k-NN query.

Lastly, me further enhanced our algorithms to also support big query data. At this step we also exploited a significant CUDA feature. CUDA streams, which aims to hide the latency of memory copy and kernel launch from different independent operations [9], are widely used in computational tasks to increase performance [10]. This resulted to our most advanced algorithm Improved Disk Symmetric Progression Partitioning with pinned memory (*DSPP+P*).

1.2 Thesis organization

The rest of thesis is organized as follows. In Chapter 2, we present the parallel and distributed spatial query processing. In Chapter 3 we present Single Dataset Spatial Query Processing with CPU and SSD. In Chapter 4 we present the k-NN query processing with GPU and RAM. In Chapter 5 we present the k-NN query processing with GPU and SSD. In Chapter 6 we present the k-NN query processing with IoT Edge Devices and SSD. In Chapter 7 we present the k-NN query processing with GPU, SSD and full dataset partitioning. In Chapter

8, we present our conclusions and future directions.

At this point we would like to note that the state of the art will be embedded in each chapter. This way, we believe will assist the reader to easily become acquainted with each chapters' context and correlate it with our novel methods and implementations.

Chapter 2

Advanced Spatial query Processing

In this chapter we will begin by documenting in detail the available Big Spatial and Spatio-Temporal Data Analytics Systems. We are living in the era of Big Data, and Spatial and Spatio-temporal Data are not an exception. These systems are rather new but they have emerged quickly and are vastly used for the management of big spatial and spatio-temporal data. We will describe their usage, common spatial and spatio-temporal datatypes and their architecture. Later in this chapter we will focus on CPU and GPU Data Processing (since nowadays centralized systems can handle quite big data), and we will depict each architectures' advantages and disadvantages.

2.1 Parallel and distributed spatial processing

This section provides a comparative overview of Big Spatial and Spatio-Temporal Data Analytics Systems based on a set of characteristics (data types, indexing, partitioning techniques, distributed processing, query Language, visualization and case-studies of applications). We will present selected systems (the most promising and/or most popular ones), considering their acceptance in the research and advanced applications communities. More specifically, we will present two systems handling spatial data only (SpatialHadoop and GeoSpark) and two systems able to handle spatio-temporal data, too (ST-Hadoop and STARK) and compare their characteristics and capabilities. Moreover, we will also present in brief other recent / emerging spatial and spatio-temporal analytics systems with interesting characteristics. The subsection closes with our conclusions arising from our investigation of the rather new, though quite large world of ecosystems supporting management of big spatial and spatio-temporal data.

2.1.1 Parallel and distributed architectures

Data mining and analysis of big data is a non-trivial task. Often, it is performed in a distributed infrastructure of multiple compute network-interconnected (through the cluster network) nodes, each of which may be equipped with multiple CPUs or GPUs (Fig.2.1).

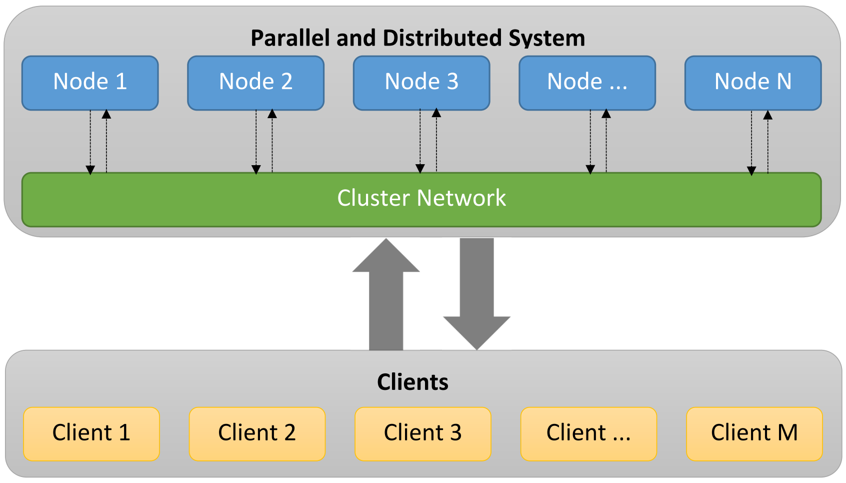


Figure 2.1: Parallel and distributed system architecture.

This kind of architectures bring a number of challenges. First of all, the resources must be effectively used. For example, one must avoid delays of CPU/GPU resources due to working data transfer over network. Second, all the available resources (CPU/GPU, storage and network) should be typically shared among different users or processes to reduce costs and increase interoperability. In order to address these challenges several architectures and frameworks have risen. In this section, we will describe the two most popular of them, Apache Hadoop and Apache Spark. Both of them are open source and widely used.

2.1.1.1 Apache Hadoop

Hadoop is a shared-nothing framework, meaning that the input data is partitioned and distributed to all computing nodes, which perform calculations on their local data only. Hadoop is a two-stage disk-based MapReduce computation engine, not well suited to repetitive processing tasks.

MapReduce [11, 12] is a programming model for distributed computations on very large amounts of data and a framework for large-scale data processing on clusters built from commodity hardware. A task to be performed using the MapReduce framework has to be specified as two phases: a) the *map* phase, which is specified by a *map function*, takes input, typically from Hadoop Distributed File System (HDFS) files, possibly performs some computations on this input, and distributes it to worker nodes, and b) the *reduce* phase which processes these results as specified by a *reduce function* (Fig.2.2). An important aspect of MapReduce is that both the input and the output of the *map* step are represented as *key/value pairs* and that pairs with same key will be processed as one group by a *reducer*. The *map* step is parallelly applied to every pair with key k_1 of the input dataset, producing a list of pairs with key k_2. Subsequently, all pairs with the same key from all lists are grouped together, creating one list for each key (*shuffling step*). The *reduce* step is then parallelly applied to each such group, producing a list of key/value pairs:

$$map : (k_1, v_1) \rightarrow list(k_2, v_2) \text{ and } reduce : (k_2, list(v_2)) \rightarrow list((k_3, v_3))$$

Additionally, a *combiner function* can be used to run on the output of the *map* phase and perform some filtering or aggregation to reduce the number of keys passed to the *reducer*. The MapReduce architecture provides good scalability and fault tolerance mechanisms. MapReduce was originally introduced by Google in 2004 and was based on well-known principles of parallel and distributed processing. It has been widely adopted through Hadoop (an open-source implementation), whose development was led by Yahoo and later became an Apache project[1].

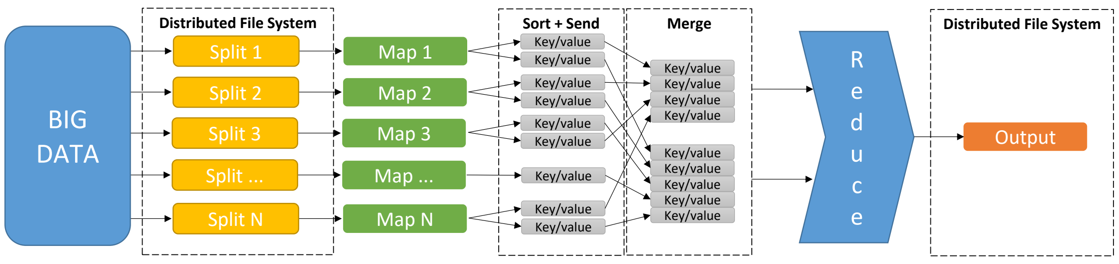


Figure 2.2: MapReduce programming model.

2.1.1.2 Apache Spark

To overcome limitations of the MapReduce paradigm and Apache Hadoop (especially regarding iterative algorithms), Apache Spark[2] was developed. This is also an open-source

[1]https://hadoop.apache.org/
[2]https://spark.apache.org/

cluster-computing framework based on Resilient Distributed Datasets (RDDs), read-only multisets of data items distributed over the computing nodes. RDDs form a kind of distributed shared memory, suitable for the implementation of iterative algorithms. Apache Spark achieves high performance for both batch and streaming data, using a state-of-the-art DAG (Directed Acyclic Graph) scheduler (an example is depicted in Fig. 2.3), a query optimizer and a physical execution engine.

DAG scheduler is the scheduling layer of Apache Spark that implements stage-oriented scheduling. It transforms a logical execution plan (i.e. RDD lineage of dependencies built using RDD transformations) to a physical execution plan (using stages).

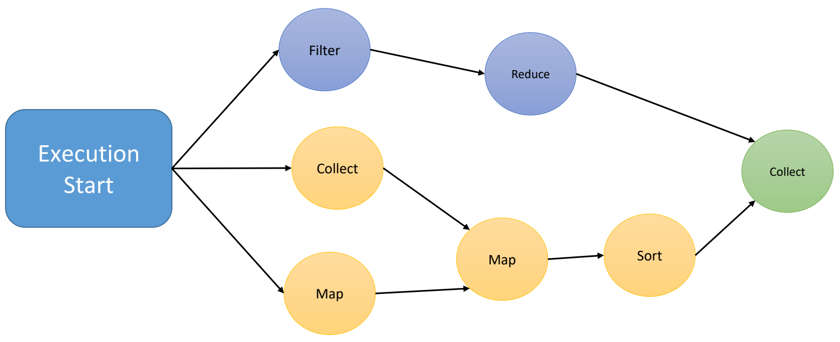


Figure 2.3: An example of a DAG (Directed Acyclic Graph) scheduler.

The data transformations that take place in Spark are executed in a "lazy" way. Transformations are lazy in nature: when we call some operation for an RDD, it does not execute immediately; it is executed when output is requested. Spark maintains a record of which operation is being called (through DAG). We can think of a Spark RDD as the data that we built up through transformations. Since transformations are lazy in nature, we can execute operations any time by calling an action on data. Hence, in lazy evaluation, data is not loaded, and computations are not performed until it is necessary.

2.1.2 Big Spatial and Spatio-Temporal Data Analytics Systems

In the next subsections, we will present in detail a popular representative of each of group of systems (Hadoop-based and Spark-based Spatial and Hadoop-based and Spark-based Spatio-temporal Data Analytics Systems). SpatialHadoop (`http://spatialhadoop.cs.umn.edu/`), a full-fledged MapReduce framework with native support for spatial data, is presented in Section 2.1.2.1. Section 2.1.2.2 is devoted to GeoSpark (`http://geospark.datasyslab.org`), an in-memory cluster computing framework for processing large-scale spatial data that uses Spark as its base layer and adds two more layers, the Spatial RDD (SRDD) Layer and Spatial Query Processing Layer, thus providing Spark with in-house spatial capabilities. ST-Hadoop (`http://st-hadoop.cs.umn.edu/`), the first full-fledged open-source MapReduce framework with a native support for spatio-temporal data, is presented in Section 2.1.2.3. ST-Hadoop is a comprehensive extension to Hadoop and SpatialHadoop that injects spatio-temporal data awareness inside each of their layers. In Section 2.1.2.4, STARK framework for scalable spatio-temporal data analytics on Spark (`https://github.com/dbis-ilm/stark`) is presented. It is built on top of Spark and provides a domain specific language (DSL) that seamlessly integrates into any (Scala) Spark program. It includes an expressive set of spatio-temporal operators for filter, join with various predicates as well as k nearest neighbor search. Moreover, in Section 2.1.2.5 we present a comparison of these systems regarding their capabilities and characteristics.

2.1.2.1 SpatialHadoop

SpatialHadoop (`http://spatialhadoop.cs.umn.edu/`) [1, 13] is a full-fledged MapReduce framework with native support for spatial data. It is an efficient disk-based distributed spatial query processing system. Note that MapReduce [11] is a scalable, flexible and fault-tolerant programming framework for distributed large-scale data analysis.

SpatialHadoop [1, 13] (see in Fig. 2.4 its architecture) is a comprehensive extension to Hadoop [14] that injects spatial data awareness in each Hadoop layer, namely, the language, storage, MapReduce, and operations layers. In the *Language* layer, SpatialHadoop adds a simple and expressive high-level language for spatial data types and operations. In the *Storage* layer, SpatialHadoop adapts traditional spatial index structures as Grid, R-tree, R^+-tree, Quadtree, etc. to form a two-level spatial index [15]. SpatialHadoop enriches the *MapReduce* layer by two new components, *SpatialFileSplitter* and *SpatialRecordReader* for efficient and scalable spatial data processing. *SpatialFileSplitter* (*SFS*) is an extended splitter that exploits the global index(es) on input file(s) to early prune file cells/blocks not contributing to answer, and *SpatialRecordReader* (*SRR*) reads a split originating from spatially indexed input file(s) and exploits the advantages of the local indices to efficiently process it. At the *Operations* layer, SpatialHadoop is also equipped with a several spatial operations, including range query, k-NN query and spatial join. Other computational geometry algorithms (e.g. polygon union, skyline, convex hull, farthest pair and closest pair) are also implemented following a similar approach [16].

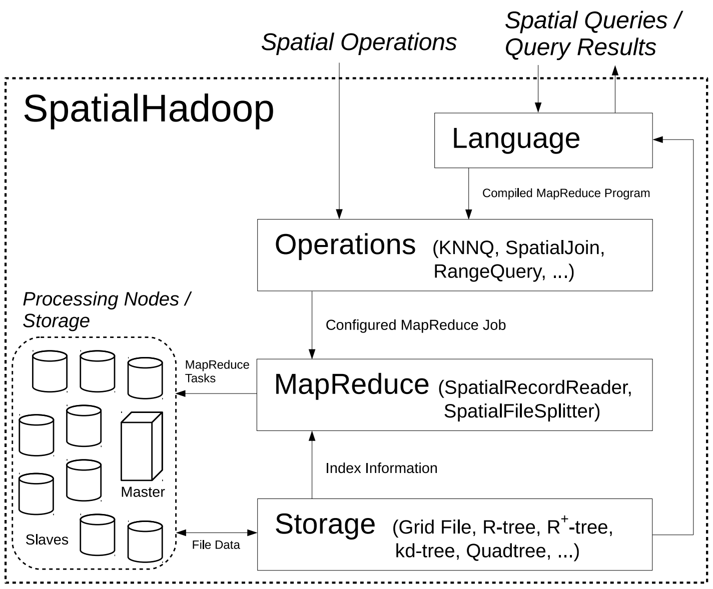

Spatial Operations
Spatial Queries / Query Results
SpatialHadoop
Language
Compiled MapReduce Program
Operations (KNNQ, SpatialJoin, RangeQuery, ...)
Processing Nodes / Storage
Configured MapReduce Job
MapReduce Tasks
MapReduce (SpatialRecordReader, SpatialFileSplitter)
Index Information
Master
Slaves
File Data
Storage (Grid File, R-tree, R+-tree, kd-tree, Quadtree, ...)

Figure 2.4: SpatialHadoop system architecture [1].

The *Language* layer provides a high-level language with standard spatial data types and operations to make the system accessible to non-technical users. In particular, the language layer provides Pigeon [17] a simple high level SQL-like language that supports OGC-compliant spatial data types and spatial operations.

In general, a spatial query processing in SpatialHadoop consists of four steps [1, 2], regardless of whether we have one or two input files (Fig. 2.5, where two files as input are shown): (1) *Preprocessing*, (2) *Pruning*, (3) *Local Spatial Query Processing*, (4) *Global Processing*.

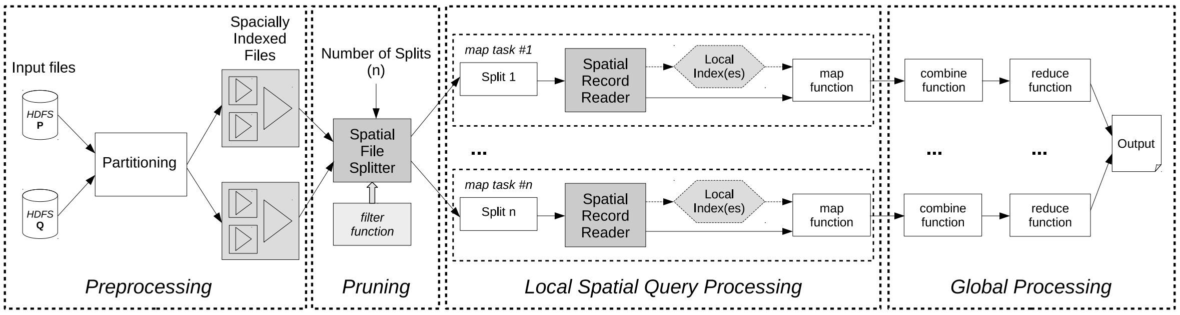


Figure 2.5: Spatial query processing in SpatialHadoop [1, 2].

The core of SpatialHadoop is used in several real applications that deal with big spatial data including MNTG [18], a web-based traffic generator; TAREEG [19], a MapReduce extractor for OpenStreetMap data; TAGHREED [20], a system for querying and visualizing twitter data, and SHAHED [21], a MapReduce system for analyzing and visualizing satellite data. SHAHED is a tool for analyzing and exploring remote sensing data publicly available by NASA in a 500 TB archive. It provides a web interface where users navigate through the map and the system displays satellite data for the selected area. HadoopViz [22] is a MapReduce-based framework for visualizing big spatial data, it can efficiently produce giga-pixel images for billions of input records.

2.1.2.2 GeoSpark

The **GeoSpark** (`http://geospark.datasyslab.org`) framework exploits the core engine of Apache Spark and SparkSQL, by adding support for spatial data types, indexes, and geometrical operations. GeoSpark extends the Resilient Distributed Datasets (RDDs) concept to support spatial data. It adds two more layers, the Spatial RDD (SRDD) Layer and Spatial Query Processing Layer, thus providing Spark with in-house spatial capabilities. The SRDD layer consists of three newly defined RDDs, PointRDD, RectangleRDD and PolygonRDD. SRDDs support geometrical operations, like Overlap and Minimum Bounding Rectangle. SRDDs are automatically partitioned by using the uniform grid technique, where the global grid file is split into a number of equal geographical size grid cells. Elements that intersect with two or more grid cells are being duplicated. GeoSpark provides spatial indexes like Quadtree and R-tree on a per partition base. The Spatial Query Processing Layer includes spatial range query, spatial join query, spatial k-NN query. GeoSpark relies heavily on the JTS (Java Topology Suite) and therefore conforms to the specifications published by the Open Geospatial Consortium. It is a robust and well implemented spatial system. Moreover, a lot of heterogeneous data sources are supported, like CSV, GeoJSON, WKT, NetCDF/HDF and ESRI Shapefile. GeoSpark does not directly support temporal data and operations.

In order to take advantage of the spatial proximity which is crucial for improving query speed, GeoSpark automatically repartitions a loaded Spatial RDD according to its internal spatial data distribution (Fig. 2.6 presents spatial partitioning techniques supported by GeoSpark). This is crucial for every computation, because it minimizes the data shuffles across the cluster and it avoids unnecessary CPU overheads on partitions that contain unwanted data.

GeoSpark can run spatial query processing operations on the SRDDs, right after the Spatial RDD layer loads, partitions are generated and indexing is completed. The spatial

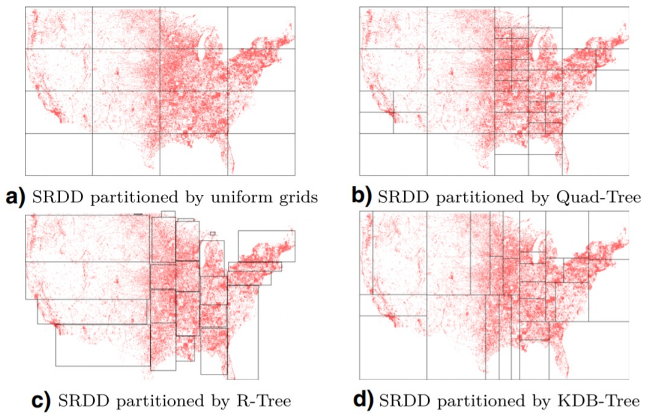

a) SRDD partitioned by uniform grids
b) SRDD partitioned by Quad-Tree
c) SRDD partitioned by R-Tree
d) SRDD partitioned by KDB-Tree

Figure 2.6: Spatial partitioning techniques [3].

query processing layer provides support for many spatial operations like range query, distance query, k Nearest Neighbors (k-NN) query, range join query and distance join query. In order to describe the distributed processing of GeoSpark we will analyze the simplest of the queries the range query. A spatial range query is faster and less resource-consuming because it just returns objects that the input query window object contains. To complete such queries, we need to issue a parallelized Filter transformation in Apache Spark, which introduces a narrow dependency. As a result, repartitioning is not needed. These is also a more efficient way, we can broadcast the query window to all workers and parallelize the processing across the cluster. The query processing algorithm needs only one stage, due to the narrow dependency which does not require data shuffle. In Fig. 2.7, the range query DAG and data flow is depicted.

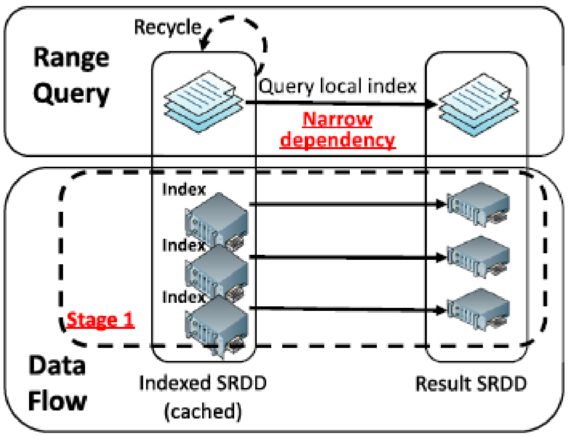


Figure 2.7: Range query DAG and data flow [3].

2.1.2.3 ST-Hadoop

ST-Hadoop (`http://st-hadoop.cs.umn.edu/`) [4, 5], see in Fig. 2.8 its architecture, is a full-fledged open-source MapReduce framework with a native support for spatio-temporal data. ST-Hadoop is a comprehensive extension to Hadoop [14] and SpatialHadoop [1] that injects spatio-temporal data awareness inside each of their layers, mainly, language, indexing, MapReduce and operations layers. In the *Language* layer, ST-Hadoop extends Pigeon language [17] to supports spatio-temporal data types and operations. In the *Indexing* layer, ST-Hadoop spatio-temporally loads and divides data across computation nodes in the Hadoop Distributed File System (HDFS). In this layer, ST-Hadoop scans a random sample obtained from the whole dataset, bulk loads its spatio-temporal index in-memory, and then uses the spatio-temporal boundaries of its index structure to assign data records with its overlap partitions. ST-Hadoop sacrifices storage to achieve more efficient performance in supporting spatio-temporal operations, by replicating its index into temporal hierarchy index structure that consists of two-layer indexing of temporal and then spatial. The *MapReduce* layer introduces two new components of *SpatioTemporalFileSplitter* and *SpatioTemporalRecordReader*, that exploit the spatio-temporal index structures to speed up spatio-temporal operations. Finally, the *Operations* layer encapsulates the spatio-temporal operations that take advantage of the ST-Hadoop temporal hierarchy index structure in the indexing layer, such as spatio-temporal range, spatio-temporal top-k nearest neighbor, and spatio-temporal join queries.

The key idea behind the performance gain of ST-Hadoop is its ability to load the data in HDFS in a way that mimics spatio-temporal index structures [23]. Hence, incoming spatio-temporal queries can have minimal data access to retrieve the query answer. The extensibility of ST-Hadoop allows others to extend spatio-temporal features and operations easily using similar approaches as described in [5].

Summit [24] is a full-fledged open-source library on ST-Hadoop MapReduce framework with *built-in* native support for trajectory data. Summit cluster contains one master node that breaks a MapReduce job into smaller tasks, carried out by slave nodes. Summit modifies three core layers of ST-Hadoop, namely, *Language*, *Indexing* and *Operations*. The *Language* layer adds new SQL-Like interface for trajectory operations and data types. The modifications and the implementation of the *Indexing* (trajectory indexing) and *Operation* (trajectory range query, trajectory k nearest neighbor query and trajectory similarity query) layers are more

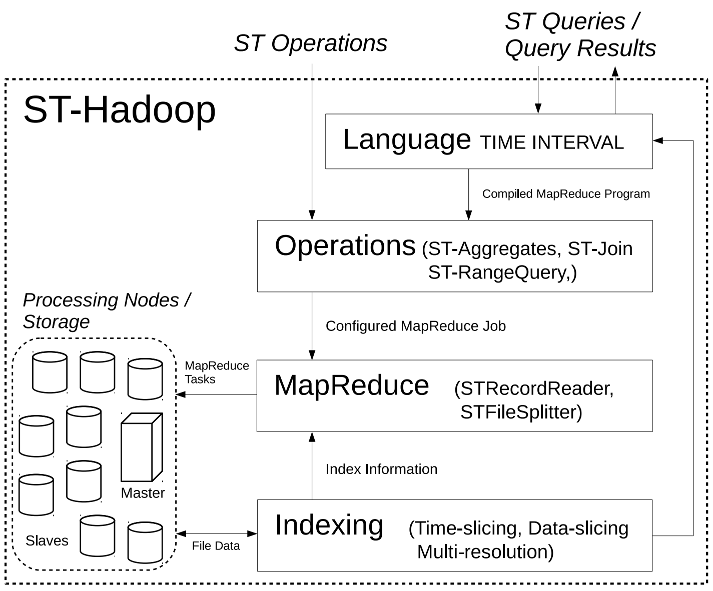

ST Operations
ST Queries / Query Results
ST-Hadoop
Language TIME INTERVAL
Compiled MapReduce Program
Operations (ST-Aggregates, ST-Join ST-RangeQuery,)
Processing Nodes / Storage
Configured MapReduce Job
MapReduce Tasks
MapReduce (STRecordReader, STFileSplitter)
Master
Index Information
Slaves
File Data
Indexing (Time-slicing, Data-slicing Multi-resolution)

Figure 2.8: ST-Hadoop system architecture [4, 5].

complicated.

2.1.2.4 STARK

The **STARK** framework (`https://github.com/dbis-ilm/stark`) [6] is a promising new spatio-temporal data analytics framework (see in Fig. 2.9 its architecture). It is tightly integrated with Apache Spark by leveraging Scala language features and it adds support for spatial and temporal data types and operations. Furthermore, STARK exploits SparkSQL functionality and implements SQL functions for filter, join with various predicates and aggregate vector as well as raster data. STARK also supports k nearest neighbor search and a density-based clustering operator allows to find groups of similar events. STARK includes spatial partitioning and indexing techniques for fast and efficient execution of the data analysis tasks.

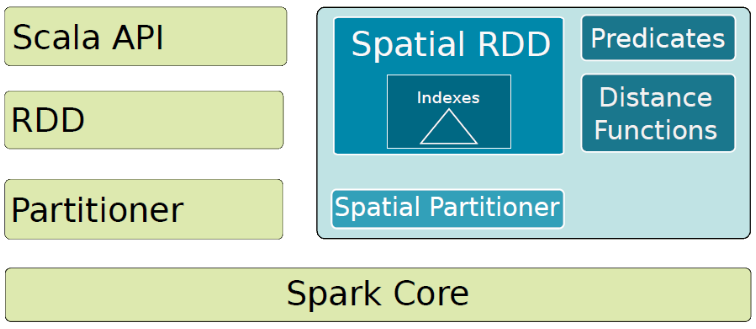


Figure 2.9: STARK framework architecture [6].

The main data structure of STARK is STObject. This class is a super-class of all spatial objects and provides a time component. STObject relies on the JTS library with the JTSplus extension, thus it supports all types of geometry objects, such as Point, Polygon, Linestring, Multipoint, Multypolygon and Multilinestring. Regarding the temporal data-type, the STObject contains o time component that facilitates temporal operations. Besides the Scala API based on the core RDDs, STARK is integrated into SparkSQL and implements SQL functions to filter, join, and aggregate vector and raster data.

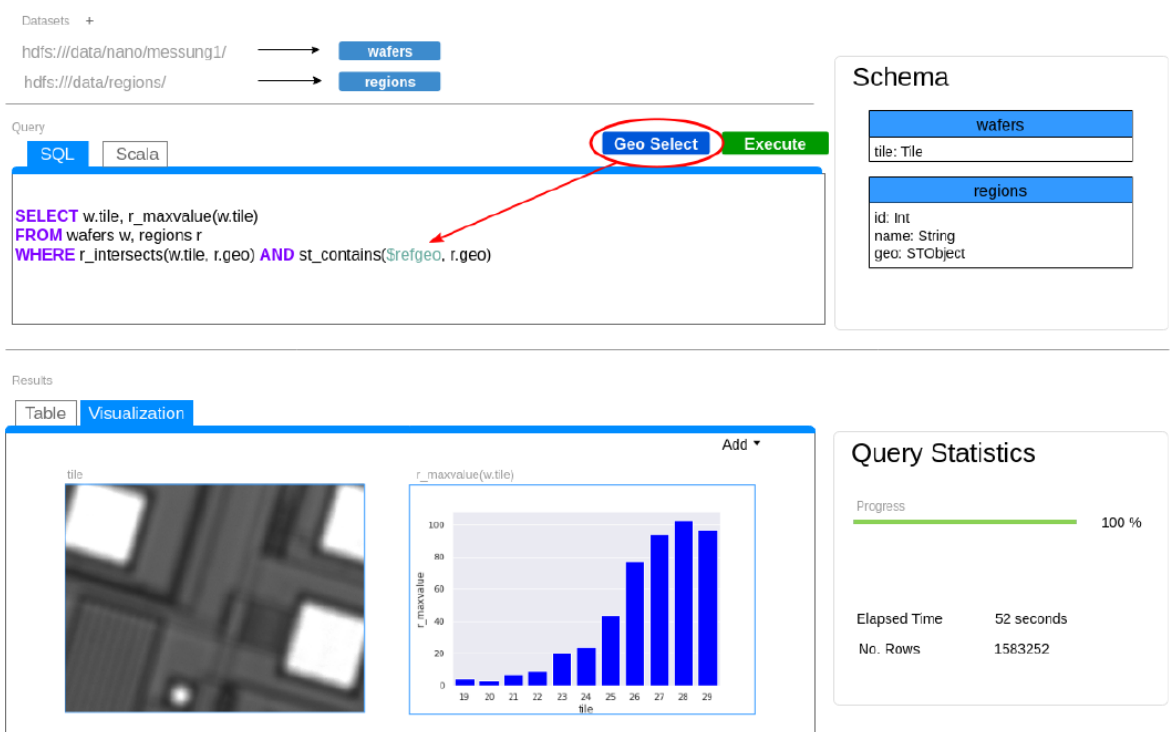
The framework can index any partition, using an in memory spatial index structure. The R-tree index structure is currently supported by STARK, because of its JTS library dependency. Also, other indexing structures are planned to be included in future versions. There are three available indexing modes: (1) No Index, (2) Live Indexing and (3) Persistent Index. Furthermore, the same index can be used among different scripts, eliminating costly index creation time.

STARK is taking advantage of the Hadoop environment, resulting to parallel execution on cluster nodes. Every node processes a fragment of the whole dataset, which is call a partition. STARK spatial and spatio-temporal partitioning does not utilize Spark's built-in partitioners, for example a hash partitioner. Currently STARK uses only spatial partitioning, temporal partitioning is under development. In order to take advantage of the locality of data, STARK uses the following partitioners: (1) Grid Partitioner, (2) Binary Space Partitioner and (3) Partitioning Polygons.

STARK is heavily depended on Spark's capabilities; therefore the visualization tools of Spark can be used to visualize STARK data. There is only one documented visualization tool designed especially for STARK spatial visualization (see Fig. 2.10), which also combines raster data in final layout. This visualization tool comes with a web interface [7] where users can interactively explore raster and vector data using SQL.

2.1.2.5 Comparison of distributed systems

In Table 2.1, we compare the four systems presented in the previous sections, regarding the features included in the presentation of these systems. Note that, there was non-available (N.A.) information available in the literature regarding some features of certain systems (language for GeoSpark, visualization for ST-hadoop and applications for GeoSpark and STARK).



hdfs:///data/nano/messung1/ → wafers
hdfs:///data/regions/ → regions

```
SELECT w.tile, r_maxvalue(w.tile)
FROM wafers w, regions r
WHERE r_intersects(w.tile, r.geo) AND st_contains($refgeo, r.geo)
```

Schema

wafers
tile: Tile

regions
id: Int
name: String
geo: STObject

Query Statistics

Progress 100 %

Elapsed Time	52 seconds
No. Rows	1583252

Figure 2.10: STARK Visualization, Web Interface [7].

2.2 CPU and GPU Data Processing

2.2.1 CPU Processing

The primary goal of conventional CPUs is to optimize their serial performance. Typically, manufacturers relied on Dennard scaling to increase processor frequency and thus processing speed [25]. Dennard scaling depends on the size of transistors in relation with their clock frequency and voltage. The transistors keep on shrinking and more of them are integrated on a single die and their clock frequency increases. In order to keep power consumption constant, the operating voltage needs to be decreased. This change has led to a 100× performance gain of recent CPUs, when compared to older CPUs. CPU manufacturers have also developed microarchitecture advances that extract implicit instruction level parallelism (ILP) from the instruction stream, to improve serial performance. The main technique of these advances is the CPU pipeline which overlaps the execution stages of different instructions. The best performance is achieved when the pipeline is full and the processor can start and complete one instruction per cycle. The major disadvantage of the pipeline is that it stalls when instructions are dependent on each other, or when the processor should wait on memory access.

A variety of techniques are being used in modern CPUs, to keep the pipeline from stalling

Table 2.1: Overview of the comparative criteria of big spatial and spatio-temporal data analytics systems

	GeoSpark	**SpatialHadoop**	**ST-Hadoop**	**STARK**
Datatypes	Point, Rectangle, LineString, Polygon	Point, Rectangle, LineString, Polygon	STPoint, Time, Interval	Point,Polygon, Linestring, Multipoint, Multypolygon, Multilinestring, Time, Interval
Indexes	R-tree, Quadtree	R-tree	Temporal hierarchy index, Temporal Slicing, Spatial index	R-tree
Partitioning	Quadtree, k-d tree, STR-tree, Voronoi, Uniform, Hilbert	Quadtree, STR-tree, STR+, k-d tree, Hilbert-curve, Z-curve	Time-partitioning slicing, Data-partitioning slicing	Grid Partitioning, Binary Space Partitioning
Operations	Range, k-NN, Spatial join, Distance join	Range, k-NN, Spatial join	Spatio-temporal range, spatio-temporal top-k nearest neighbor, spatio-temporal join	intersect, contains, containedBy, spatial join, nearest neighbors, clustering, skyline
Processing	DAG execution model	MapReduce	MapReduce	DAG execution model
Language	N.A.	Pigeon	Pigeon	Piglet, Pig Latin
Visualization	GeoSparkViz	Single level image, Multilevel images	N.A.	Web UI
Applications	N.A.	MNTG, TAREEG, TAGHREED, SHAHED, HadoopViz	Summit	N.A.

and increase ILP [26]. One example is the that branch prediction continues to fetch and decode instructions of the predicted branch in the instruction stream, which keeps the early stages of the pipeline full. Speculative execution also executes the instructions of predicted branches and only discards their results, if the prediction later proves not to be correct. The Out-of-order execution optimization reorders the instruction stream to achieve less instruction dependencies and memory stalls. Dennard scaling and these microarchitecture advances have produced a gain of 52 per cent per year between 1986 and 2003.

The use of ILP is limited by the performance of the memory system [26]. Data I/Os stall the processor pipeline, when the processor cannot find independent instructions to feed in the pipeline. The total stall is dependent on the memory latency and the capability of concurrent memory accesses that can be served depending on the memory bandwidth. The rate that memory improves over time has lagged processor performance, both for latency and bandwidth.

Early CPUs access memory in a one clock cycle. Current CPus have to wait hundreds of cycles. In many applications, even more in those that rely on fast integer performance [26], there is not enough instruction level parallelism available to overcome this access latency [27]. One way To reduce memory access latency is for modern CPUs to use large caches. This feature allows CPUs to exploit temporal and spatial data access locality.

However, even with large and fast caches, data-intensive operations are limited by memory access due to unavoidable cache misses when loading fresh (not cached) data. The frequencies increase in processor has stopped since 2003 [26]. Physical limitations have risen and manufacturers cannot further reduce operating voltages. So, the operating frequency has reached a threshold which can not be surpassed without excessive power consumption and heat dissipation. On the other hand, the microarchitecture advances to increase ILP are not energy-efficient because their implementation requires an increasing amount of the processor's transistor budget. Since scalar performance is no longer increasing, manufacturers have turned to increase throughput, by exploiting explicit data parallelism. Multi-core CPUs integrate multiple processor cores on a single die. Simultaneous multi-threading (SMT) enables independent threads to utilize different execution units of a core which explicitly increases ILP. SIMD instructions work on multiple data items in a single cycle. These developments mean that multi-core CPUs are becoming more and more similar or even identical to GPUs.

2.2.2 GPU Processing

Originally, GPUs were developed for special purposes, just to accelerate graphics rendering, mostly in 3D games. The main concern of GPUs manufacturers was to make the graphics rendering pipeline more flexible and better support a most of the 3D games. As a result, GPUs are primarily optimized for applications that render graphics. These applications mainly in need of the following three interrelated features: (1) high degree of data parallelism, (2) latency tolerance and (3) high demands on memory bandwidth.

In one hand, CPUs rely on extracting the implicit ILP from an instruction stream, on the other hand, GPUs rely on explicit data parallelism to keep processing cores busy. As a result, GPUs consist of multiple simple processing cores instead of fewer advanced and complex processing cores, as CPUs do. The computational performance of GPUs can scale about linearly depending on the transistor budget, whereas the microarchitectural features of CPUs, scale proportional to the square root of the transistor budget [28].

It makes sense that the latency of processing an individual data item is less important for GPUs, because they are aggregate throughput-bound, instead of being serial-bound in performance. This results to two major effects concerning the hardware design. The first one is that it allows us to downgrade the processing frequency and include more transistors to implement processing cores within a specific power budget. The second one is that instead of reducing the latency of an individual data item through caches and microarchitectural advances, the latency is hidden by processing other data items. To aid latency hiding, the GPU hardware allows for a massive over-subscription of threads. One example is that each streaming multiprocessor (SM) of an Ampere A100 GPU can execute a total of four independent warps at a time [29] (A warp is a group of threads that execute the same instruction). Each SM can manipulate 64 different warps that are waiting for execution. At every cycle, the SM can switch between active and inactive warps without overhead.

To implement this microarchitecture and support so many threads, GPUs contain big register files, much greater than than the ones of CPUs. When compared to CPUs, for the GPU side, large L1 cache also places more emphasis, which are closely integrated to the processing cores. In contrast with CPUs, the shared last-level cache is smaller on GPUs. The Ampere A100 has 8× fewer cache resources per SM than the EPYC 7702P has per core.

Due to the streaming data access of GPU graphics, the computational workloads slightly depend on data reuse so caches are not very useful. GPU memory is optimized for high bandwidth transactions, so that the input data adequately feeds the large number of processing cores. The memory bus is also clocked faster, up to 7 GHz for GDDR6 memory. Furthermore, high-performance GPUs use three-dimensional stacked memory. The memory is joined with the GPU processor in the same die in a single package. Stacked memory is addressed through an ultra-wide data bus. For example,the Ampere A100 uses ten 512-bit memory controllers which results in a overall bus width of 5120 bits. This is an order of magnitude wider than the 8x 64-bit bus width of the EPYC 7702P.

2.2.3 GPU Programming

Programmers use a special programming model to develop parallel programms in a scalable and extensible way [30]. This model represents GPU hardware as an abstract parallel processor. It helps programmers to define how their parallel program will be executed on the device and also aids in the workload partitioning in order to achieve scalable parallelism. GPU programming is different than CPU programming in many important ways. The main and most popular implementations of this programming model are CUDA [30] and OpenCL [31]. OpenCL represents parallel processors, e.g., multicore CPUs or GPUs, as computational devices consisting of compute units (CUs). On NVIDIA GPUs, each compute unit maps to a Streaming Multiprocessor (SM), and on multi-core CPUs, a compute unit represents a logical CPU core.

NVIDIA GPUs contain 1-N Streaming Multiprocessors (SM). Each SM has 1-4 warp schedulers. Each warp scheduler has a register file and multiple execution units. The execution units may be exclusive to the warp scheduler or shared between schedulers. Execution units include CUDA cores (FP/INT), special function units, texture, and load store units.

Both of these programming models divide programming into host code and device code. Host code is executed in as a single thread process on the host CPU. It's main purpose is to coordinate operations on the device, such as initiating the execution of device code and transferring data between host and device memory spaces. The device code is executed in parallel on the device. The device program is executed in kernels which are scalar functions, in which the operations are processed on a single datum of a data-parallel task. Whenever the programmer launches a kernel, he can specify the hierarchy of independent kernel instances that execute on the device. Each kernel instance is a thread (in CUDA Fig.2.11). Individual threads are arranged into thread blocks. All the thread blocks of a kernel invocation make up the grid. The CUDA threads of a single thread block can cooperate with each other through special instructions, fast barrier synchronizations, atomic functions and a very fast memory space called shared memory. Because it is on-chip, shared memory is much faster than local and global memory. In fact, shared memory latency is roughly 100x lower than uncached global memory latency (provided that there are no bank conflicts between the threads) [32]. Shared memory is allocated per thread block, so all threads in the block have access to the same shared memory. Threads can access data in shared memory loaded from global memory by other threads within the same thread block. This capability (combined with thread

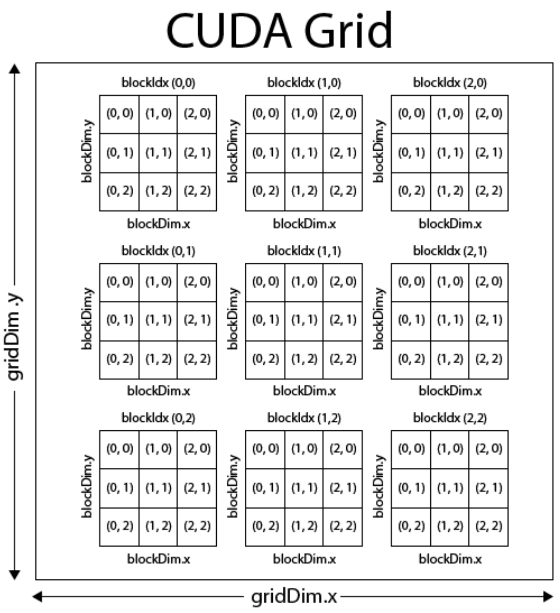


Figure 2.11: CUDA threads, thread blocks and grid.

synchronization) has a number of uses, such as user-managed data caches, high-performance cooperative parallel algorithms (parallel reductions, for example), and to facilitate global memory coalescing in cases where it would otherwise not be possible. In contrast, threads from different thread blocks execute completely independently.

Using this two-tiered hierarchy of threads and thread blocks, CUDA programming models support scalable parallelism. The programmer, in order to use this programming model, should partition the algorithm in two discrete levels. The first one, the individual thread blocks, work on coarse-grained subproblems which can be solved independently in parallel. Each thread block executes on a dedicated SM (or compute unit for OpenCL). Multiple thread blocks can execute on different SMs in parallel or on the same compute unit sequentially. The second one, the threads within a single thread block, work on fine-grained subproblems which can be solved cooperatively in parallel. The GPU hardware supports this fine-grained thread and data parallelism through fast barrier synchronization, access to shared memory, lightweight thread creation, and zero-overhead scheduling. Additionally, independent grids can execute concurrently given sufficient hardware resources.

2.2.4 CPU vs GPU Data Processing

The main and most obvious difference between programming on GPUs and CPUs are the total number of running threads, and how these threads cooperate. Generally, on CPUs, few threads operate independently on coarse-grained subproblems. On multi-core CPUs, each CPU core typically occupies a single hardware thread that operates on an independent partition of the data. Usually threads running on different CPU cores that communicate with each other, should avoid programming practices that cause decreased performance like accessing shared or nearby data. In contrast, GPUs execute a plethora of threads to hide the latency of such operations. Occasionally, individual threads have to cooperate with each other to achieve maximum performance. A classic example of a data processing task, where threads cooperate to achieve high throughput, is parallel reduction on GPUs [33]. Another significant difference between GPU and CPU programming is the Single Instruction, Multiple Thread (SIMT) [34] execution model.

SIMT is an execution model used in parallel computing where single instruction, multiple data is combined with multithreading. It is different from SPMD in that all instructions in all “threads” are executed in lock-step. The SIMT execution model has been implemented on several GPUs and is relevant for general-purpose computing on graphics processing units (GPGPU), e.g. some supercomputers combine CPUs with GPUs. The SIMT execution model is similar to the Single Instruction, Multiple Data (SIMD) execution model supported by CPU vector instructions. However, a crucial difference is that GPU kernels are written as scalar functions, independent of the SIMD instruction width of the processor.

The GPU hardware also takes care of masking results when different threads follow separate branches in the kernel code. Nevertheless, to maximize performance, programmers still have to take hardware details, such as the warp size, into account. Programmers should avoid diverging code paths for the threads inside a warp; utilize warp-level primitives, e.g., warp-level reductions or ballot and shuffle instructions; and let the threads of a warp access adjacent global memory locations, so that the GPU can coalesce these accesses into as few memory transactions as possible.

2.3 Architectural research focus

Distributed systems such as Apache Hadoop and Apache Spark are already heavily used for big data analysis in data centers. Surely, these systems to achieve efficient performance, they need to use a multitude of processing nodes. Unavoidably, the hardware cost will increase analogously to the number of the nodes. If we need to operate on big spatial data using mainstream hardware, the only viable solution is to exploit the multi-core CPU on a single workstation or take advantage of its GPU. Furthermore, this solution is not only cost efficient but also eliminates the latency derived from large amounts of data transferred between the nodes of a distributed system. In the next chapters we will focus our research in the multi-core CPU and GPU architectures. We will present novel methods and we will conduct extensive experiments to determine their performance.

Chapter 3

Query Processing with CPU and SSD

Nowadays, the volume of available spatial data (e.g. location, routing, navigation data, etc.) is continuously increasing world-wide. To exploit these data, efficient processing of spatial queries is of great importance due to the wide area of applications that may address such queries. The most common spatial queries where points are involved are point-location, window, distance-range and K nearest-neighbor queries (PLQs, WQs, DRQs and k-NNQs, respectively, in the sequel). At a higher level, such queries have been used as the basis of many complex operations in advanced applications, for example, geographical information systems (GIS), location-based systems (LBS), geometric databases, CAD, or economic forecasting [35].

The use of efficient spatial indices is very important for performing spatial queries and retrieving efficiently spatial objects from datasets according to specific spatial constraints [36]. Hierarchical indices are useful due to their ability to focus on the interesting subsets of data. This focusing results in an efficient representation and execution times on query processing and thus, it is particularly useful for performing spatial operations. An example of such indices is the Quadtree [37], which is based on the principle of recursive decomposition of space and has become an important access method for spatial applications [38].

The External Balanced Regular (xBR)-tree [39] is a secondary memory structure that belongs to the Quadtree family (widely used in GIS applications, which is suitable for storing and indexing points and, in extended versions, line segments, or other spatial objects). We use an improved version of xBR-tree, called xBR$^+$-tree [40], which is also a disk-resident structure. The xBR$^+$-tree improves the xBR-tree in the node structure and in the splitting process. The node structure of the xBR$^+$-tree stores information which makes query processing more efficient. In addition, the xBR$^+$-tree outperforms R*-tree and R$^+$-tree (in terms of I/O activity and execution time) for the most common spatial queries, like PLQs, WQs, DRQs, k-NNQs, etc. [41].

The advent of non-volatile memories (NVM) has enabled a brand-new class of storage devices with exciting features that will prevail in the storage market in the near future. Their high read and write speeds, small size, low power consumption and shock resistance are some of the reasons that made them storage medium of choice. NAND flash is undoubtedly the most popular NVM today. Storage devices based on NAND Flash are found both in consumer devices and enterprise data-centers. However, upcoming technologies, such as 3D XPoint from Intel and Micron, make possible even more efficient devices [42]. At the very beginning, raw Flash memory chips were embedded in mobile devices and other electronics. However, soon enough, the increasing needs for efficient storage drove the emergence of Solid-State Drives (SSDs). SSDs are composed by Flash chips, embedded controllers and DRAM [43]. Contemporary devices incorporate from a few to many NAND chips, supplying capacities even of tens of terabytes in high-end systems. Flash controller, usually a 32-bit embedded CPU, executes the firmware that controls SSD operation, while DRAM is utilized to store metadata, information regarding address mapping and for user data caching. Firmware is fundamental for SSD operation [44]. Its main responsibility is to map virtual addresses, as they are seen by the host, to physical addresses in flash chips. For this reason is also known as Flash Translation Layer (FTL). FTL performs tasks for garbage collection, wear leveling and management of bad blocks. SSDs exhibit higher write and especially read performance than Hard Disk Drives. This performance advantage is maximized when issuing commands that massively write to / read from SSDs large sequences of consecutive pages (due to exploiting the internal parallelism of SSDs), instead of issuing commands that write to / read from SSDs the pages of such sequences in small subsequences, or even, one-by-one [45].

Due to power limitations of chips, the possible increase of CPU clock speed that can be achieved in new generations of CPUs is limited and, therefore, manufacturers improve performance by creating CPUs with multiple cores. Multi-core CPUs are common in today's commodity hardware. On the other hand, processing of spatial queries has been widely investigated for decades, focusing mainly on single threaded code. Processing of spatial queries usually demands significant processing power and developing algorithms that utilize multiple cores can improve overall performance considerably.

New algorithms where presented [46] for processing large sequences of common spatial queries (PLQs, WQs, DRQs) using xBR$^+$-trees in SSDs. These algorithms are especially designed to massively read from SSDs large sequences of pages needed for answering such queries and are also included in this chapter. Such large sequences of queries (batch queries) appear frequently in applications. Moreover, in this chapter, we elaborate on the algorithms of [46] and create new algorithms that additionally take advantage of the multiple CPU cores. Extending the experimentation presented in [46], based on small and large datasets, we experimentally study the performance of these new, SSD based, algorithms against processing of batch queries by repeatedly applying existing algorithms for these queries and further study the performance of the new algorithms that utilize parallelism against the ones taking advantage of SSDs only. Our experiments show that the new algorithms taking advantage of SSDs and even further the ones that also utilize multiple cores are clear performance winners. However, the performance achieved by the mutiple cores of a CPU, or the amount of data a centralized system can process might not be enough for future applications. Therefore, we also discuss how these new parallel algorithms can be extended to work in a distributed environment, like the ones presented in Section 2.1, taking advantage of parallelism between machines, while processing data of larger scales.

The sequel is organized as follows. In Section 3.1 we review related work on spatial query processing over xBR-trees, as well as, on indices taking advantage of SSDs performance, on processing spatial queries using multiple cores and provide the motivation of this work. In Section 3.2, we describe the most important characteristics of the xBR$^+$-tree. Section 3.3 presents new algorithms for batch queries processing using xBR$^+$-tree in SSDs. Section 3.4 presents extensions of these new algorithms that further take advantage of multiple cores. The results of our experiments are discussed in Section 3.5. Section 3.6 discusses how the new parallel algorithms can be extended to work in a distributed environment. Finally, Section 3.7 provides the conclusions arising from our work and discusses future work directions.

3.1 Related Work and Motivation

In this section, we first briefly review the xBR-tree family and continue with the most representative spatial indexes, taking advantage of SSD performance and spatial processing on multi-core processors. Finally, the main motivation of this work is exposed.

3.1.1 The xBR-tree Family

The xBR-tree was initially proposed in [39] as a secondary-memory pointer-based structure that belongs to the Quadtree family. The original xBR-tree was enhanced in [47]. The xBR$^+$-tree [40, 41] is a further improved extension of the xBR-tree regarding performance of tree creation and spatial query processing. Bulk-loading and bulk-insertion methods for xBR$^+$-trees are presented in [48] and [49], respectively.

In [41], an exhaustive performance comparison (I/O activity and execution time) of xBR$^+$-trees (non-overlapping trees of the Quadtree family), R$^+$-trees (non-overlapping trees of the R-tree family) and R*-trees (industry standard belonging to the R-tree family) is performed. In this comparative study, several performance aspects are studied, like tree building and processing single point dataset queries (PLQs, WQs, DRQs and k-NNQs) and distance-based join queries (DJQs), using medium and large spatial (real and synthetic) datasets. As a conclusion, the xBR$^+$-tree is a clear winner for tree building and query performance. The excellent building performance of the xBR$^+$-tree is due to the regular subdivision of space that leads to much fewer and simpler calculations. The higher query performance of the xBR$^+$-tree is due to the combination of the regular subdivision of space, the additional representation of the minimum rectangles bounding the actual data objects, the algorithmic improvements of certain spatial queries and the storage order of the entries of internal nodes.

3.1.2 Spatial Indexes for Flash SSDs

NAND Flash provides superior performance compared to traditional magnetic disks but has some intrinsic characteristics. It exhibits asymmetry in the read, write, and erasure speeds and a page must be erased first before being re-programmed. Erase operations take place at block level, while reads and writes are performed at page level. SSDs inherit some of these characteristics, thus in most devices read operations are faster than writes, while difference exist among the speeds of sequential and random I/Os as well. Especially, random writes may initiate garbage collection which is impacts the efficiency of the device. On the other hand, the high degree of internal parallelism of latest SSDs substantially contributes to the improvement of I/O performance [50]. Many research efforts have been made for Flash efficient database indexes. The works for spatial data processing mostly concern the R-tree.

The RFTL [51] is the first effort towards a flash efficient implementation of the R-tree.

It is based on recording deltas for update operations. An in-memory buffer is utilized to hold the deltas before be persisted in batches. The same method has also been applied for the Aggregated R-tree in [52].

The LCR-tree [53] exploits a small section of SSD to log update operations. In contrary to other works it accumulates all the deltas for a particular node to one page in Flash. This way it ensures only one additional page reading to reach a tree node. The LCR-tree exhibits better performance than the original R-tree and the RFTL in mixed search/insert experimental scenarios. In the FOR-tree [54] authors aim to reduce small random writes by introducing overflow nodes to the R-tree. They propose new search and insert algorithms and a buffering algorithm for efficient caching of original and overflow nodes.

Regarding to non R-tree spatial indexes, the F-KDB [55] is a log-structured implementation of the K-D-B-tree for Flash, the MicroGF [56] is a 2D Grid File like structure for raw Flash, that is embedded in wireless sensor nodes, while a first effort towards to an efficient Grid-File for SSDs is presented in [57].

Furthermore two generic frameworks for spatial indexing have been proposed so far, which can encapsulate different data structures. FAST [58] utilizes the original insertion and update algorithms, buffers updates in RAM and flashes them to the SSD at once. eFIND [59] is a newer generic framework that provides better performance than FAST.

MPSearch [50] [60] is a multi-path search algorithm for the B$^+$-tree that performs batch searches considering the characteristics of SSDs to accelerate performance. To the best of our knowledge, there are not any works concerning spatial batch-queries processing for Flash SSDs. Motivated by this observation, in [46], we developed new algorithms for processing common spatial batch queries (PLQs, WQs, DRQs), using xBR$^+$-trees in SSDs. These algorithms are designed for maximizing performance by exploiting the internal parallelism of SSDs.

3.1.3 Spatial Processing on Multi-core Processors

Multi-core processors usually have a small number of cores (in most cases, between two and eight) as opposed to high performance computing systems. This characteristic constitutes a challenge for the parallelization of algorithms, since extensive use of parallelism can degrade performance, due to the overhead of parallelization and competition for resources, like shared memory. Instead, parallelism on a small scale might prove more appropriate. Spatial operations used for processing spatial data, even I/O bound ones, require significant core processing power; therefore, it is worthwhile to re-design and parallelize such spatial algorithms for multi-core CPU architectures that are commonly available in nowadays computers.

There are several parallelisms that can be mapped to different parallel hardware (e.g., multi-core CPUs, GPUs and MICs) at different levels (e.g., multi-processor, thread-blocks, SIMD elements). In the context of spatial processing, most of the research has been applied on multi-core CPUs, GPUs and MICs. In [61], data-parallel designs and implementations of point-to-polyline shortest distance computation and point-in-polygon topological test on different commodity hardware using real large-scale spatial data are proposed, comparing their performance and discussing important factors that may significantly affect the performance. Moreover, parallel designs and implementations of spatial indexes on commodity parallel hardware are becoming available, like GPU-based R-trees [62], Quadtrees [63] and simple flat Grid files [64] for spatial indexing and spatial filtering.

In [65] a parallel version of the plane-sweep algorithm targeted towards the small number of processing cores available on commonly available multi-core systems is presented. Experimental results show that the proposed algorithm significantly outperforms the serial plane-sweep on such spatial systems. An improved version of parallelizing plane-sweep algorithms for spatial computations on multi-core processors has been published in [66].

To the best of our knowledge, there has not appeared any work in the literature on processing spatial batch-queries which combines the use of SSDs and also takes advantage of multiple CPU cores, to improve performance. Therefore, in this chapter, we elaborate on the algorithms of [46] and develop new algorithms for processing such queries that combine the utilization of an efficient index (the xBR$^+$-tree), exploit the massive I/O advantages of SSDs and make use of the multiple cores existing in a modern CPU.

3.2 The xBR$^+$-tree Structure

In this section, we present the basics of the xBR$^+$-tree (its advantages over trees belonging to the R-tree family are summarized in Section 3.1.1). The xBR$^+$-tree [40] is a hierarchical, disk-resident Quadtree-based index for multidimensional points (i.e. it is a totally disk-resident, height-balanced, pointer-based tree for multidimensional points). For 2d space, the space indexed is a *square* and is recursively subdivided in 4 $(= 2^2)$ equal subquadrants, while for 3d space, the space indexed is a *cube* and it is recursively subdivided in 8 ($= 2^3$). In this chapter, we focus on 2d data. The tree nodes are disk pages of two kinds: *leaves*, which store the actual multidimensional data and *internal nodes*, which provide a multiway indexing mechanism.

Internal node entries in xBR$^+$-trees contain entries of the form (*Shape*, *qside*, *DBR*, *Pointer*). Each entry corresponds to a child-node, having a region related to a subquadrant of the original space. *Shape* is a flag that determines if this region is a complete or non-complete square (the area remaining, after one or more splits; explained later in this subsection). This field is heavily used in queries. *qside* is the side length of the subquadrant of the original space that corresponds to this child-node. *DBR* (Data Bounding Rectangle) stores the coordinates of the rectangular subregion of this child-node region that contains point data (at least two points must reside on the sides of the *DBR*), while *Pointer* points to this child-node.

The subquadrant of the original space related to a child-node is expressed by an *Address*. This *Address* (which has a variable size) is not explicitly stored in the xBR$^+$-tree, although it is uniquely determined and can be easily calculated using *qside* and *DBR*. Here, we depict the *Address* only for demonstration purposes. Each *Address* represents a subquadrant that has been produced by Quadtree-like hierarchical subdivision of the current space (of the subquadrant of the original space related to the current node). It consists of a number of directional digits that make up this subdivision. The NW, NE, SW and SE subquadrants of a quadrant are distinguished by the directional digits 0, 1, 2 and 3, respectively. For example, the *Address* 1 represents the NE quadrant of the current space, while the *Address* 12 the SW subquadrant of the NE quadrant of the current space.

The actual region of the child-node is, in general, the subquadrant of its *Address* minus a number of smaller subquadrants, i.e. the ones corresponding to the next entries of the internal node. The entries in an internal node are saved in sequential groups, consisting of subgroups. The first entry of each group is the parental entry of the rest entries of this group. Each entry of a group is a descendant of the entry on its left, or it is the parent of a new (sub)group. To study the contents of a node more easily and understand the mechanism behind subtracting of subregions, it is suggested to examine the node entries from right to left. For example, in Fig. 3.1 an internal node (a root) that points to 5 internal nodes that point to 15 leaves is depicted. The region of the root is the original space, which is assumed to have a quadrangular shape with origin (0,0) on the upper left corner and side length 1. The region of the rightmost entry

(220*) is the NW subquadrant of the SW subquadrant of the SW quadrant of the original space (the * symbol is used to denote the end of a variable size *Address*). The flag *shape* is set at the value 'S' which expresses that this subquadrant is a complete square and thus, no part of its region will be found anywhere in the index, except for the child nodes of the subtree rooted at this entry. The region of the next (on the left) subquadrant is the SW subquadrant of the SW quadrant of the whole space. For this subquadrant, the *Address* is 22* (non-complete square, denoted by 'noS', since the descendent region 220* has been subtracted, while handling an overflow during an insertion, or update). The next two (on the left) entries cover the whole space of the NE quadrant (1*) and the NW quadrant (0*) of the whole space, respectively. Finally, the first entry in the root of this example expresses the whole space minus the four descendant regions (0*, 1*, 22* and 220*), and of course it is a non-complete square area. To further demonstrate the tree structure and the relation between parent region and child, the child of the last root entry (220*) can be examined. This child divides the region addressed by 220* in two parts: the first part is the SW subquadrant of address 220* (denoted by 2*), corresponding to the absolute address 2202*, and the second part is the remaining area of address 220* (denoted by *), after subtracting its SW subquadrant. During a search, or an insertion of a data element with specified coordinates, the appropriate leaf and its region is determined by descending the tree from the root.

External nodes (leaves) of the xBR$^+$-tree simply contain the data elements and have a predetermined capacity C_L. When C_L is exceeded, due to an insertion in a leaf, the region of this leaf is partitioned in two subregions.

An example that demonstrates split of a leaf and an internal node follows. In the upper part of the Fig. 3.2, an xBR$^+$-tree having one internal (root) node with 5 entries (its cardinality equals the maximum capacity of internal nodes, $C_i = 5$) is depicted. The 5 entries point to 5 leaves containing the first 25 points of the total number of 64 points of the dataset of Fig. 3.1. The next (26^{th}) point p must be inserted in a leaf that already contains 6 points and is pointed by the first entry of the root (*) . Since $C_L = 6$, this leaf overflows and is split in two (itself and a new leaf). The new leaf covers the region of the subquadrant 2* and holds 3 points (middle part of Fig. 3.2). The other 4 points remain in the existing leaf (*). An entry for the new leaf (2*) must be inserted in the root node which is already full. The root overflows and is split in two internal nodes (itself and a new node). In order to maintain the cohesion of the tree, a new root node having 2 entries is created. The first entry (*) points to the old root

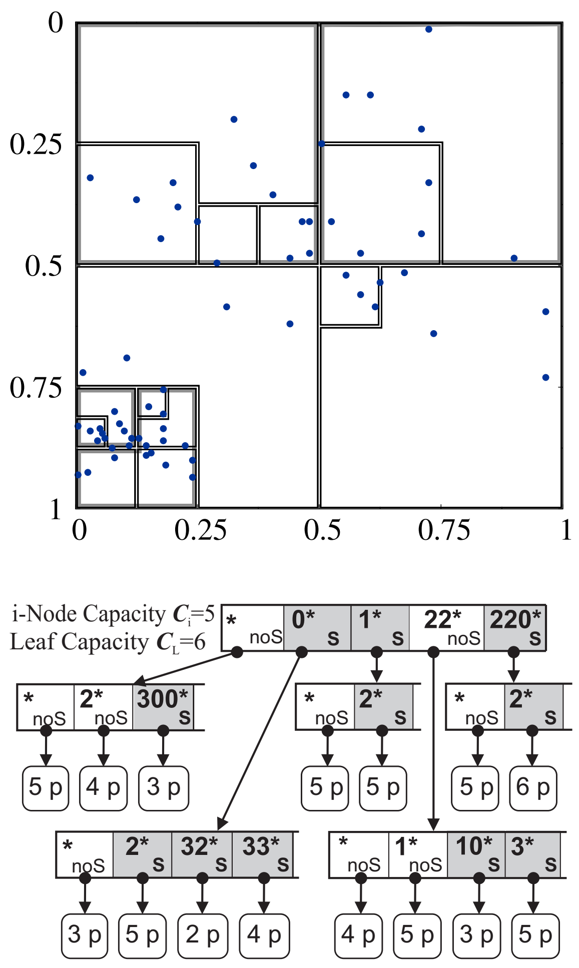


Figure 3.1: A collection of 64 points, its grouping to xBR$^+$-tree nodes and its xBR$^+$-tree.

node and the second entry points to the new node (0*). The resulting xBR$^+$-tree, consisting of 3 internal nodes with 6 entries pointing to 6 leafs, is depicted in the lower part of Fig. 3.2. The final tree, after inserting the rest of the 64 points and the space partitioning of the xBR$^+$-tree are shown in Fig. 3.1. Note that the leaf corresponding to the SW (2*) of the NE (1*) subquadrant of the whole space contains 5 points, since there is one point on the top left corner of this subquadrant. The minimum coordinates of a leaf / internal node region belong to these regions, while the maximum ones do not. This means that regions are closed regarding their left and top borders. Details on the algorithms for splitting leaf and internal nodes appear in [40].

3.3 Algorithms for Batch-Queries Processing

In the following, we present algorithms for processing the batch versions of three common single-dataset queries, using xBR$^+$-trees in SSDs. These algorithms are designed for

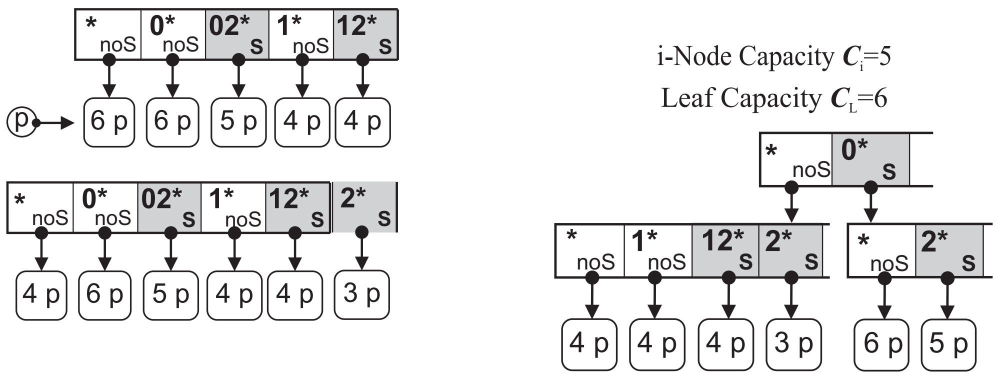


Figure 3.2: Up: an xBR$^+$-tree root pointing to 5 leaves. Middle: an overflown xBR$^+$-tree root pointing to 6 leaves. Down: the resulting xBR$^+$-tree after splitting of the root.

maximizing performance when applied on SSDs. They make use of a main memory area (denoted by M in the following), group read accesses needed by several queries of the batch, reorder the pages to be read and, at the same time, avoid unnecessary re-reading of the same pages and issue massive read operations of large sequences of consecutive pages (exploiting the internal parallelism of SSDs).

3.3.1 Algorithm for Processing of Batch Point-Location Queries

In this subsection, we present our new processing method for Batch Point - Location Queries (*BPLQ*) using xBR$^+$-trees in SSDs. The definition of this query is as follows: Given an index $\mathcal{I}_P$ of a dataset P of points and a set of query points Q, the **BPLQ** returns the largest subset $R \subseteq Q$ such that $R = \{p : p \in Q \wedge p \in P\}$.

The basic idea is as follows. We use the main memory area M (the size of M is defined by the system administrator and its size, a few MBs, is not significant in comparison to the size of the datasets) and we divide Q in subsets such that each subset can be processed within M. Hierarchically, we visit the tree nodes and partition the query points in groups as follows: we examine the node entries from right to left and for each entry, using a variation of quicksort, we partition the query points to the points falling inside the current node entry and the remaining ones; we repeat for the next (to the left) node entry with the remaining points, until we reach the leftmost node entry, or the remaining points are exhausted. This procedure guarantees that each group of points uniquely falls within one subregion of the current node. Next, we massively read the child nodes corresponding to the node entries containing non-empty groups of points into M. This process continues down to the leaf level, where we read the leaves corresponding to the resulting subregions into M. For each leaf, we determine all the query points of Q that exist in this leaf. The algorithm is described in more details as follows.

- Considering the maximum memory size of M available to our program, we calculate the maximum cardinality of each subset of Q that can be processed within M. We divide Q in subsets that do not exceed this maximum cardinality.
- For each of these subsets, we begin at the root.
  - For a subset of query points, we call a procedure that visits a tree node and partitions this subset in groups such that each group uniquely falls within one subregion of this node.
  - If this node points to internal nodes, we calculate and allocate the memory (part of M) that is required for reading the node entries that contain query points and massively read the nodes pointed by these entries.
    - For each of the nodes read and the group of points that fall within the region of this node, we recursively apply this procedure.
  - If a node points to leaf nodes, we calculate and allocate the memory (part of M) that required for reading the leaves that contain query points and massively read the leaves pointed by the node entries.
    - For each of the leaves which have been read and the group of points that fall within this leaf, we sort this group of points according to the axis along which the leaf points have been sorted and determine the query points that exist in the leaf (using a plane-sweep based technique to minimize comparisons).

3.3.2 Algorithm for Processing of Batch Window Queries

Here, we present our processing method for Batch Window Queries (*BWQ*) using xBR$^+$-trees in SSDs.

The definition of this query is as follows: Given an index $\mathcal{I}_P$ of a dataset P and a set of rectangular query windows W, the **BWQ** returns the largest set R that contains pairs of objects (p, w) such that $R = \{(p, w) : p \in P \wedge p$ falls inside $w \in W\}$.

The basic idea (an extension of the method in Section 3.3.1) is as follows. We use a main memory area M and we divide W in subsets such that processing of each subset can be done within M. Hierarchically, we visit the tree nodes and for each node we process the regions of the entries contained within, to create a list of the query windows corresponding to each entry, since each region intersected with a query window may be a candidate for containing points of the pairs of the result (R). For the entries of the current node that contain a non-empty list of query windows, we massively read the nodes corresponding to these entries into M. This process continues down to the leaf level, where we read the leaves corresponding to these entries into M. For each leaf, we determine all the points of P that exist into the leaf and fall inside the regions of the query windows of the list. The algorithm is described in more details as follows.

- Considering the maximum memory size of M available to our program, we calculate the maximum cardinality of each subset of W that can be processed within M. We divide W in subsets that do not exceed this maximum cardinality.

- For each of these subsets of query windows, we begin at the root.

  - For a subset of query windows, we call a procedure that visits a tree node and in each entry of the node we append a list of query windows whose region is intersected with the region of the entry (in Section 3.3.1, we didn't need such lists, since a query point falls in at most one region).

  - If this node points to internal nodes, we calculate and allocate the memory (part of M) that is required for reading the node entries that contain non-empty lists of query windows and massively read the nodes pointed by these entries.

    - For each of the nodes read and the list of query windows that has intersection with the region of this node, we recursively apply this procedure.

  - If a node which has been read points to leaf nodes, we calculate and allocate the memory (part of M) that is required for reading the leaves that contain non-empty lists of query windows and massively read the leaves pointed by the node entries.

    - For each of the leaves read and the list of query windows that have intersection with the region of this leaf we apply the refinement step as follows. We sort this list of windows using as key the maximum coordinate of the axis along which the leaf points have been sorted and determine the leaf points that fall inside the regions of the query windows (using a plane-sweep based technique, to minimize comparisons).

3.3.3 Algorithm for Processing of Batch Distance-Range Queries

In this subsection, we present our processing method for Batch Distance-Range Queries (*BDRQ*) using xBR$^+$-trees in SSDs.

The definition of this query is as follows: Given an index $\mathcal{I}_P$ of a dataset P and a set of query pairs of form (query point, distance threshold) Q, the **BDRQ** returns the largest set R that contains objects $(p,(q,\varepsilon))$ such that $R = \{(p,(q,\varepsilon)) : p \in P, (q,\varepsilon) \in Q \wedge distance(p,q) \leq \varepsilon\}$.

The basic idea is as follows. We utilize a main memory area M. We divide Q in subsets such that processing of each subset can be done within M.

- One method of processing is the following. Every query pair could be seen as a query window with circular schema. Therefore, we can follow the filtering step of the *BWQ* method (presented in Section 3.3.2) down to the leaf level for the *Minimum Bounding Square* (*MBS*) of each query pair. At the leaf level, we apply a refinement step for the leaves and the actual query pairs (which are circles). Hierarchically, we visit the tree nodes and for each node we process the regions of the entries contained within, to create a list of the corresponding query pairs for each entry, since each region intersected with the minimum bounding square of a query pair may be a candidate for containing points of the objects of the result (R). For the entries of the current node that contain a non-empty list of query pairs we massively read the nodes corresponding to these entries into M. This process continues down to the leaf level, where we read the leaves corresponding to these entries into M. For each leaf, we determine all the points of P that exist into the leaf and fall inside the regions of the query pairs of the list. Since, in this method we utilize bounding squares, we call it *BDRQ-B*.

- According to an alternative processing method, every query pair could be seen as the actual circle it represents. Therefore, we can apply the filtering step as follows. Hierarchically, we visit the tree nodes and for each node we process the regions of the entries contained within, to create a list of the corresponding query pairs for each entry. For each entry e of a tree node, we calculate the minimum distance between the region of the entry and the point of the query pair q ($minDist(e, q)$). Every point q with $minDist(e, q) \leq \varepsilon$ is added into the query list of e. It is expected that each region entry having intersection with a query pair may be a candidate to contain points of the objects of the resulting set (R). For the entries of the current node that contain a non-empty list of query pairs, we massively read the nodes corresponding to these entries into M. To simplify the calculations (reduce the execution time) we calculate the square of $minDist$ between a point and the region of an entry and we compare this metric with the square of the given ε. This process continues down to the leaf level, where we read the leaves corresponding to these entries into M. For each leaf, we determine all the points of P that exist into the leaf and fall inside the regions of the query pairs of the list. Since, in this method we utilize minimum distance, we call it *BDRQ-D*.

3.4 Algorithms for Parallel Batch-Queries Processing

In this section, we present the techniques we have utilized, the general idea and the specific algorithms that we developed for processing of the queries studied by taking advantage of the multiple cores of a CPU.

3.4.1 Parallelization techniques

Although, the queries we study are I/O bound (the factor that dominates performance is the cost of accessing secondary storage) and the algorithms we present in Section 3.3 are designed to take advantage of the internal parallelism of SSDs, the CPU cost of processing such queries is not negligible. In this section we further evolve our algorithms so as to utilize the multiple cores of the CPU as much as possible and minimize the time of CPU processing (the cumulative time of CPU processing may be enlarged, since multiple cores are used, however, the actual time of CPU processing is reduced).

Processing of batch PLQs, WQs, or DRQs includes the following operations.

- Filtering at internal nodes, where we determine which query objects (points, windows, circular ranges, or their bounding squares) satisfy the query criterion for the regions of the entries (children) of each internal node.

- Refinement at leaf nodes, where we determine which query objects (points, windows, circular ranges, or their bounding squares) exist (in the case of query points), or include data points (in the other cases of query objects) in each leaf.

If we have to process multiple internal nodes, filtering can be done in parallel, by assigning a number of such nodes to each CPU core. If, however, we have to process one, or a few internal nodes, filtering can still take advantage of parallelism by assigning a number of entries of this (these) node(s) to different cores, during partitioning the query objects according to the query criterion for the regions of these entries.

Accordingly, since in practice during refinement we always have to process multiple leaves, refinement can be done in parallel by assigning a number of leaves to each CPU core.

Moreover, when processing PLQs, query points that fall within the region of a leaf should be sorted by one of their coordinates, before applying plane-sweep to discover which of these points exist within this leaf. We utilize QuickSort which can also be done with parallelism if the number of elements to be sorted exceed a predefined number (through experimentation we concluded that an effective setting for this threshold is 256).

Note that, if each of the available cores is engaged in some kind of parallel processing of the ones mentioned above, using an additional type of parallel processing might not improve the total CPU processing efficiency. For example, if parallel processing between nodes during filtering is employed, using parallel processing between entries also would likely have no positive effect. On the contrary, since employing parallelism has always some overhead, it might have a negative effect. Note also that the effect of each type of parallelism depends on the distribution of the specific dataset and the query being processed.

Therefore, for each type of query, we have tested four configurations for the parallel algorithms we developed. These are depicted in Table 3.1. The 1st configuration employs no parallel processing. However, the algorithm used is the same as in the other configurations (described in the following) and, depending on the dataset and the query, it may be faster due to the absence of overhead.

3.4.2 Basic Ideas of Parallel Query Processing

The algorithms we presented in Section 3.3 work on a depth-first basis. Processing starts at the root and recursively reaches the leaves. In order to better utilize the parallelization

config.	filtering	refinement	sorting
1	no	no	no
2	yes (nodes)	no	yes (>256)
3	yes (partitioning)	yes (nodes)	yes (>256)
4	yes (nodes)	yes (nodes)	yes (>256)

Table 3.1: The configurations of parallel processing tested.

techniques exposed previously, we redesigned our algorithms to work on a breadth-first basis as much as possible. This means that each algorithm processes (exploiting multiple cores) as many nodes of each level as possible, before proceeding to the next level. “As many $\cdots$ as possible” is related to the main memory available. Since main memory is limited, loading the data for a whole level of the tree might not be possible. Therefore, considering the memory limit, for each level (iteratively) we load as many nodes and as many query objects as possible, process them in parallel and continue processing in-depth (recursively), before returning to the level where there are more nodes waiting to be processed. This idea is applied on every new tree level we visit.

Our technique employs a combination of breadth-first and depth-first processing to exploit parallelism. Therefore, we call the technique we use as *restricted breadth depth-first processing*. We believe that this technique can also be applied to other algorithms working on trees, so as to take advantage of parallel processing.

3.4.3 Parallel Algorithm for Processing of Batch Point-Location Queries

In this subsection, we present our new processing method for Parallel Batch Point-Location Queries (*PBPLQ*) using xBR$^+$-trees in SSDs. The basic idea is as follows. We use two buffers with predefined sizes by the system administrator, the node buffer M_N and the query point buffer M_Q. We divide Q in subsets such that each subset can be processed within M_Q. For each tree level, starting at the root level, we massively read as many nodes of the current level as possible into M_N. Using parallelism between nodes, or between entries of nodes (depending on the number of nodes read), we partition the query points in groups and distinguish the ones that uniquely fall within a subregion (region of a child entry) of the nodes read. Next, we massively read the child nodes corresponding to the resulting subregions (belonging to possibly multiple nodes of the same level) into M_N. This process continues recursively down to the leaf level, where we read the leaves (children of possibly multiple nodes of the same level) corresponding to the resulting subregions into M_N. We process leaves in parallel and, for each leaf, we determine all the query points of Q that exist in this leaf. If the number of query points to be examined in such a leaf is large enough, parallel sorting can also be employed. As recursive calls return, as many nodes as possible of the respective level that have not been processed yet are loaded into M_N and the process is repeated for them.

The algorithm is described in more details in Alg. 1. To be able to handle multiple nodes of the same level, using of stacks is needed. Although, a single stack could be used, for efficiency of copying operations of consecutive multiple stack records, we use two synchronized stacks, a stack of pointers to nodes, called S_N, and a stack holding information of query points groups, called S_{Qi}.

Algorithm 1 Algorithm PBPLQ

```
Input: Stack of pointers to nodes,=$S_N$, Stack of query points groups=$S_Q i$
Output: PBPLQ Result=Result
 1: $M_Q \leftarrow$ readQueryPoints();
 2: $S_Q i$.push($M_Q$);
 3: $M_N \leftarrow$ readRoot();
 4: $S_N$.push($M_N$);
 5: if $\#nodes > threshold$ then
 6:     for r in $S_N$ do      // Parallel For
 7:         createStack($S_N r$);
 8:         createStack($S_Q ir$);
 9:         for e in r.nodes() do
10:             partitionInParallel(e.queryPoints());
11:             if points.intersect(e.region) then
12:                 $S_N r$.push(e.childNode);
13:                 $S_Q ir$.push(e.points);
14:         $S_N$.merge($S_N r$);
15:         $S_Q i$.merge($S_Q ir$);
16:         if level is leafParentLevel then
17:             while $S_N$.size>0 or $S_Q i$.size>0 do
18:                 transferAndReplacePointedLeavesInto($M_N$);
19:                 if $\#leaves > threshold$ then
20:                     for each leaf in parallel do
21:                         sort(leaf.points);
22:                         Result $\leftarrow$ PlaneSweep(leaf.points)
23:                 else
24:                     transferAndReplacePointedLeavesInto($M_N$);
25:                     callRecursivelyPBPLQ($S_N$,$S_Q i$);
```

3.4.4 Parallel Algorithm for Processing of Batch Window and Distance-Range Queries

In this subsection, we present our processing methods, using xBR$^+$-trees in SSDs, for Parallel Batch Window Queries (*PBWQ*) and our Parallel Batch Distance-Range Queries based on circular ranges (*PBDRQ-D*), or on range bounding squares (*PBDRQ-B*).

The basic idea (an extension of the method in Section 3.4.3) is as follows. We use two buffers with predefined sizes by the system administrator, the node buffer M_N and the query object (window, circular range, or bounding square) buffer M_Q. We divide Q in subsets such that each subset can be processed within M_Q. For each tree level, starting at the root level, we massively read as many nodes of the current level as possible into M_N. Using parallelism between nodes, or between entries of nodes (depending on the number of nodes read), we partition the query objects in groups and distinguish the ones that intersect with a subregion of the nodes read, since each subregion intersected with a query object may be a candidate for containing points of the pairs of the result (R). Next, we massively read the child nodes corresponding to the intersected subregions (belonging to possibly multiple nodes of the same level) into M_N. This process continues recursively down to the leaf level, where we read the leaves (children of possibly multiple nodes of the same level) corresponding to the resulting subregions into M_N. We process leaves in parallel and, for each leaf, we determine the points of this leaf that fall inside any of the query objects of Q. As recursive calls return, as many nodes as possible of the respective level that have not been processed yet are loaded into M_N and the process is repeated for them. The algorithm is described in more details as follows.

Again, we use two synchronized stacks, a stack of pointers to nodes, called S_N, and a stack holding information of query object (window, circular range, or bounding square) groups, called S_{Qi} and two buffers with predefined sizes, the node buffer M_N and the query object buffer M_Q (Alg. 2).

3.5 Experimental Results

We run a large set of experiments to compare the repetitive application of the existing algorithms for processing batch queries to the new algorithms, designed especially for batch queries in SSDs, which either use one [46], or multiple CPU cores.

Algorithm 2 Algorithm PBPLQ / PBDRQ-D / PBDRQ-B

```
Input: Stack of pointers to nodes,=$S_N$, Stack of query points groups=$S_Q i$
Output: PBPLQ / PBDRQ-D / PBDRQ-B Result=Result
 1: $M_Q \leftarrow$ readQueryObject();
 2: sort($M_Q$);
 3: $S_Q i$.push($M_Q$);
 4: $M_N \leftarrow$ readRoot();
 5: $S_N$.push($M_N$);
 6: if $\#nodes > threshold$ then
 7:     for r in $S_N$ do        // Parallel execution
 8:         createStack($S_N r$);
 9:         createStack($S_Q ir$);
10:         for e in r.nodes() do        // Parallel execution
11:             if queryObjectsIntersect(e.region) then
12:                 $S_N r$.push(e.childNode);
13:                 $S_Q ir$.push(e.points);
14:         $S_N$.merge($S_N r$);
15:         $S_Q i$.merge($S_Q ir$);
16:         if level is leafParentLevel then
17:             while $S_N$.size>0 or $S_Q i$.size>0 do
18:                 transferAndReplacePointedLeavesInto($M_N$);
19:                 if $\#leaves > threshold$ then
20:                     for each leaf do        // Parallel execution
21:                         Result $\leftarrow$ PlaneSweep(leaf.points)
22:                 else
23:                     transferAndReplacePointedNodesInto($M_N$);
24:                     callAlgorithmRecursively($S_N$,$S_Q i$);
```

We used real spatial datasets of North America representing roads (NArdN with 569082 line-segments) and rail-roads (NArrN with 191558 line-segments). To create sets of 2d points, we transformed the MBRs of line-segments from NArdN and NArrN into points by taking the center of each MBR (i.e. |NArdN| = 569082 points, |NArrN| = 191558 points). Moreover, to get the double amount of points from NArdN, we chose the two points with *min* and *max* coordinates of the MBR of each line-segment (i.e. we created a new dataset, |NArdND| = 1138164 points. The data of these three files were normalized in the range $[0, 1]^2$. We have also created synthetic clustered datasets of 250000, 500000 and 1000000 points, with 125 clusters in each dataset (uniformly distributed in the range $[0, 1]^2$), where for a set having N points, $N/125$ points were gathered around the center of each cluster, according to Gaussian distribution. We also used three big real datasets (retrieved from `http://spatialhadoop.cs.umn.edu/datasets.html`), which represent water resources of North America (Water) consisting of 5836360 line-segments and world parks or green areas (Park) consisting of 11503925 polygons and world buildings (Build) consisting of 114736539 polygons. To create sets of points, we used the centers of the line-segment MBRs from Water and the centroids of polygons from Park and Build.

The C programming language was used for implementing the algorithms and the OpenMP library was used for implementing parallelism. All experiments were performed on a Dell Precision T3500 workstation, running CentOS Linux 7 with Kernel 4.15.4 and equipped with a quad-core Intel Xeon W3550 CPU (supporting Hyper-Threading technology, with 8 logical cores), 8GB of main memory, an 1TB 7.2K SATA-3 Seagate HDD used for the operating system and a 512GB SM951A Samsung SSD hosted on PCI-e 2.0 interface, storing our executables and data. Since our algorithms are especially designed for maximizing performance when applied on SSDs, the xBR$^+$-tree was stored on the SSD of our system. However, we tested storing our structure on the HDD, too and obtained execution times 2 orders of magnitude larger than the ones on the SSD. Therefore, we present results of execution on the SSD only.

We run experiments for studying the performance of existing and new algorithms for processing batches of PLQs, WQs and DRQs (DRQs were processed in two versions, using an MBS and the actual circular range). We tested batches consisting of $2^{17}, 2^{18}, 2^{19}$ and 2^{20} queries. We also tested tree node sizes equal to 4KB, 8KB and 16KB. In each experiment, we counted actual disk accesses and total execution time.

The existing algorithms answer batch queries by repetitive application for each query of the batch (One-by-One, or ObO, execution) with and without the use of LRU buffer equal to 256 internal nodes and 256 leaf nodes. This discrimination of the two parts of LRU buffer is necessary, since internal nodes are significantly fewer and a common LRU buffer would get frequently emptied from internal nodes, although the same internal nodes are more likely to be needed for separate queries of the batch. Our experiments showed that this buffer size is adequate for maximizing performance, even for the largest trees tested. The maximum size of M was comparable to LRU size (although, in many cases this maximum size was not utilized by the algorithms studied). Following the results presented in [46], in this chapter we present only results for ObO execution with LRU enabled.

Therefore, for each query (PLQ, WQ and DRQ in two versions), we tested LRU ObO, the respective new, SSD optimized, single-core algorithm and the respective new multi-core algorithm (using 2, 4 and 8 cores and 4 configurations for each case). The total number of experiments performed equals 6048 (combinations of 9 datasets, 3 node sizes, 4 batch sizes, 4 queries and 14 algorithmic versions for each query). Due to space limitations, in the following we present indicative (or, a limited part of the) experimental results, expressing the general trends found. In all experiments performed, the query batches used contain 2^{17}, 2^{18}, 2^{19} and 2^{20} query objects (column Q in tables with experimental results).

3.5.1 PLQ Experiments

To study PLQs, we created batches consisting of 50% existing and 50% non-existing points in each dataset. Both existing and non-existing points cover the whole indexed space.

In Table 3.2, for the 500KC dataset and 3 page sizes, we depict the gain (as a percentage) of BPLQ over LRU ObO, regarding disk accesses. We also depict the absolute total execution times (in ms) of LRU ObO and BPLQ and the execution time gain (as a percentage) of BPLQ over LRU ObO. Note that the gain is defined (for both metrics) as the fraction of the difference of performance of the second and the first algorithms over the performance of the second algorithm (gain $=\frac{LRUObO-BPLQ}{LRUObO}$) and expresses the speedup obtained by first algorithm, in relation to the time cost of the first algorithm.

Q	**Disk Read Acc**			**Exec Time**								
	4KB	8KB	16KB	4KB			8KB			16KB		
#	gain	gain	gain	time	time	gain	time	time	gain	time	time	gain
BPLQ vs LRU ObO												
2^{17}	70.6%	72.6%	64.4%	1995	163.4	91.8%	1530	164.0	89.3%	1183	180.2	84.8%
2^{18}	71.8%	73.2%	64.8%	2428	273.5	88.7%	2099	283.6	86.5%	1790	323.2	81.9%
2^{19}	72.1%	73.3%	64.9%	3135	486.7	84.5%	3268	518.1	84.1%	2994	592.7	80.2%
2^{20}	74.5%	75.4%	68.1%	4666	925.8	80.2%	4776	986.2	79.3%	5506	1140	79.3%

Table 3.2: PLQ: (Disk Accesses % performance gain and Total Exec. Time absolute values and % time performance gain) BPLQ vs LRU ObO, for the 500KC dataset.

As another indicative result set, in Table 3.3, we depict analogous data for the Park dataset. It is evident from both tables that BPLQ saves a significant number of disk accesses over LRU ObO and it is even more efficient regarding execution time. Analogous remarks can be made for the other dataset cases.

Q	**Disk Read Acc**			**Exec Time**								
	4KB	8KB	16KB	4KB			8KB			16KB		
#	gain	gain	gain	time	time	gain	time	time	gain	time	time	gain
BPLQ vs LRU ObO												
2^{17}	32.0%	34.7%	36.7%	1039	454.7	56.3%	1095	451.6	58.8%	1469	530.9	63.9%
2^{18}	33.7%	35.9%	37.1%	1585	668.5	57.8%	1709	680.0	60.2%	2431	845.5	65.2%
2^{19}	34.4%	36.1%	37.0%	2566	1057	58.8%	2870	1092	61.9%	4232	1413	66.6%
2^{20}	37.7%	39.0%	40.6%	4568	1818	60.2%	5065	1914	62.2%	7918	2562	67.6%

Table 3.3: PLQ: (Disk Accesses % performance gain and Total Exec. Time absolute values and % time performance gain) BPLQ vs LRU ObO, for the Park dataset.

In Tables 3.4 and 3.5, for the same datasets as the ones of Tables 3.2 and 3.3, respectively, we depict the total execution time gain (as a percentage) of PBPLQ over BPLQ, for 3 page sizes and 2, 4 and 8 cores used (for each number of cores, this gain has been calculated using the best of the 4 configurations of Table 3.1). Studying these results, it can be noted that the parallel algorithms tend to maximize their gain over BPLQ for the larger page size. For both datasets depicted, the parallel algorithms are clearly faster than BPLQ and justify their use, although the query studied is I/O bound.

Q	**Exec Time Gain**								
	2 cores			4 cores			8 cores		
#	4KB	8KB	16KB	4KB	8KB	16KB	4KB	8KB	16KB
PBPLQ vs BPLQ									
2^{17}	07.1%	10.1%	13.6%	17.1%	23.8%	30.6%	14.7%	23.9%	21.7%
2^{18}	09.5%	11.4%	14.2%	21.0%	24.4%	32.6%	18.8%	24.9%	27.0%
2^{19}	09.9%	11.5%	13.8%	21.3%	25.2%	32.0%	21.2%	24.2%	24.7%
2^{20}	10.5%	12.0%	14.8%	21.3%	25.9%	32.3%	20.5%	25.7%	25.8%

Table 3.4: PLQ: (Total Exec. Time % performance gain) PBPLQ vs BPLQ, for the 500KC dataset.

Q	**Exec Time Gain**								
	2 cores			4 cores			8 cores		
#	4KB	8KB	16KB	4KB	8KB	16KB	4KB	8KB	16KB
PBPLQ vs BPLQ									
2^{17}	14.9%	10.8%	19.4%	18.0%	13.6%	26.1%	17.7%	15.9%	23.6%
2^{18}	20.2%	12.1%	23.2%	20.5%	14.9%	30.7%	19.1%	17.8%	29.0%
2^{19}	18.1%	13.3%	25.9%	21.2%	18.6%	31.8%	21.2%	20.5%	30.4%
2^{20}	19.4%	17.9%	29.3%	22.1%	25.1%	37.0%	23.3%	22.7%	32.8%

Table 3.5: PLQ: (Total Exec. Time % performance gain) PBPLQ vs BPLQ, for the Park dataset.

In Fig. 3.3, for the larger page size, we depict the smaller, larger and average total execution time gain (represented by the lower, upper and middle dash of each vertical line, respectively) of the parallel algorithms over BPLQ for all datasets, small real datasets, small synthetic datasets and big real datasets. The top, middle and lower diagram corresponds to parallel algorithms utilizing 2, 4 and 8 cores, respectively. These diagrams further justify that it is worth utilizing the parallel algorithms and making use of the multiple CPU cores for processing batch PLQs. For example, for 8 cores, considering all datasets, the minimum gain is over 10% and the maximum gain is close to 40%. This is important, considering that the PLQ is I/O bound.

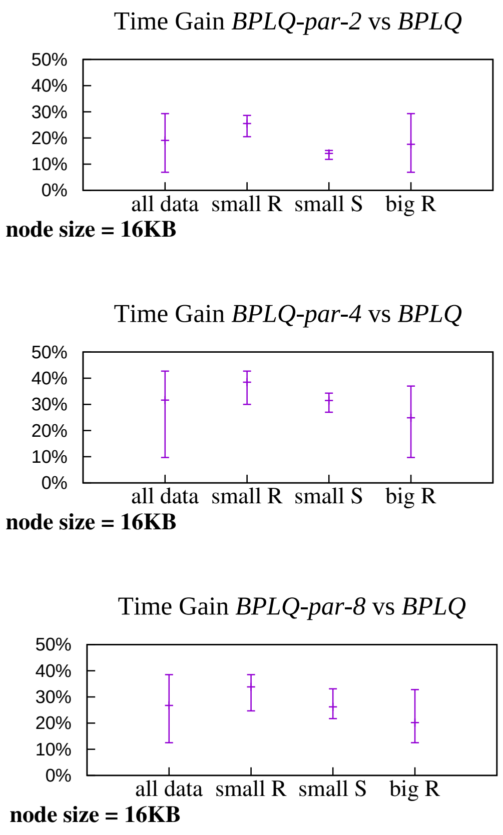


Figure 3.3: PLQ: min, max and average gain of PBPLQ vs BPLQ.

3.5.2 WQ Experiments

To study WQs, we created batches with query windows that cover the whole indexed space. Tables 3.6 and 3.7 are analogous to Tables 3.2 and 3.3, for the NArdND and Water datasets, respectively. Depending on the dataset, BWQ saves disk accesses over LRU ObO, while for both datasets and it is significantly more efficient than LRU ObO regarding execution time.

Tables 3.8 and 3.9 are analogous to Tables 3.4 and 3.5, for the NArdND and Water datasets, respectively. Studying these results, we reach analogous conclusions the ones for PLQ: the parallel algorithms maximize their gain over BWQ for the larger page size. For both datasets depicted, the parallel algorithms are clearly faster than BWQ and justify their use, although the query studied is I/O bound.

Q	Disk Read Acc			Exec Time								
	4KB	8KB	16KB	4KB			8KB			16KB		
#	gain	gain	gain	time	time	gain	time	time	gain	time	time	gain
BWQ vs LRU ObO												
2^{17}	4.4%	12.2%	26.3%	1447	386.0	73.3%	1322	271.1	79.5%	1192	246.8	79.3%
2^{18}	4.4%	12.2%	26.3%	1927	483.9	74.9%	2064	445.6	78.4%	1900	433.2	77.2%
2^{19}	4.4%	12.2%	26.4%	2689	789.0	70.7%	2890	836.4	71.1%	2747	820.9	70.1%
2^{20}	4.4%	12.2%	26.4%	4145	1499	63.8%	5781	1612	72.1%	4806	1598	66.7%

Table 3.6: WQ: (Disk Accesses % performance gain and Total Exec. Time absolute values and % performance gain) BWQ vs LRU ObO, for the NArdND dataset.

Q	Disk Read Acc			Exec Time								
	4KB	8KB	16KB	4KB			8KB			16KB		
#	gain	gain	gain	time	time	gain	time	time	gain	time	time	gain
BWQ vs LRU ObO												
2^{17}	0.2%	0.2%	0.3%	1841	673.9	63.4%	1387	597.5	56.9%	1369	630.8	53.9%
2^{18}	0.1%	0.1%	0.2%	3011	1033	65.7%	2254	913.0	59.5%	2313	1044	54.9%
2^{19}	0.1%	0.1%	0.2%	4630	1516	67.3%	3476	1344	61.3%	3802	1717	54.8%
2^{20}	0.1%	0.1%	0.2%	6906	2248	67.4%	5062	2062	59.3%	6299	2908	53.8%

Table 3.7: WQ: (Disk Accesses % performance gain and Total Exec. Time absolute values and % performance gain) BWQ vs LRU ObO, for the Water dataset.

Q	**Exec Time Gain**								
	2 cores			4 cores			8 cores		
#	4KB	8KB	16KB	4KB	8KB	16KB	4KB	8KB	16KB
PBWQ vs BWQ									
2^{17}	06.5%	05.3%	10.0%	09.2%	10.8%	18.3%	08.5%	10.3%	17.6%
2^{18}	10.3%	07.1%	10.4%	14.2%	12.7%	19.6%	13.1%	11.6%	18.4%
2^{19}	11.5%	10.5%	11.8%	16.8%	14.6%	21.6%	16.4%	13.5%	19.8%
2^{20}	14.6%	09.9%	13.4%	19.5%	15.7%	23.1%	18.3%	14.9%	21.4%

Table 3.8: WQ: (Total Exec. Time % performance gain) PBWQ vs BWQ, for the NArdND dataset.

Q	**Exec Time Gain**								
	2 cores			4 cores			8 cores		
#	4KB	8KB	16KB	4KB	8KB	16KB	4KB	8KB	16KB
PBWQ vs BWQ									
2^{17}	04.0%	05.1%	08.7%	06.1%	08.2%	12.3%	05.1%	06.4%	12.8%
2^{18}	06.7%	08.7%	15.6%	08.0%	12.8%	19.5%	07.3%	11.6%	20.0%
2^{19}	09.0%	13.2%	20.8%	11.0%	17.7%	26.1%	10.7%	17.2%	25.9%
2^{20}	11.1%	17.1%	22.9%	14.8%	23.1%	29.5%	13.4%	22.4%	28.9%

Table 3.9: WQ: (Total Exec. Time % performance gain) PBWQ vs BWQ, for the Water dataset.

Fig. 3.4 is analogous to Fig. 3.3. The diagrams of this figure, like in the case of PLQ, further justify that it is worth utilizing the parallel algorithms and making use of the multiple CPU cores for processing batch WQs. For example, for 8 cores, considering all datasets, the minimum gain is positive, the average gain is over 20% and the maximum gain is around 35%.

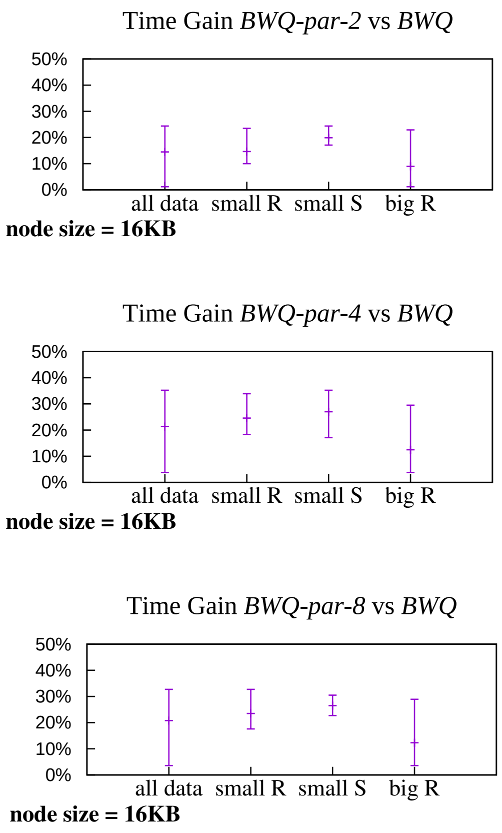


Figure 3.4: WQ: min, max and average gain of PBWQ vs BWQ.

3.5.3 DRQ Experiments

To study DRQs, we created batches with query windows that cover the whole indexed space. We run experiments for both BDRQ-D and BDRQ-B and the respective parallel versions.

Tables 3.10 / 3.12 and 3.11 / 3.13 are analogous to Tables 3.2 and 3.3, for the 1000KC / NArd and Build / Park datasets, respectively. Depending on the dataset, BDRQ-D / PBDRQ-B saves disk accesses over LRU ObO, while for both datasets and it is significantly more efficient than LRU ObO regarding execution time.

Q	Disk Read Acc			Exec Time								
	4KB	8KB	16KB	4KB			8KB			16KB		
#	gain	gain	gain	time	time	gain	time	time	gain	time	time	gain
BDRQ-D vs LRU ObO												
2^{17}	35.9%	45.6%	50.6%	2054	298.8	85.4%	1762	289.8	83.6%	1732	316.1	81.8%
2^{18}	35.4%	45.1%	50.3%	2497	475.3	81%	2288	502.5	78%	2427	559.4	76.9%
2^{19}	35.1%	44.9%	50.2%	3702	882.7	76.2%	3437	949.5	72.4%	4616	1091	76.4%
2^{20}	35.0%	44.9%	50.2%	5182	1751	66.2%	5715	1871	67.3%	7026	2515	64.2%

Table 3.10: DRQ: (Disk Accesses % performance gain and Total Exec. Time absolute values and % performance gain) BDRQ-D vs LRU ObO, for the 1000KC dataset.

Q	Disk Read Acc			Exec Time								
	4KB	8KB	16KB	4KB			8KB			16KB		
#	gain	gain	gain	time	time	gain	time	time	gain	time	time	gain
BDRQ-D vs LRU ObO												
2^{17}	04.1%	03.1%	01.7%	20314	3021	85.1%	13631	2576	81.1%	9021	2174.0	75.9%
2^{18}	12.1%	09.2%	05.8%	35928	4725	86.8%	23801	4038	83%	15175	3440.4	77.3%
2^{19}	25.4%	21.0%	14.1%	66770	7136	89.3%	42339	6129	85.5%	26075	5059	80.6%
2^{20}	45.3%	39.6%	28.4%	127312	9919	92.2%	77916	8931	88.5%	44075	7285	83.5%

Table 3.11: DRQ: (Disk Accesses % performance gain and Total Exec. Time absolute values and % performance gain) BDRQ-D vs LRU ObO, for the Build dataset.

Q	**Disk Read Acc**			**Exec Time**								
	4KB	8KB	16KB	4KB			8KB			16KB		
#	gain	gain	gain	time	time	gain	time	time	gain	time	time	gain
BDRQ-B vs LRU ObO												
2^{17}	11.3%	24.9%	32.5%	944.3	216.6	77.1%	830.2	201.0	75.8%	852.9	229.8	73.1%
2^{18}	11.4%	25.0%	34.2%	1260	359.7	71.4%	1181	371.6	68.5%	1352	446.4	67.0%
2^{19}	11.4%	25.0%	34.2%	1851	695.5	62.4%	1875	716.8	61.8%	2374	1025	56.8%
2^{20}	11.5%	25.0%	34.2%	3049	1424	53.3%	3273	1437	56.1%	5134	2128	58.6%

Table 3.12: DRQ: (Disk Accesses % performance gain and Total Exec. Time absolute values and % performance gain) BDRQ-B vs LRU ObO, for the NArd dataset.

Q	**Disk Read Acc**			**Exec Time**								
	4KB	8KB	16KB	4KB			8KB			16KB		
#	gain	gain	gain	time	time	gain	time	time	gain	time	time	gain
BDRQ-B vs LRU ObO												
2^{17}	02.9%	03.6%	05.0%	3546	1094	69.1%	2607	926.5	64.5%	2466	918.2	62.8%
2^{18}	03.3%	03.2%	03.9%	5664	1613	71.5%	4297	1305	69.6%	3995	1408	64.8%
2^{19}	07.1%	03.9%	03.9%	8900	2299	74.2%	6284	1794	71.5%	6409	2123	66.9%
2^{20}	14.9%	08.4%	04.7%	13111	3273	75.0%	9194	2627	71.4%	10143	3598	64.5%

Table 3.13: DRQ: (Disk Accesses % performance gain and Total Exec. Time absolute values and % performance gain) BDRQ-B vs LRU ObO, for the Park dataset.

Tables 3.14 / 3.16 and 3.15 / 3.17 are analogous to Tables 3.4 and 3.5, for the 1000KC / NArd and Build / Park datasets, respectively. Studying these results, we reach analogous conclusions the ones for PLQ: the parallel algorithms maximize their gain over BDRQ-D / BDRQ-B for the larger page size. For all datasets depicted, the parallel algorithms are clearly faster than BDRQ-D / BDRQ-B and justify their use, although the query studied is I/O bound.

Q	**Exec Time Gain**								
	2 cores			4 cores			8 cores		
#	4KB	8KB	16KB	4KB	8KB	16KB	4KB	8KB	16KB
PBDRQ-D vs BDRQ-D									
2^{17}	09.8%	12.7%	19.9%	18.6%	21.1%	32.5%	16.3%	18.8%	26.9%
2^{18}	12.0%	15.0%	21.2%	20.6%	24.6%	34.7%	19.1%	20.0%	27.8%
2^{19}	15.7%	16.7%	23.7%	25.4%	26.4%	36.4%	23.7%	21.8%	29.0%
2^{20}	20.3%	19.3%	35.6%	28.7%	28.4%	44.3%	25.0%	22.9%	38.3%

Table 3.14: DRQ: (Total Exec. Time % performance gain) PBDRQ-D vs BDRQ-D, for the 1000KC dataset.

Q	**Exec Time Gain**								
	2 cores			4 cores			8 cores		
#	4KB	8KB	16KB	4KB	8KB	16KB	4KB	8KB	16KB
PBDRQ-D vs BDRQ-D									
2^{17}	05.6%	04.8%	03.1%	07.2%	06.4%	05.8%	06.9%	04.9%	04.5%
2^{18}	04.9%	05.4%	03.9%	08.0%	07.7%	08.3%	06.8%	07.7%	06.9%
2^{19}	07.4%	07.1%	05.7%	10.7%	10.1%	11.0%	08.6%	09.7%	09.6%
2^{20}	08.9%	09.8%	08.4%	12.2%	14.0%	16.1%	11.5%	14.1%	15.3%

Table 3.15: DRQ: (Total Exec. Time % performance gain) PBDRQ-D vs BDRQ-D, for the Build dataset.

Q	**Exec Time Gain**								
	2 cores			4 cores			8 cores		
#	4KB	8KB	16KB	4KB	8KB	16KB	4KB	8KB	16KB
PBDRQ-B vs BDRQ-B									
2^{17}	05.7%	11.1%	18.1%	15.9%	14.7%	27.1%	05.7%	11.1%	18.1%
2^{18}	07.9%	11.9%	19.7%	17.9%	15.6%	30.8%	07.9%	11.9%	19.7%
2^{19}	13.3%	13.9%	30.2%	21.5%	18.3%	42.0%	13.3%	13.9%	30.2%
2^{20}	17.6%	17.0%	34.0%	25.7%	20.3%	43.8%	17.6%	17.0%	34.0%

Table 3.16: DRQ: (Total Exec. Time % performance gain) PBDRQ-B vs BDRQ-B, for the Nard dataset.

Q	**Exec Time Gain**								
	2 cores			4 cores			8 cores		
#	4KB	8KB	16KB	4KB	8KB	16KB	4KB	8KB	16KB
PBDRQ-B vs BDRQ-B									
2^{17}	04.3%	01.4%	04.8%	05.3%	04.8%	06.1%	04.3%	01.4%	04.8%
2^{18}	05.9%	02.2%	06.3%	07.9%	06.0%	07.8%	05.9%	02.2%	06.3%
2^{19}	09.7%	03.9%	09.0%	12.7%	09.9%	10.8%	09.7%	03.9%	09.0%
2^{20}	14.1%	11.0%	15.6%	17.8%	17.2%	17.7%	14.1%	11.0%	15.6%

Table 3.17: DRQ: (Total Exec. Time % performance gain) PBDRQ-B vs BDRQ-B, for the Park dataset.

Fig. 3.5 / 3.6 is analogous to Fig. 3.3. The diagrams of this figure, like in the case of PLQ, further justify that it is worth utilizing the parallel algorithms and making use of the multiple CPU cores for processing batch DRQs. For example, for 8 cores and PBDRQ-D / PBDRQ-B, considering all datasets, the minimum gain is positive, the average gain is around 35% / 20% and the maximum gain is over 40% / 35%.

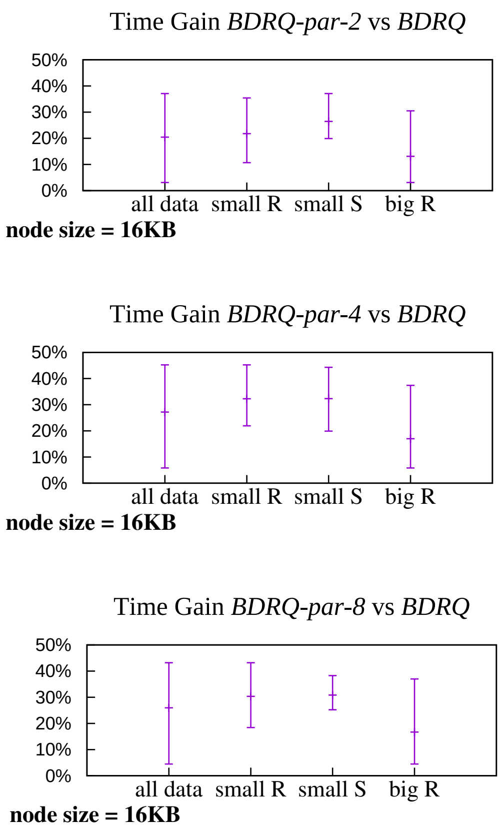
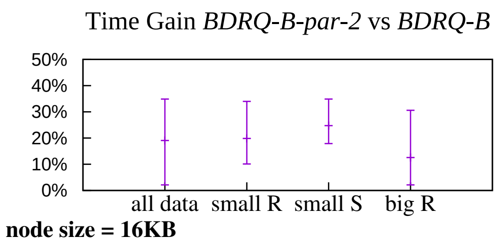


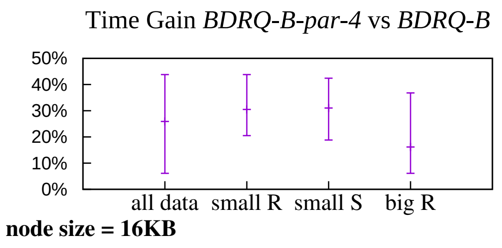
Figure 3.5: WQ: min, max and average gain of PBWQ vs BWQ.

3.6 Processing in a Distributed Environment

So far, we presented centralized algorithms that take advantage both of SSDs and multiple CPU cores, to accelerate spatial query processing. If the data to be processed is too large to be handled in a centralized system, a distributed computing environment could be utilized. More specifically, a cluster of computers interconnected through a fast LAN could handle data which are larger by orders of magnitude.

For several problems, related algorithms cannot be easily transferred to such a shared-nothing computing environment. However, this is not the case for the algorithms we have presented. Due to the nature of the queries studied, these algorithms can be easily applied





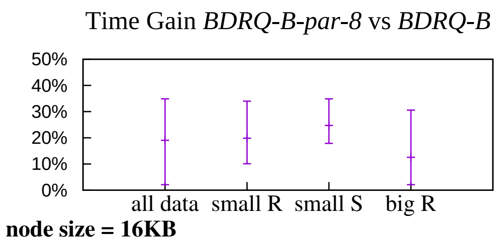


Figure 3.6: WQ: min, max and average gain of PBWQ vs BWQ.

in such a distributed setting. If data is distributed among nodes, each node could index its data using an xBR$^+$-tree, based on SSDs. The query batch would have to be transmitted to every node of the cluster, each node would compute the answer, as far as the data it stores are concerned, and the results from all nodes could be merged in a node that will act as a coordinator of query processing. This is possible, since the answer of any of the queries we studied for a node would be independent to the answers for other nodes.

Although, this approach will work, efficiency requires that data are not blindly distributed among nodes. Each node should keep data that are spatially close, even in case partial intersection between the areas covered by nodes is allowed. In the case, the coordinating node should transmit to each computing node only the part of the query batch that spatially intersects the area of this node. Therefore, the coordinating node should be aware of the areas covered by computing nodes.

Such processing could be embedded in a parallel and distributed system, like SpatialHadoop (`http://spatialhadoop.cs.umn.edu/`) that already incorporates spatial indexing methods and distributes data to nodes according to such a method. In [67,68], spatial query processing techniques have been added to SpatialHadoop. We could build on [67, 68] to embed into Spatiahadoop SSD-based xBR$^+$-trees that process spatial queries, taking advantage of multiple cores.

3.7 Conclusions

In this chapter, extending the algorithms presented in [46], for the first time in the literature, we present algorithms for common spatial batch queries on single datasets, using xBR$^+$-trees in SSDs that take advantage of multiple cores. Processing of spatial queries in SSDs has not received considerable attention in the literature, so far. Even more, the utilization of multiple cores, based on a combination of breadth-first and depth-first tree traversals, is a new approach that further accelerates processing. The algorithms proposed in [46] exploit the massive I/O advantages of SSDs and outperform the repetitive application of existing algorithms by exploiting the massive I/O advantages of SSDs, both regarding actual disk access and execution time, even if the I/O of existing algorithms are assisted by LRU buffering. The parallel extensions of these algorithms clearly outperform the ones of [46], although the three queries studied are I/O bound. The new algorithms can be applied to a parallel and distributed environment and deal with very big data. The algorithms documented in this chapter were published in [69].

Chapter 4

$k-$NN Query Processing with GPU and RAM

The k Nearest Neighbor (k-NN) algorithm is widely used for classification in many problems areas (medicine, economy, entertainment, etc.). For example, k-NN classification has been used for economic forecasting, including bankruptcy prediction. [70] present a model for bankruptcy prediction using adaptive fuzzy k-NN, where k and the fuzzy strength parameter are adaptively specified by particle swarm optimization, while [71] use k-NN for predicting financial distress (a key factor for bankruptcy).

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, it can be proved challenging to deliver low latency results and GPU-based techniques may 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.

Processing of big spatial data is demanding, and it is often assisted by parallel processing. GPU-based parallel processing has become very popular during last years [72]. In general, GPU devices have much larger numbers of processing cores than CPUs and device memory which is faster than main memory accessed by CPUs, providing high computing capabilities even to commodity computers.

In this chapter, we propose and implement three in-memory GPU-based algorithms for the k-NN query, using the CUDA API [73] and the Thrust library [72]. The first one is Brute-force based, the second one is using heuristics to minimize the reference points near a query point and the third one is using an in-memory partitioning algorithm. The first two algorithms use extensively the Thrust Library. The third one doesn't have external library dependencies and is designed to run as a kernel function, in the GPU device. More over we have implemented two different list buffers that facilitate the temporary point distance storage, needed by our method.

We also present an extensive experimental comparison against existing algorithms, using synthetic and real datasets. The results show that our algorithms outperform these algorithms, in terms of execution time as well as total volume of in-memory reference points that can be handled.

The rest of this chapter is organized as follows. In Section 4.1, we review related work and present the motivation for our work. Next, in Section 4.2, we present the new algorithms that we developed for the k-NN GPU-based Processing and in Section 4.3, we present the experimental study that we performed for studying the performance of our algorithms and for comparing them to their predecessors. Finally, in Section 4.4, we present the conclusions arising form our work.

4.1 Related Work and Motivation

In this section, we review the most representative algorithms to solve k-NN queries in GPU. k-NN is typically implemented on GPUs using *brute force* (BF) methods applying a two-stage scheme: (1) the computation of distances and (2) the selection of the nearest neighbors. 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.

There are different approaches for these two stages. In [74], the distance matrix is split into blocks of rows and each matrix row is sorted using radix sort method. In [75], the previous distance matrix calculation scheme is used, but insertion sort method is applied instead of radix sort. [76] uses the same approach as [74] and [75] to compute the distance matrix and, for the selection phase, they calculate a local k-NN for each block of threads and obtain a global k-NN by merging. In [77], for the matrix computation uses the [74] and [75] scheme and modifies the selection phase with a quicksort-based selection. Each block performs a selection operation with a large number of threads per block. In [78], the truncated sort algorithm was introduced in the selection phase.

In [79], the *GPU-FS-kNN* algorithm was presented. It divides the computation of the distance matrix into squared chunks. Each chunk is computed using a different kernel call, reusing the allocated GPU-memory. A selection phase is performed after each chunk is processed.

In [80], an incremental neighborhood computation that eliminates the dependencies between dataset size and memory is presented. The iterative process copies several reference point subsets into the GPU. Then, the algorithm runs the local neighborhood search to find the k nearest neighbors from the query points to the reference point subsets. Merging candidate result sets with new reference point subsets is used to reach the final solution.

In [81], a GPU heap-based algorithm (called Batch Heap-Reduction) is presented. Since it requires large shared memory, it is not able to solve k-NN queries for high k values. In [82], new approaches to solving k-NN queries in GPU using exhaustive algorithms based on the selection sort, quicksort and state-of-the-art heaps-based algorithms are presented.

[83] proposes a new algorithm that is also suitable for several GPU devices. The distance matrix is split into blocks of rows. Each thread computes the distances for a row. Parallel threads push new candidates to a max-heap using atomic operations. [84] presents a multi-GPU implementation of a k-NN algorithm.

In [85], a new BF 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. This brute force k-NN approach has been used in [86] to accelerate calibration process of a k-nearest neighbors classifier using GPU.

Some of these algorithms (like the ones of [75] and their improved implementations [87]) consume a lot of device memory, since a Cartesian product matrix, containing the distances of reference points to the query points, is stored. In this chapter, we present alternative algorithms that focus on maximizing the total reference points stored in the device memory, which could accelerate execution by avoiding the creation of extra (unnecessary) data chunks and could scale-up to larger reference datasets.

4.2 k-NN In-memory GPU-based Algorithms

In this section, we present the two new in-memory algorithms that we developed and implemented using the CUDA Thrust library.

4.2.1 Thrust Brute-Force

The first method is based on a "brute force" algorithm (Fig.4.1), using the Thrust library (denoted by T-BF). Brute force algorithms are highly efficient when executed in parallel. The algorithm has three input parameters, the k value, a dataset R consisting of m reference points $R = \{r_1, r_2, r_3..r_m\}$ in a 3d space and a dataset Q of n query points $Q = \{q_1, q_2, q_3..q_n\}$ also in a three-dimensional space. For every query point $q \in Q$, the following steps are executed (Alg. 3):

1. Calculate all the Euclidean distances between the query point q and the reference points r E R. Store the calculated distances in a dataset D consisting of m distances $D = \{d_1, d_2, d_3..d_m\}$
2. Sort the distances D dataset
3. Create the k-NN dataset K of the k nearest neighbor points $K = \{k_1, k_2, k_3..k_k\}$. These points contain the sorted distances as well as the R dataset indices.

Algorithm 3 Brute-force Host algorithm

```
Input: NN cardinality=K, Reference filename=RF, Query filename=QF
Output: Host k-NN Buffer=HostKNNBufferVector
 1: HostQueryVector ← readFile(QF);
 2: queryPoints ← HostQueryVector.size();
 3: HostReferenceVector ← readFile(RF);
 4: referencePoints ← HostReferenceVector.size();
 5: DeviceReferenceVector ← HostReferenceVector;
 6: DeviceDistanceListVector(referencePoints);
 7: DeviceKnnListVector(K);     // k-size device vector
 8: for q in HostQueryVector do
 9:     thrust::transform(DeviceReferenceVector.begin(),DeviceReferenceVector.end(),DeviceDistanceListVector.begin(), CalcDist(q));
10:     thrust::sort( DeviceDistanceListVector.begin(),DeviceDistanceListVector.end(), DistanceCompare());
11:     thrust::copy(DeviceDistanceListVector.begin(), DeviceDistanceListVector.begin() + k, DeviceKnnBufferVector.begin());
12: HostKNNBufferVector ← DeviceKnnBufferVector;
```

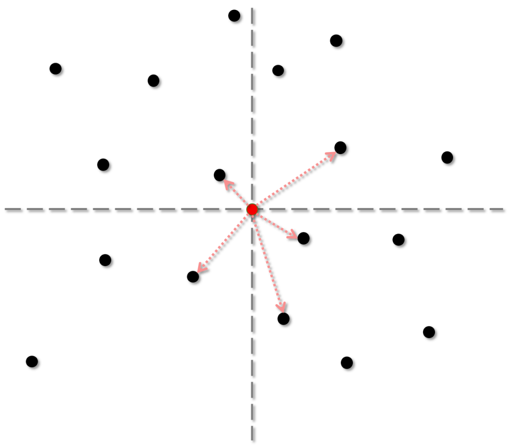


Figure 4.1: Query point (red), $k = 5$ nearest neighbors (black)

After n repetitions, all the k-NN points will be calculated. Although the BF is perfectly suitable for a GPU implementation, we noticed that the sorting step is extremely GPU computationally bound. The CUDA profiler revealed the 90% (or more in large datasets) of the GPU computation is dedicated to sorting.

4.2.2 Thrust Distance Refinement

The aforementioned method T-BF is memory efficient and well-performing but when the reference dataset is extremely big, the distance sorting step deteriorates the overall performance. This observation led us to our next implementation, which radically refines the nearest reference points.

The main concept of distance refinement method (denoted by T-DS) is that we can calculate the number of reference points by searching in concentric ranges. We count the reference points of each concentric range until the total points counted exceed the needed k points.

Our first approach was to search in concentric rings of equal width. The width of the ring l is constant and is calculated as k times double the maximum width of reference points area $tmax$, divided by the number of reference points tp.

$$l = k * 2t * tmax/tp$$

Experimentation revealed that this approach was only efficient in dense and uniformly distributed reference points without gaps. When we used synthetic or real data, the results were the same or only a bit better than the T-BF method. The problem was that the search area did not scale quickly enough.

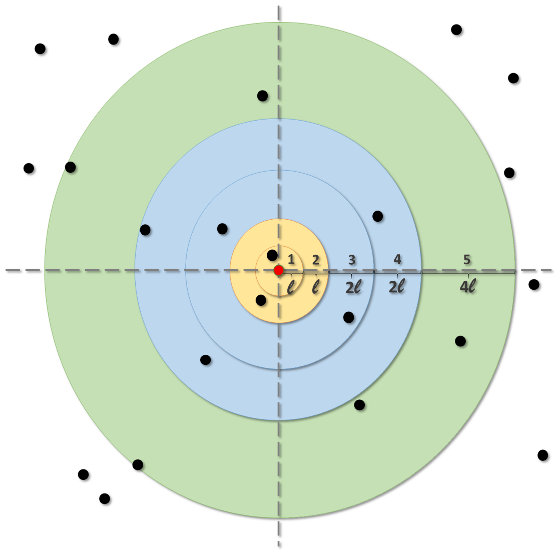


Figure 4.2: Distance Refinement, query point in red, reference points in black, $k = 10$

On our second approach, we used a hybrid area incrementation step. Every two search rings, we doubled the search ring width (Fig.4.2). This way the search increment was semi-exponential. The algorithm can scale quickly and produce excellent results for all kinds of datasets.

By refining the initial dataset, we create a very small intermediate dataset that contains at least k reference points. This dataset will be used in the costly sorting part of the algorithm and will speed up the execution time (as we will experimentally show).

The algorithm (Alg. 4) has three input parameters, the k value, a dataset R consisting of m reference points $R = \{r_1, r_2, r_3..r_m\}$ in a three-dimensional space and a dataset Q of n query points $Q = \{q_1, q_2, q_3..q_n\}$ also in a three-dimensional space. For every query point $q \in Q$, the following steps are executed:

1. Calculate the starting ring width l
2. Calculate all the Euclidean distances between the query point q and the reference points r E R. Store the calculated distances in a dataset D consisting of m distances $D = \{d_1, d_2, d_3..d_m\}$
3. While the count of reference points c is less than equal of k repeat
    - If $repetion\%2 = 0$, count the distance points of the area ring between the circles with radius $repetion * l$ and $(repetition + 1) * l$

    - If $repetion\%2 = 1$, count the distance points of the area ring between the circles with radius $repetion * l$ and $(repetition + 1) * l$. Double the ring width $l = 2 * l$

4. Refine the distance points of less than or equal to distance $(repetition + 1) * wr$ and copy them to dataset DR consisting of c distances $DR = \{dr_1, dr_2, dr_3..dr_c\}$

5. Sort the distances DR dataset

6. Create the k-NN dataset K of the k nearest neighbor points $K = \{k_1, k_2, k_3..k_k\}$. These points contain the sorted distances as well as the R dataset indices.

Algorithm 4 Distance Refinement Host algorithm

```
Input: NN cardinality=K, Reference filename=RF, Query filename=QF
Output: Host $k$-NN Buffer=HostKNNBufferVector
1: HostQueryVector ← readFile(QF);
2: queryPoints ← HostQueryVector.size();
3: HostReferenceVector ← readFile(RF);
4: referencePoints ← HostReferenceVector.size();
5: DeviceReferenceVector ← HostReferenceVector;
6: DeviceDistanceListVector(referencePoints);
7: DeviceKnnListVector(K);      // k-size device vector
8: for q in HostQueryVector do
9:     thrust::transform(DeviceReferenceVector.begin(),DeviceReferenceVector.end(),DeviceDistanceListVector.begin(), CalcDist(q));
10:    countpoints←0;
11:    step←0;
12:    sensestep ←0;
13:    while countpoints < k do
14:        countpoints← countpoints + thrust::count_if(DeviceDistanceListVector.begin(), DeviceDistanceListVector.end(), countsteppoints(step,sensestep));      // Caclulate next concentric ring point cardinality
15:        step++;
16:        if (step % 2 == 0) then
17:            step ← step / 2;
18:            sensestep ← sensestep * 2;
19:    ReducedDistanceList(countpoints);      // “countpoints”-size device vector
20:    thrust::copy_if(DeviceDistanceListVector.begin(), DeviceDistanceListVector.end(), ReducedDistanceList.begin(), countsteppoints(0,sensestep*step));
21:    sensestep ← (sensestep + step * sensestep ) / 2;
22:    if countpoints >= 2 * k then
23:        sensestep ← sensestep / 2;
24:    thrust::sort(ReducedDistanceList.begin(), ReducedDistanceList.end(), DistanceCompare());
25:    thrust::transform(ReducedDistanceList.begin(), ReducedDistanceList.begin() + k, DeviceKnnBufferVector.begin(), calc_sqrt());
26: HostKNNBufferVector ← DeviceKnnBufferVector;
```

After n repetitions, all the k-NN points will be calculated. For example in Fig. 4.2, we calculated a distance refinement of radius $4l$ resulting to a total of 12 distance points.

4.2.3 Symmetric Progression Partitioning Algorithm

A common practice to address the need of extremely big data analysis is data partitioning, which has driven us to design and implement the novel method "Symmetric Progression Partitioning", denoted as SPP. 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. Our algorithm consists of 4 main steps (Fig. 4.3):

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 SPP
4. The k-NN list is populated with the results of all the query points

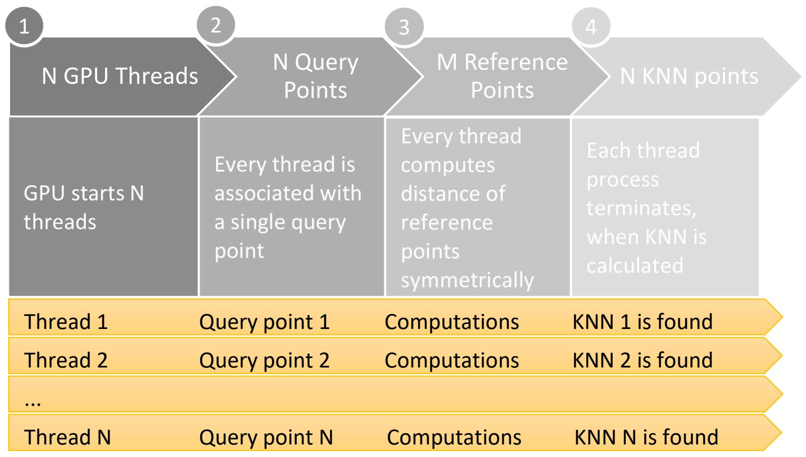


Figure 4.3: SPP Algorithm steps. Every thread computes one query point.

At this point, we should outline that SPP approach is completely different than the approach we used in our previous methods T-BF and T-DS [88]. In our previous methods we used all the available GPU threads to calculate k-NN for one query point at a time. In our current implementation every thread is in charge of delivering the k-NN of a single query point. We have experimentally verified that our previous methods, and especially T-DS, when using small groups of query points performed better than previous methods by other researchers. In our new implementation, when using larger query point groups, SPP should perform much better. In our experiments we will show that SPP outperforms all the existing algorithms.

In order to properly partition the reference dataset, the dataset points should be sorted. The partitioning technique we are using, partitions the dataset in equally sized bins throughout the X-axis. SPP searches for k-NN starting from the partition that it's bounding box contains the query point (partition number 5, in our example, Fig.4.4). If k-NN are not found the thread searches for k-NN in the next closest partition (partition 6). Similarly, the process continues until all reference points are processed. In Fig. 4.4 we search for 20 nearest neighbours. We processed 7 out of 10 partitions and found the k-NN. Partitions 1, 9 and 10 were excluded because the 20 nearest neighbors were already found.

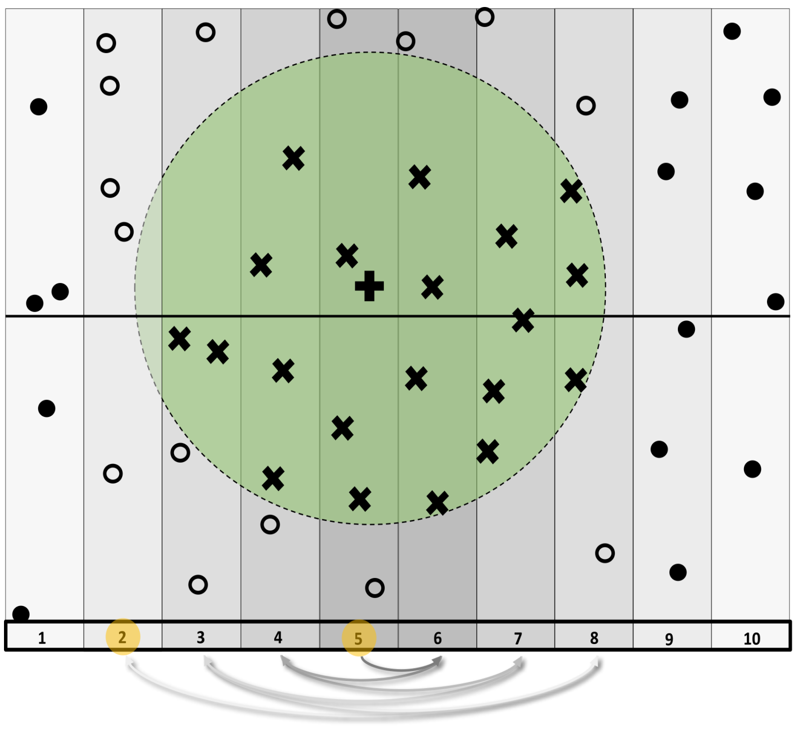


Figure 4.4: SPP in-memory Partition example, query point represented by + symbol, reference points represented by $\times$ symbols (k-NN points), analyzed points represented by empty circles, non-analyzed points represented by filled circles, $k = 20$ [8].

A new optimization that we introduce in SPP is the way the k-NN distance buffer is populated. This buffer is presented in Section 4.2.5. The k-NN Distance List Buffer (KNN-DLB) is an array where all calculated distances are stored. KNN-DLB array size is K per thread, resulting to a minimum possible device memory utilization. Another advantage of

KNN-DBL is that we do not need to use a sorting algorithm like radix sort, insertion sort or quick sort. The resulting buffer contains the right k-NNs, but not in an ascending order.

The algorithm is divided in two parts. The first one is executed in the host (Alg. 5) and the second one is executed in the device (Alg. 6), as a device kernel. The algorithm has two input parameters, a dataset R consisting of m reference points $R = \{r_1, r_2, r_3 \cdots r_m\}$ in a three-dimensional space sorted by the X-axis and a dataset Q of n query points $Q = \{q_1, q_2, q_3 \cdots q_n\}$ also in a three-dimensional space. The following steps are executed:

1. Partition the whole dataset in j equally sized bins $B = \{b_1, b_2, b_3 \cdots b_j\}$
2. For every $q \in Q$, assign a GPU thread
3. Find the initial partition b_q that contains the query point q
4. Repeat until all k-NN are found
   - Calculate all the Euclidean distances of the query point q and the initial partition b_q or the next closest partition reference points $b_r \in B$. For example, the partition calculation steps in Fig. 4.4 are $5 \rightarrow 6 \rightarrow 4 \rightarrow 7 \rightarrow 3 \rightarrow 8 \rightarrow 2$
   - Add or replace (Fig. 4.1) the calculated distances in the k-NN Distance List Buffer D consisting of k distances, $D = \{d_1, d_2, d_3 \cdots d_K\}$

4.2.4 Heap Symmetric Progression Partitioning

Heap Symmetric Progression Partitioning Algorithm is a new algorithm that extends SPP by adding max heap (presented in Section 4.2.6) for k-NN as the list buffer. SPP is using k-NN Distance List Buffer (KNN-DLB). KNN-DLB is adequate for smaller ks but when the k number increases SPP performance deteriorates, primarily due to KNN-DLB O(n) complexity. On the other hand max heap is O(log(n)) complexity and we will prove experimentally that HSPP outperforms SPP, especially for larger k values.

4.2.5 $k-$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) [89, 90]. KNN-DLB is an array where all calculated distances are stored (Table 4.1). KNN-DLB array size is k per thread, resulting to

Algorithm 5 SPP Host algorithm

```
Input: NN cardinality=K, Reference filename=RF, Query filename=QF, Partition size=S
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: HostReferenceVector ← readFile(RF);
 8: referencePoints ← HostReferenceVector.size();
 9: DeviceReferenceVector ← HostReferenceVector;
10: cudaSort(DeviceReferencePartition);
11: PartitionsCount ← referencePoints/S
12: createEmptyHostVector(HostReferencePartitionIndex,1+CB);
13: HostReferencePartitionIndex.add(0,HostReferenceVector[0].x);
14: for i=0 to PartitionsCount-1 do
15:     HostReferencePartitionIndex.add(HostReferenceVector[i*S].x, HostReferenceVector[(i+1)*S].x,i*S,(i+1)*S);
16: DeviceReferencePartitionIndex ← HostReferencePartitionIndex;
17: runKNN<<<(queryPoints-1)/256 +1, 256>>>(DeviceReferencePartition,DeviceQueryVector,DeviceKNNBufferVector,K,S);
    // 256 cores
18: HostKNNBufferVector ← DeviceKNNBufferVector;
```

Algorithm 6 SPP Device Kernel algorithm (runKNN)

```
Input: NN cardinality=K, 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: for currentPartition ← 0 to DeviceReferencePartitionIndex.size()-1 do
 3:     if DeviceReferencePartitionIndex[currentPartition].right-X-Limit<Q[qIdx].x then break;
 4: if currentPartition < DeviceReferencePartitionIndex.size()-1 then currentPartition- -;
 5: while maxdistance>Q[qIdx].x-DeviceReferencePartitionIndex[currentPartition].left-X-Limit or
          maxdistance>DeviceReferencePartitionIndex[currentPartition].right-X-Limit-Q[qIdx].x do
 6:     idx1 ← DeviceReferencePartitionIndex(currentPartition).left-X-Limit-Index;
 7:     idx2 ← DeviceReferencePartitionIndex(currentPartition).right-X-Limit-Index;
 8:     for i ← idx1 to idx2-1 do
 9:         dist ← $\sqrt[2]{(R[i].x - Q[qIdx].x)^2 + (R[i].y - Q[qIdx].y)^2 + (R[i].z - Q[qIdx].z)^2}$
10:         insertIntoBuffer(DKB,i,qIdx,dist);
11:     maxdistance ← calcMaxDistance(DKB);
12:     currentPartition ← FindNextClosestPartition;
```

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 [8].

	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 4.1: k-NN Distance List Buffer, k=10

4.2.6 $k-$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) [89, 90]. 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 is the constant DBL_MAX). (Fig.4.5). 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.

4.3 Experimental Study

We run a large set of experiments to compare the repetitive application of the existing algorithms and the newly implemented ones for processing batch k-NN queries. We conducted

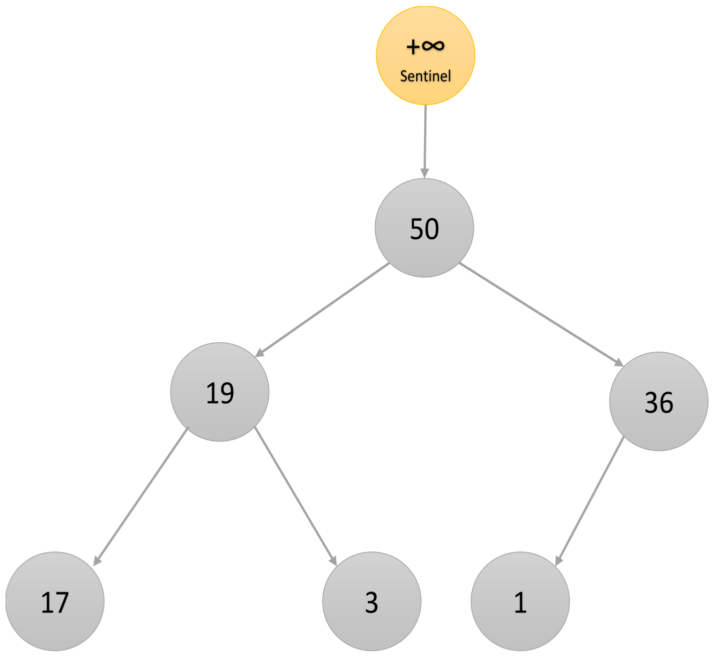


Figure 4.5: Max-Heap structure example.

three different sets of in-memory experiments.

The first set evaluates the performance of T-BF and T-DS. T-DS is designed to query small query datasets, suitable for streaming applications that require fast response times. In this set, we run experiments to compare the performance of batch K-NN queries, regarding execution time as well as memory utilization. We tested a total of five algorithms. Three of them are existing ones and the other two are the new algorithms we implemented. The list of algorithms is as follows:

1. BF-Global, Brute Force algorithm using the Global memory [75]
2. BF-Texture, using the GPU texture memory [87]
3. BF-Cublas, using CUBLAS (BLAS highly optimized linear algebra library) [75]
4. T-BF, Brute Force algorithm using Thrust library [88]
5. T-DS, Distance Refinement algorithm using Thrust library [88]

The second set evaluates the performance of SPP. SPP is targeting larger query datasets. We run experiments to compare the performance of multiple k-NN queries, regarding execution time as well as memory utilization. We tested a total of seven algorithms. Four of them are existing ones, T-BF and T-DS are our previous k-NN algorithms and SPP is the new proposed algorithm. The list of algorithms is as follows:

1. BF-Global, Brute Force algorithm using the Global memory [75]
2. BF-Texture, using the GPU texture memory [87]
3. BF-Cublas, BLAS highly optimized linear algebra library [75]
4. AIDW, Fast k-NN search used for Adaptive Inverse Distance Weighting (AIDW) interpolation algorithm [91]
5. T-BF, Brute Force algorithm using Thrust library [88]
6. T-DS, Distance Refinement algorithm using Thrust library [88]
7. SPP, Symmetric Progression Partitioning algorithm [8]

The third set evaluates the performance of HSPP. HSPP is designed to perform better for higher k values.We are testing the following three algorithms:

1. T-DS, Distance Refinement algorithm using Thrust library [88],
2. SPP, Symmetric Progression Partitioning algorithm [8]
3. HSPP, Heap Symmetric Progression Partitioning algorithm. [89]

4.3.1 In-memory, 1st set of experiments

All experiments query at least 100K reference points. We did not include less than 100K reference points because we target the maximum in-memory utilization and reference point less than 100K do not fit in this context. The maximum reference points limit is 200M, which only our algorithm T-DS achieved.

We have created random and synthetic clustered datasets of 100.000, 250.000, 500.000 and 1.000.000 points. All the existing methods could only be executed with 100.000 reference points and only one of them scaled to 1.000.000 points. Furthermore, in order to check our method scaling we also created random datasets of 10.000.000, 100.000.000 and 200.000.000 reference points. We also used three big real datasets [1], which represent water resources of North America (Water Dataset) consisting of 5.836.360 line-segments and world parks or green areas (Parks Dataset) consisting of 11.503.925 polygons and world buildings (Buildings Dataset) consisting of 114.736.539 polygons. To create sets of points, we used the centers of the line-segment MBRs from Water and the centroids of polygons from Park and Build. For all datasets the 3-dimensional data space is normalized to have unit length (values [0, 1] in each axis).

We run a series of experiments searching for 20 Nearest Neighbors ($k = 20$), using the aforementioned datasets with groups of 1,10,100,250,500,750 random query points. Every experiment was run at least 10 times and the mean of the experiment results were calculated for every method.

All experiments were performed on a Dell Inspiron 7577 laptop, running Windows 10 64bit, equipped with a quad-core (8-thread) Intel I7 CPU, 16GB of main memory, a 256SSD disk used for the operating system, a 1TB 7.2K SATA-3 Seagate HDD storing our data and a NVIDIA Geforce 1060 (Mobile Max-Q) GPU with 6GB of memory.

4.3.1.1 Random reference points

In our first series of tests, we used random datasets for the reference points. We created the datasets in various volumes using normalized random points. The resulting datasets' density increased analogously to the reference points cardinality. The datasets created are near uniformly distributed.

In the first chart (Fig. 4.6) with one query point, we can see that our algorithms T-BF and T-DS are extremely faster than the other methods. For one query point T-BF finished in 0,53ms and T-DS in 0,59ms for the 100K reference points experiment. The best of the other methods was BF-Texture achieving 39,68ms. It worths mentioning that the 100K experiment was the only one that BF-Texture and BF-Cublas finished. These two methods (BF-Texture and BF-Cublas) could not scale at higher volumes of reference points, mainly due to execution exceptions regarding memory allocation problems. For the 1M reference points experiment the execution time difference is much greater, the BF-Global implementation finished in 369,18ms while T-BF finished in 3,61ms and T-DS in 0.92ms. The maximum speedup gain that T-DS achieved was 529 (times faster) than BF-Global, at 750K reference points (Table 4.2).

In the second chart with 10 query points, we can observe about the same results as with the one query experiment. At 100K reference points, the best of existing algorithms was again BF-Texture, finishing at 40ms, while T-BF finished at 4.29ms and T-DS at 5.11ms. For the 1M reference points experiment the execution time difference is again much greater, the BF-Global implementation finished in 377,91ms while T-BF finished in 36.11ms and T-DS in

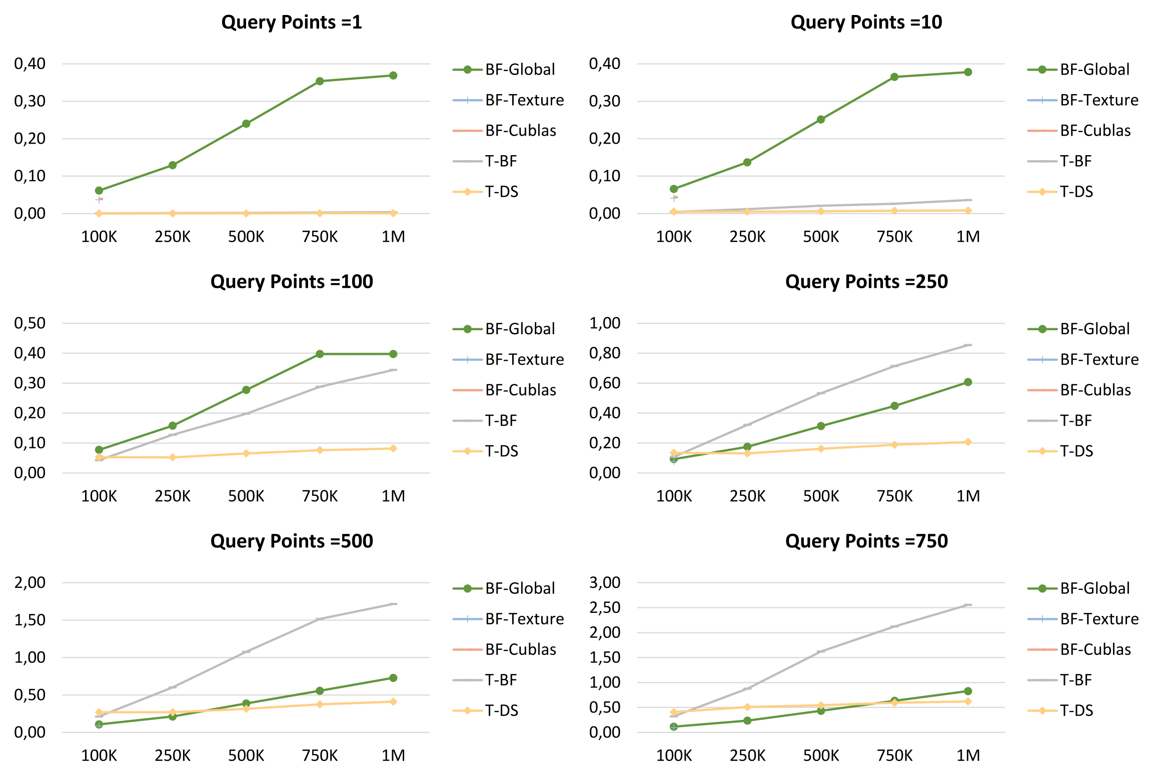


Figure 4.6: Random reference points, $k = 20$. X-axis: reference point cardinality, Y-axis: execution time measured in ms.

8.28ms. T-DS was 45 times faster than BF-Global.

When reaching the group of 100 query points, we can see that the execution time of T-BF is a slightly better than the BF-Global. The overhead of sorting large volumes of distance datasets begin to emerge. The T-DS on the other hand continued its excellent performance, finishing in 81.8ms at 1M reference points, while BF-Global finished in 397.42ms.

In the following experiments, using groups of 250,500 and 700 query points, T-BF performance was inferior compared to BF-Global. The T-DS algorithm kept on outperforming the other ones, especially on the large volumes of reference points.

In order to explore the limits of the algorithms, we created random datasets ranging from 10M to 200M reference points (Fig. 4.7). The existing algorithms could not scale higher than 1M reference points. The T-BF reached 100M reference points. The only algorithm that succeeded in 200M reference points is T-DS. T-BF using one query point, executed in 10M reference points in 28.96ms and T-DS in just 4.07ms. The T-DS finished in the 200M reference points experiment in 61.93ms. The 10 points query group, resulted analogously in 291.93ms for T-BF and 37.08ms for T-DS. All the other experiments resulted to close linear

		Random Reference Points				
Algorithm	Query Points	100K	250K	500K	750K	1M
T-BF	1	117,27	101,69	123,58	122,25	102,41
T-DS	1	104,33	251,95	405,79	529,63	401,72
T-BF	10	15,36	11,62	11,98	13,82	10,47
T-DS	10	12,88	26,07	41,74	48,79	45,64
T-BF	100	1,80	1,24	1,40	1,38	1,15
T-DS	100	1,46	3,03	4,26	5,21	4,86
T-BF	250	0,86	0,54	0,59	0,63	0,71
T-DS	250	0,69	1,33	1,94	2,39	2,93

Table 4.2: Speedup gain of new methods T-BF,T-DS versus BF-Global, using Random dataset

performance, in respect to query points groups. The T-DS implementation executed 10 times faster than the T-BF one, in extremely large volumes of reference datasets.

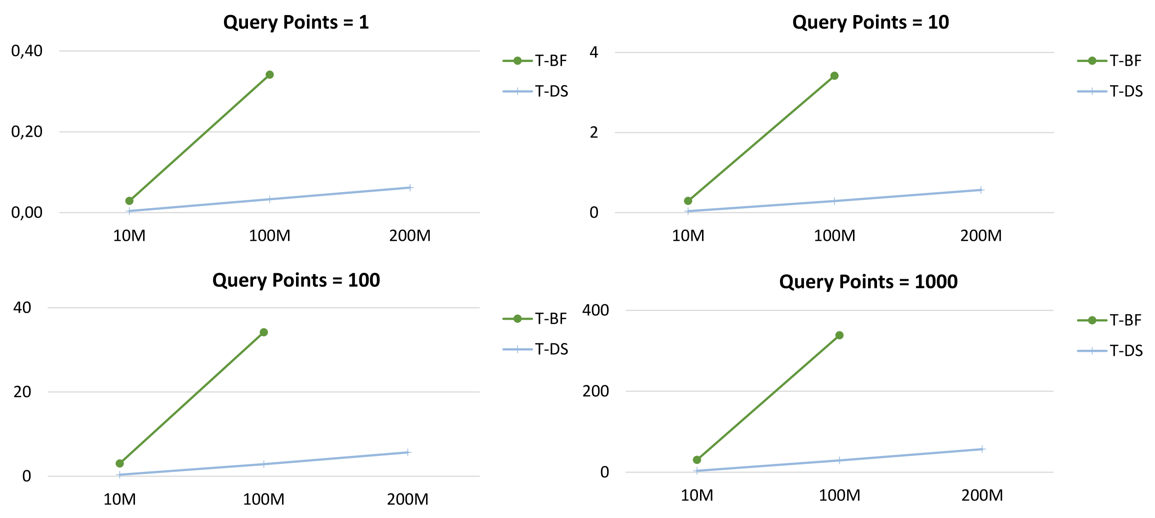


Figure 4.7: Maximum reference points, $k = 20$. X-axis: reference point cardinality, Y-axis: execution time measured in ms.

In terms of memory scaling the new algorithm T-DS can compute up to 200M reference points. The maximum reference points that other methods could achieve are 1M reference points, thus our methods can scale up to 200 times more than the other ones. It worths mentioning that T-DS is much faster than T-BF, because of the sorting overhead of T-BF.

4.3.1.2 Synthetic reference points

We have also used synthetic datasets following different data distributions, and each data set contains 100.000, 250.000, 500.000, 750.000 and 1.000.000 points (Fig. 4.8). We have created these datasets because they are not uniformly distributed, like the random ones and they resemble real-life data distributions. The synthetic reference points datasets are created according to Zipf's law. The Zipf distribution is defined as follows, and the value of z that we have used is 0.7.

$$f(i) = \frac{\frac{1}{i^z}}{\sum_{j=1}^{n} \frac{1}{j^z}}, i = 1, 2, \cdots, n$$

The random experiment results are confirmed in this experiment. In the one query point experiment, in ascending order, T-BF finished in 0.48ms, T-DS in 0.65ms, BF-Texture in 49,76ms, BF-Global in 62.04ms and BF-Cublas in 74.41ms, for the 100K reference dataset. The 1M experiment resulted to T-DS 2.18ms, T-BF 3.56ms and BF-Global 367.57ms. The maximum speedup gain that T-DS achieved was 270 (times faster) than BF-Global, at 750K reference points (Table 4.3).

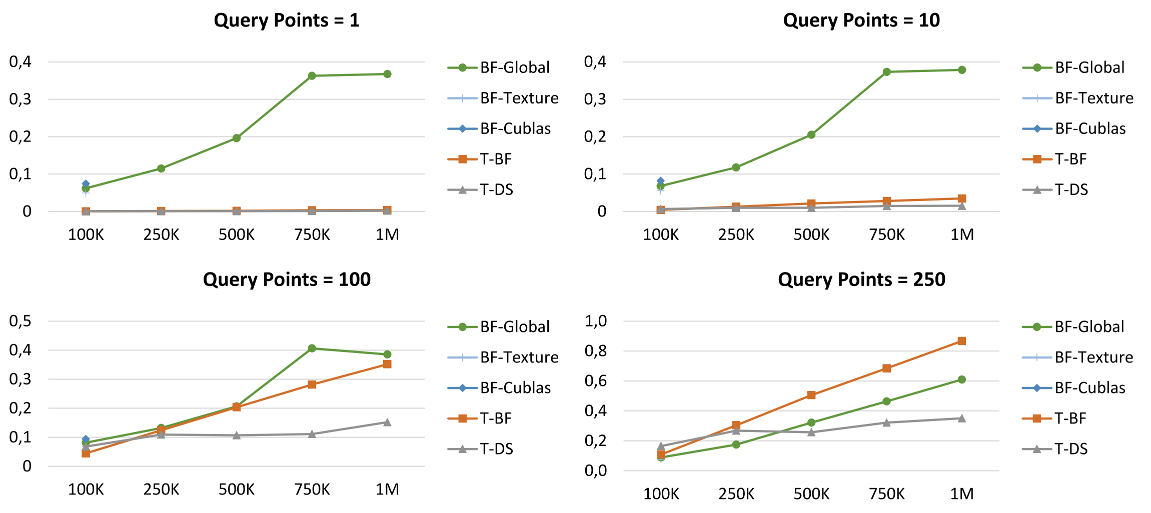


Figure 4.8: Synthetic reference points, $k = 20$. X-axis: reference point cardinality, Y-axis: execution time measured in ms.

The 10 query point experiment resulted to a similar outcome. The T-DS again was faster and outperformed BF-Global by 25 times. When the query points reached up to 100, we noticed that the performance of T-BF is similar and slightly better than BF-Global. On the other hand T-DS is still performing good, especially for $\geq$ 500K reference points. Finally

		Synthetic Reference Points				
Algorithm	Query Points	100K	250K	500K	750K	1M
T-BF	1	128,45	89,87	101,04	118,75	103,25
T-DS	1	94,57	134,70	196,90	270,53	168,76
T-BF	10	15,64	9,02	9,52	13,22	10,86
T-DS	10	10,34	11,56	20,48	25,04	24,58
T-BF	100	1,82	1,06	1,02	1,44	1,10
T-DS	100	1,20	1,21	1,94	3,65	2,54
T-BF	250	0,82	0,58	0,64	0,68	0,70
T-DS	250	0,54	0,66	1,25	1,44	1,74

Table 4.3: Speedup gain of new methods T-BF,T-DS versus BF-Global, using Synthetic dataset

in the 250 query points experiment, BF-Global surpasses T-BF, but is still slower than T-DS algorithm.

For one more time, T-DS is much faster than T-BF, because of the sorting advantage reduction, as documented in the previous section.

4.3.1.3 Real reference points Comparison

We conducted 6 experiments using three different real datasets using groups of query points of 10 and 100 points (Fig. 4.9). In order to compare all the algorithms we created two reference datasets of 100K and 1M points per every real dataset, by reducing them (the real datasets) uniformly.

The only experiment that all the algorithms completed, is the one with 100K reference points. The BF-Texture and BF-Cublas algorithm failed in all subsequent experiments. The fastest algorithm in the 100K case was T-BF, finishing in 4.53ms (Water), 4.38ms (Parks) and 4.36ms (Buildings), while querying 10 points and 43.78ms (Water), 43.99ms (Parks) and 41.1ms (Buildings) while querying 100 points. In the case of 1M reference points the T-DS was slightly faster than T-BF. When we queried the hole datasets the T-DS was about 2 times faster than T-BF.

The maximum speedup gain that T-DS achieved was 391 (times faster) than BF-Global, at 1M water reference points, using one query point (Table 4.4). Furthermore, in the real

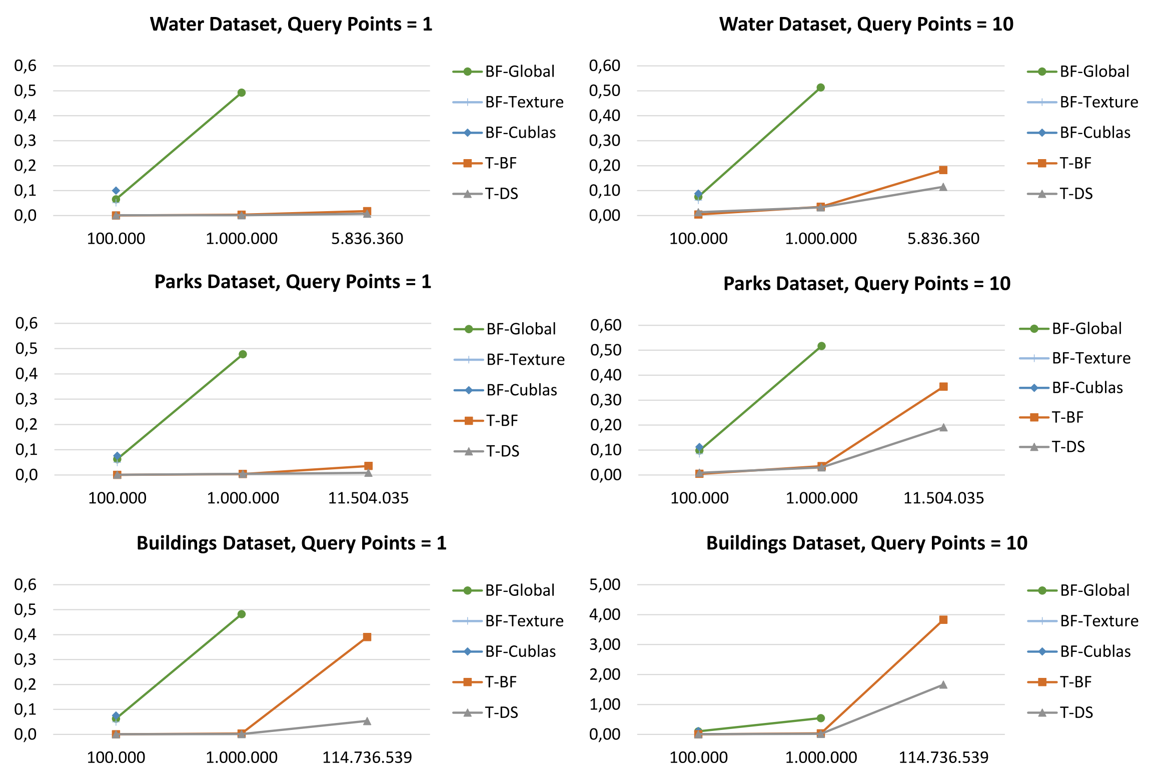


Figure 4.9: Real reference points, $k = 20$. X-axis: reference point cardinality, Y-axis: execution time measured in ms.

dataset example, we certify once more that T-DS is much faster than T-BF, because of the sorting advantage.

		Water Ref. Points		Parks Ref. Points		Buildings Ref. Points	
Algorithm	Query Points	100K	1M	100K	1M	100K	1M
T-BF	1	120,98	135,49	128,43	131,26	119,25	136,10
T-DS	1	38,31	391,19	74,74	102,16	78,85	352,03
T-BF	10	16,71	14,55	22,33	14,54	23,81	15,64
T-DS	10	5,55	15,73	11,07	16,95	10,97	32,28
T-BF	100	2,72	1,84	2,52	1,71	4,00	1,98
T-DS	100	1,14	1,81	1,53	2,31	2,19	3,22
T-BF	250	1,51	1,02	1,37	0,76	1,50	0,89
T-DS	250	0,61	1,08	0,79	1,04	0,80	1,47

Table 4.4: Speedup gain of new methods T-BF,T-DS versus BF-Global, using Real datasets

4.3.2 In-memory, 2nd set of experiments

We run a large set of experiments to compare the repetitive application of the existing algorithms and the proposed one (SPP) for processing batch k-NN queries. All experiments query at least 100K reference points. We did not include less than 100K reference points because we target the maximum in-memory utilization and reference point less than 100K, do not fit in this context. All the experiments, fit the reference and input datasets into device memory.

We have created random and synthetic datasets of 100K, 250K, 500K and 1M points. All the existing methods could only be executed with 100K reference points and only two of them scaled to 1M points. Furthermore, in order to check our method scaling we also created random datasets of 10M, 100M, 200M and 300M reference points. We also used three big real datasets [1], which represent water resources of North America (Water Dataset) consisting of 5,836,360 line-segments and world parks or green areas (Parks Dataset) consisting of 11,503,925 polygons and world buildings (Buildings Dataset) consisting of 114,736,539 polygons. To create sets of points, we used the centers of the line-segment MBRs from Water and the centroids of polygons from Park and Buildings. For all datasets the 3-dimensional data space is normalized to have unit length (values [0, 1] in each axis).

We run a series of experiments searching for 20 Nearest Neighbors (k= 20), using the aforementioned datasets with groups of 250, 500, 750 and 1,000 random query points. By using these larger groups of query points we will demonstrate the effectiveness of SPP. Every experiment was run at least 10 times and the mean of the experiment results were calculated for every method.

All experiments were performed on a Dell Inspiron 7577 laptop, running Windows 10 64bit, equipped with a quad-core (8-thread) Intel I7 CPU, 16GB of main memory, a 256GB SSD disk used for the operating system, a 1TB 7.2K SATA-3 Seagate HDD storing our data and a NVIDIA Geforce 1060 (Mobile Max-Q) GPU with 6GB of memory.

4.3.2.1 Random reference points Comparison

In our first series of tests, we used random datasets for the reference points. We created the datasets in various volumes using normalized random points. The resulting datasets' density increased analogously to the reference points cardinality. It is worth mentioning that due to

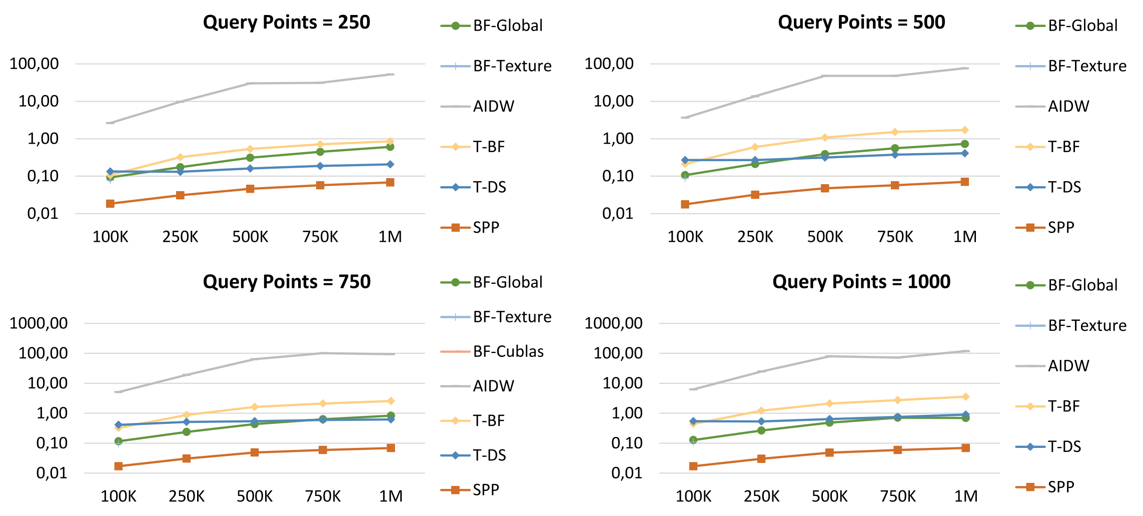


Figure 4.10: Random reference points, $k = 20$. X-axis denotes reference point cardinality, Y-axis (in logarithmic scale) denotes execution time measured in sec.

the nature of the random number generator, the datasets created, are in a way, near uniformly distributed.

In the first chart (Fig. 4.10), we can see that our algorithm SPP is faster than the other methods. For 250 query points SPP finished in 18.51ms for the 100K reference points experiment. The best of the other methods was BF-Texture achieving 77.07ms and the worst was AIDW with 2.63 sec. It worth mentioning that the 100K experiment was the only one that BF-Texture and BF-Cublas finished. These two methods (BF-Texture and BF-Cublas) could not scale at higher volumes of reference points, mainly due to execution exceptions regarding memory allocation problems. For the 1M reference points experiment the execution time difference is even larger, the BF-Global implementation finished in 606.88ms, T-BF finished in 854.04ms, T-DS in 206.98ms and SPP in just 68.31ms. The AIDW method was by far the slowest finishing in 52.16sec. The maximum speedup gain that SPP achieved was 8.88 (times faster) than BF-Global, at 1M reference points (Table 4.5). In this gain comparison we notice that some numbers are less than zero. This means that the performance was worse than the basic comparison method BF-Global. For example, at 100K, for 250 query points T-BF is slower by 0.86 ($1/0.86 = 1.1627$ times slower).

In the second chart with 500 query points, we can observe about the same results as with the 250 query points experiment. With 100K reference points, the best of existing algorithms was again SPP, finishing at 17.76ms, while T-BF finished at 212.54ms, T-DS at 270.61ms, BF-Global at 107.07ms, and BF-Texture at 89.27ms. AIDW was the slowest at 3.64sec. For

Algorithm	Query	Random Reference Points					Query	Random Reference Points				
	Points	100K	250K	500K	750K	1M	Points	100K	250K	500K	750K	1M
AIDW	250	0.04	0.02	0.01	0.01	0.01	500	0.03	0.02	0.01	0.01	0.01
T-BF	250	0.86	0.54	0.59	0.63	0.71	500	0.5	0.35	0.36	0.37	0.42
T-DS	250	0.69	1.33	1.94	2.39	2.93	500	0.4	0.79	1.22	1.48	1.78
SPP	250	**5.02**	**5.65**	**6.79**	**7.84**	**8.88**	500	**6.03**	**6.6**	**8.11**	**9.66**	**10.3**
AIDW	750	0.02	0.01	0.01	0.01	0.01	1000	0.02	0.01	0.01	0.01	0.01
T-BF	750	0.36	0.27	0.27	0.3	0.32	1000	0.29	0.22	0.23	0.26	0.19
T-DS	750	0.28	0.46	0.8	1.07	1.34	1000	0.24	0.5	0.76	0.94	0.77
SPP	750	**6.8**	**7.75**	**8.95**	**10.69**	**12.04**	1000	**7.46**	**8.71**	**10.07**	**12.00**	**10.02**

Table 4.5: Speedup gain of AIDW, T-BF, T-DS, SPP versus BF-Global, Random dataset.

1M reference points, the execution time difference is once more larger, the SPP algorithm finished in 70.8ms and it was 10.3 times faster than BF-Global (Table 4.5).

When reaching the group of 1,000 query points, we can see that the execution time of SPP is again better than the other methods. The overhead of sorting large volumes of distance datasets begins to emerge resulting to poor performance for T-BF. The SPP on the other hand continues its excellent performance, finishing in 69.42ms 10.2 times faster than BF-Global.

In the following experiments, using groups of 250, 500, 750 and 1,000 query points, T-BF, T-DS and AIDW performance was inferior compared to BF-Global. The SPP algorithm kept on outperforming all of them, especially on the large volumes of reference points.

SPP is faster because of two main reasons. The first one is because it only processes part of the reference dataset to find k-NN, in contrast to the other algorithms, which always process the whole dataset. The second one is that the whole k-NN algorithm is executed in the GPU device and there is no need to hand over the process control and transfer data between the GPU and the host. In contrast, T-BF and T-DS always hand over the process control and transfer data between the GPU device and the host for every single query point. We observe that in the whole range of reference points, SPP outperforms all other methods.

In all cases, the AIDW algorithm was several times of magnitude slower than the other methods. In the 1M with 250 query points experiment, AIDW was 763 times slower than SPP and 61 times slower than T-BF. The only way to visualize all methods in the same charts was to display the X-axis in a logarithmic scale. In the forthcoming experiments we decided not to include AIDW.

In order to explore the memory capacity limits of the algorithms, we created random datasets ranging from 10M, 100M, 200M and 300M reference points (Fig. 4.11). The existing algorithms of other researchers could not scale higher than 1M reference points. T-BF implementation reached 100M reference points and T-DS reached 200M. The only algorithm that succeeded in 300M reference points is SPP. T-BF using 250 query points, executed 10M reference points in 7.55sec, T-DS in 927ms and SPP in just 254ms. The T-DS finished in the 200M reference points experiment in 14.31sec, while SPP in 1.63sec, 8.7 times faster. The 500, 750 and 1,000 points query groups, resulted analogously in favor of SPP.

Therefore, it is obvious that SPP is able to handle much larger reference datasets, because it uses a buffer (KNN-DLB) which holds only distances and IDs to the closest k points, in contrast to other algorithms which store distances and IDs for the whole reference dataset.

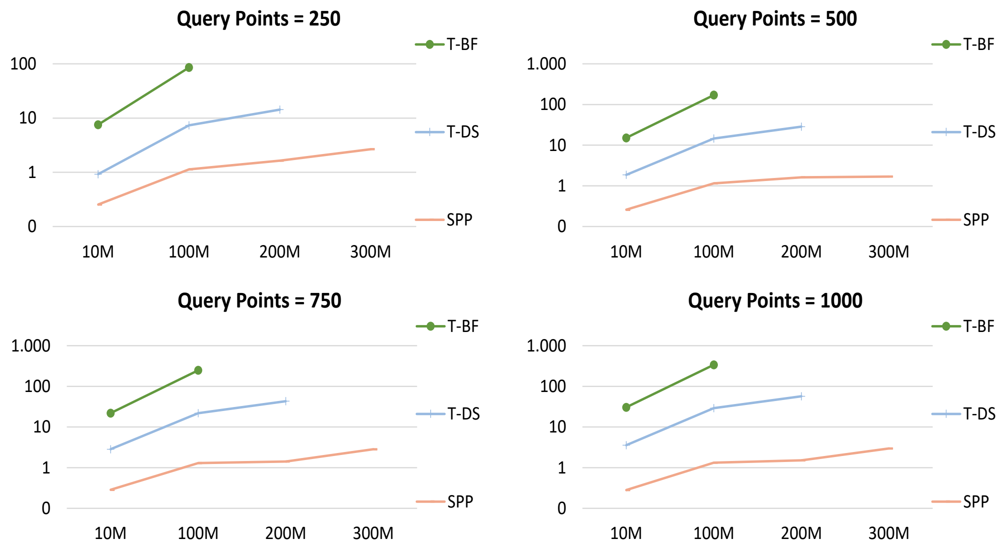


Figure 4.11: Maximum reference points, $k = 20$. X-axis denotes reference point cardinality, Y-axis denotes execution time measured in sec

In terms of memory scaling, SPP can compute up to 300M reference points, taking advantage of our KNN-DLB optimization. T-DS and T-BF could scale up to 200M and 100M respectively. Thus our new method can scale up to 300 times more than other existing ones and 100M reference points more than T-DS.

4.3.2.2 Synthetic reference points Comparison

We have also used synthetic datasets following different data distributions, and each data set contains 100K, 250K, 500K, 750K and 1M points (Fig. 4.12). We have created these datasets because they are not uniformly distributed, like the random ones and they resemble real-life data distributions. The synthetic reference points datasets are created according to Zipf's law. The Zipf distribution is defined as follows, and the value of *z* that we used is 0.7.

$$f(i) = \frac{\frac{1}{i^z}}{\sum_{j=1}^{n} \frac{1}{j^z}}, i = 1, 2, \cdots, n$$

The random experiment results are confirmed in this experiment. In the 250 query points experiment, in ascending order, SPP finished in 55.55ms, BF-Texture in 74.65ms, BF-Global in 89.20ms, T-BF in 109ms and T-DS in 165ms. BF-Cublas did not finish any of the synthetic tests. The 1M experiment resulted to the following: T-DS 350ms, SPP 559ms, BF-Global 609ms and T-BF 867ms. T-DS is still performing very well in small volumes of query point. In larger volumes of query points, T-DS's performance decreases.

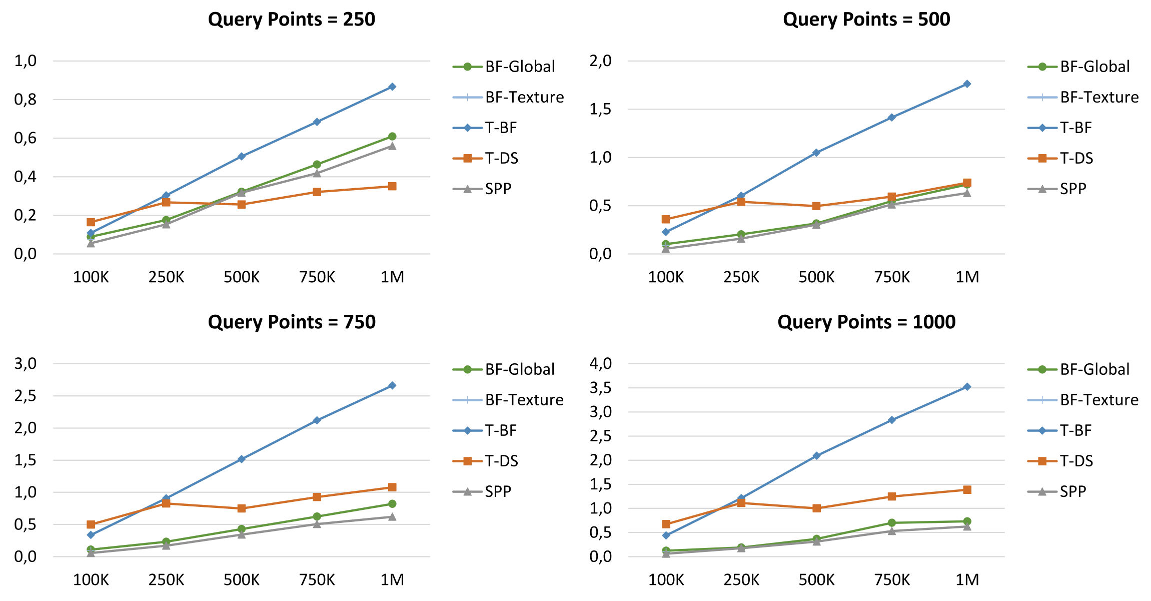


Figure 4.12: Synthetic reference points, $k = 20$. X-axis denotes reference point cardinality, Y-axis denotes execution time measured in sec.

The maximum speedup gain that T-DS achieved was 1.74 (times faster) than BF-Global, at 1M reference points (Table 4.6). For SPP the maximum gain was 1.61 at 100K. T-DS and SPP perform equally at 250 query points and they are faster than the other methods.

	Query	Synthetic Reference Points					Query	Synthetic Reference Points				
Algorithm	Points	100K	250K	500K	750K	1M	Points	100K	250K	500K	750K	1M
T-BF	250	0.82	0.58	0.64	0.68	0.7	500	0.45	0.34	0.3	0.39	0.41
T-DS	250	0.54	0.66	**1.25**	**1.44**	**1.74**	500	0.28	0.38	0.64	0.92	0.98
SPP	250	**1.61**	**1.14**	1.01	1.11	1.09	500	**1.86**	**1.28**	**1.04**	**1.07**	**1.14**
T-BF	750	0.32	0.25	0.28	0.29	0.31	1000	0.28	0.16	0.18	0.25	0.21
T-DS	750	0.22	0.28	0.57	0.67	0.76	1000	0.18	0.17	0.37	0.56	0.53
SPP	750	**1.92**	**1.34**	**1.25**	**1.23**	**1.32**	1000	**2.04**	**1.08**	**1.18**	**1.33**	**1.17**

Table 4.6: Speedup gain of T-BF, T-DS, SPP versus BF-Global, Synthetic dataset.

The 500 query point experiment resulted to slightly different outcome. SPP was the fastest one and outperformed BF-Global by 1.86 times. T-DS and T-BF where slower than BF-Global, and only T-DS could match BF-Global's performance at 1M points. When the query points reached up to 1,000, we noticed that the performance of SPP is continuously ascending and reached to a maximum gain of 2.04 over BF-Global. On the other hand, T-DS and T-BF are much slower than BF-Global. In the synthetic dataset experiment, SPP is much faster than BF-Global, T-BF and T-DS.

SPP is again faster than all the other methods because of the two same reasons exposed in Section 4.3.2.1 (processing part of the reference dataset and execution of the whole algorithm in the GPU device).

4.3.2.3 Real reference points Comparison

We conducted 6 experiments using three different real datasets using groups of 250, 500, 750 and 1,000 query points (Fig. 4.13). In order to compare all the algorithms, we created two reference datasets of 100K and 1M points per every real dataset, by reducing them (the real datasets) uniformly.

All the algorithms except BF-Cublas finished at least one experiment. BF-texture could only complete the one with 100K reference points. BF-Texture and BF-Cublas algorithm failed in all subsequent experiments. The fastest algorithm in the 100K case was SPP, finishing in 49.59ms (Water), 107.49ms (Parks) and 55.24ms (Buildings), while querying 250 points and 53.64ms (Water), 97.54ms (Parks) and 102.84ms (Buildings) while querying 1,000

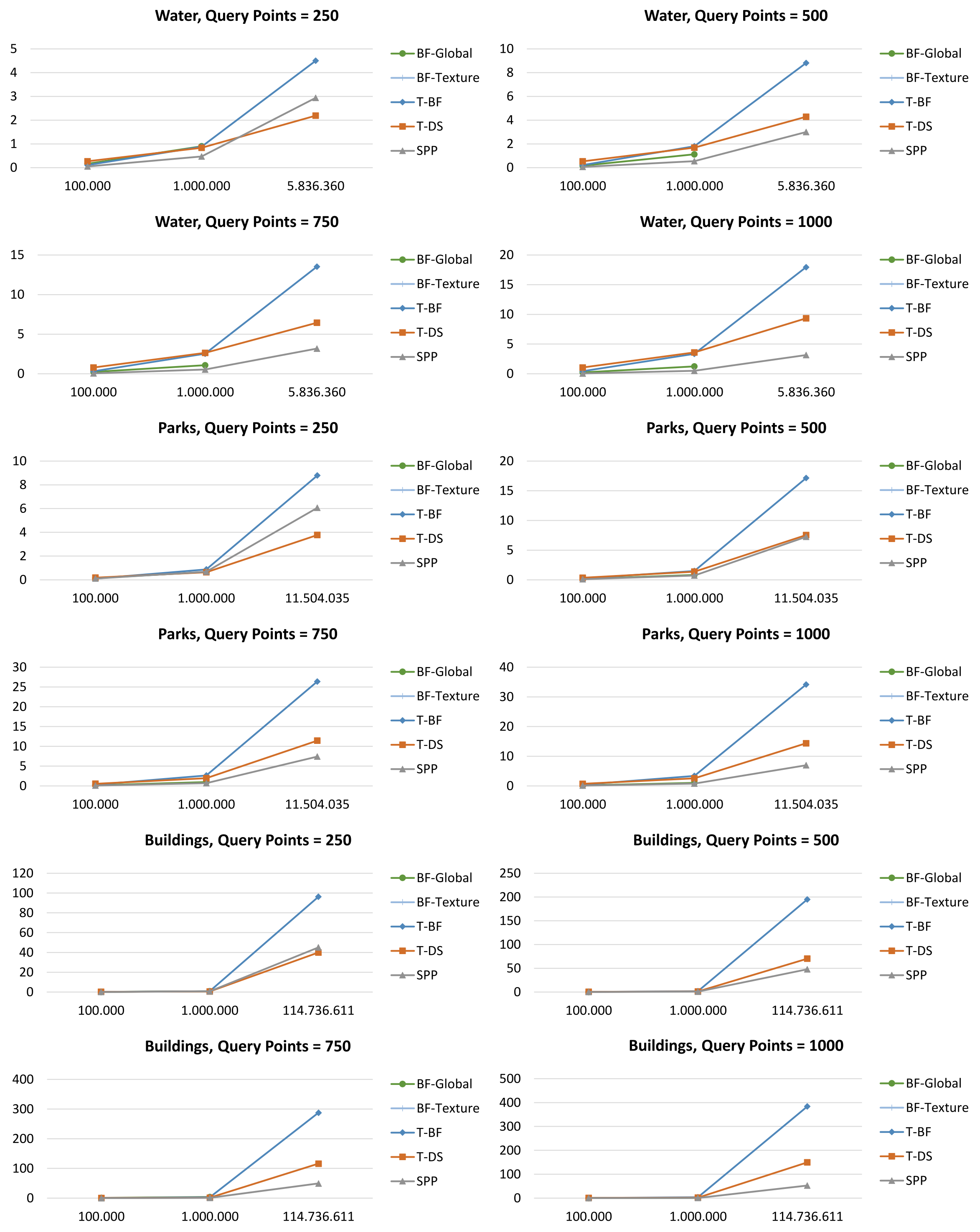


Figure 4.13: Real reference points, $k = 20$. X-axis denotes reference point cardinality, Y-axis denotes execution time measured in sec.

points. In the case of 1M reference points, the T-DS was slightly faster than SPP, only in the 250 query points experiment. When we queried the hole datasets with 1,000 query points, SPP was about 3 times faster than T-DS and about 6 times faster than T-BF.

We could only compare all the methods at the smaller reference datasets of 100K and 1M. SPP had a maximum of 4.87 gain over BF-Global at 1M buildings reference points, when queried with 750 points (Table 4.7). Generally, SPP was over than 2 times faster than BF-Global at the real dataset experiment.

Furthermore, we showed SPP's efficiency in big real datasets, that reach well over than 100M reference points. So SPP is the fastest method in the real dataset experiment.

Again, processing part of the reference dataset and execution of the whole algorithm in the GPU device (as explained in Section 4.3.2.1), makes SPP much faster than all the other methods. Moreover, again the use of KNN-DLB allows SPP to handle much larger reference datasets than BF-Global and AIDW (as explained in Section 4.3.2.1).

	Query	Water Ref. Points		Parks Ref. Points		Buildings Ref. Points	
Algorithm	Points	100K	1M	100K	1M	100K	1M
T-BF	250	1.51	1.02	**1.37**	0.76	1.5	0.89
T-DS	250	0.61	1.08	0.79	**1.04**	0.8	**1.47**
SPP	250	**3.3**	**1.91**	1.34	0.97	**2.86**	1.09
T-BF	500	0.77	0.62	0.65	0.55	0.88	0
T-DS	500	0.31	0.67	0.43	0.6	0.54	0
SPP	500	**2.73**	**2.06**	**1.4**	**1.15**	**2.24**	0
T-BF	750	0.73	0.43	0.51	0.37	2.86	1.44
T-DS	750	0.29	0.41	0.3	0.5	1.57	2.49
SPP	750	**4.42**	**1.99**	**1.78**	**1.38**	**7.72**	**4.87**
T-BF	1000	0.53	0.37	0.44	0.31	0.49	0.33
T-DS	1000	0.22	0.34	0.26	0.4	0.26	0.54
SPP	1000	**4.23**	**2.49**	**1.88**	**1.33**	**1.97**	**1.34**

Table 4.7: Speedup gain of T-BF, T-DS, SPP versus BF-Global, Real datasets.

4.3.3 In-memory, 3rd set of experiments

We run a large set of experiments to compare the repetitive application of our algorithms and the newly implemented one for processing batch k-NN queries. All experiments query 100K and 1M reference points. We have created random and synthetic clustered datasets of 100K and 1M points. Furthermore we created analogously 100K and 1M sample reference points from real world buildings (Buildings Dataset) dataset [1]. For all datasets the 3-dimensional data space is normalized to have unit length (values [0, 1] in each axis).

We run a series of experiments searching for 10, 20, 50 and 100 Nearest Neighbors, using the aforementioned datasets with groups of 100 and 1,000 random query points. By using small and larger groups of query points and a wide range o ks, we will demonstrate the effectiveness of HSPP versus the other two methods. Every experiment was run at least 10 times and the mean of the experiment results were calculated for every method.

4.3.3.1 Experimental evaluation

In our experimental evaluation (Fig.4.14), we used three kinds of reference datasets: random (uniform), synthetic (random clustered) and real. The experiments were conducted firstly with the 100 and then with the 1,000 query dataset.

The first set of experiments where against random reference dataset. HSPP and SPP performed equally at $k = 10$ and $k = 20$, at both reference datasets. At smaller ks than 10, SPP is faster than HSPP, in experiments we conducted and they are not presented in this chapter. T-DS has performed very well,it was very close to the HSPP's performance and was better than SPP with $k = 50$ and $k = 100$. HSPP completed the $k = 50$ and $k = 100$ experiments, finishing at 16ms and 26ms respectively when querying the 100K reference dataset and at 41ms and 67ms when querying the 1M reference dataset. The highest speedup gain of HSPP versus T-DS was approximately 27 times (Table 4.8). HSPP outperformed the other ones in all random dataset experiments.

The second set of experiments where against synthetic reference dataset. At the 100 query point dataset experiment, HSPP was better than the other ones at 100K reference points at all ks. HSPP performed slightly better than T-DS perfomed at $k = 100$. HSPP finished at 29ms, 31ms, 34ms and 36ms for $k = 10, 20, 50, 100$ and 100 query points respectively. At the 1,000 query point dataset experiment, HSPP was even faster than the other. It worth's

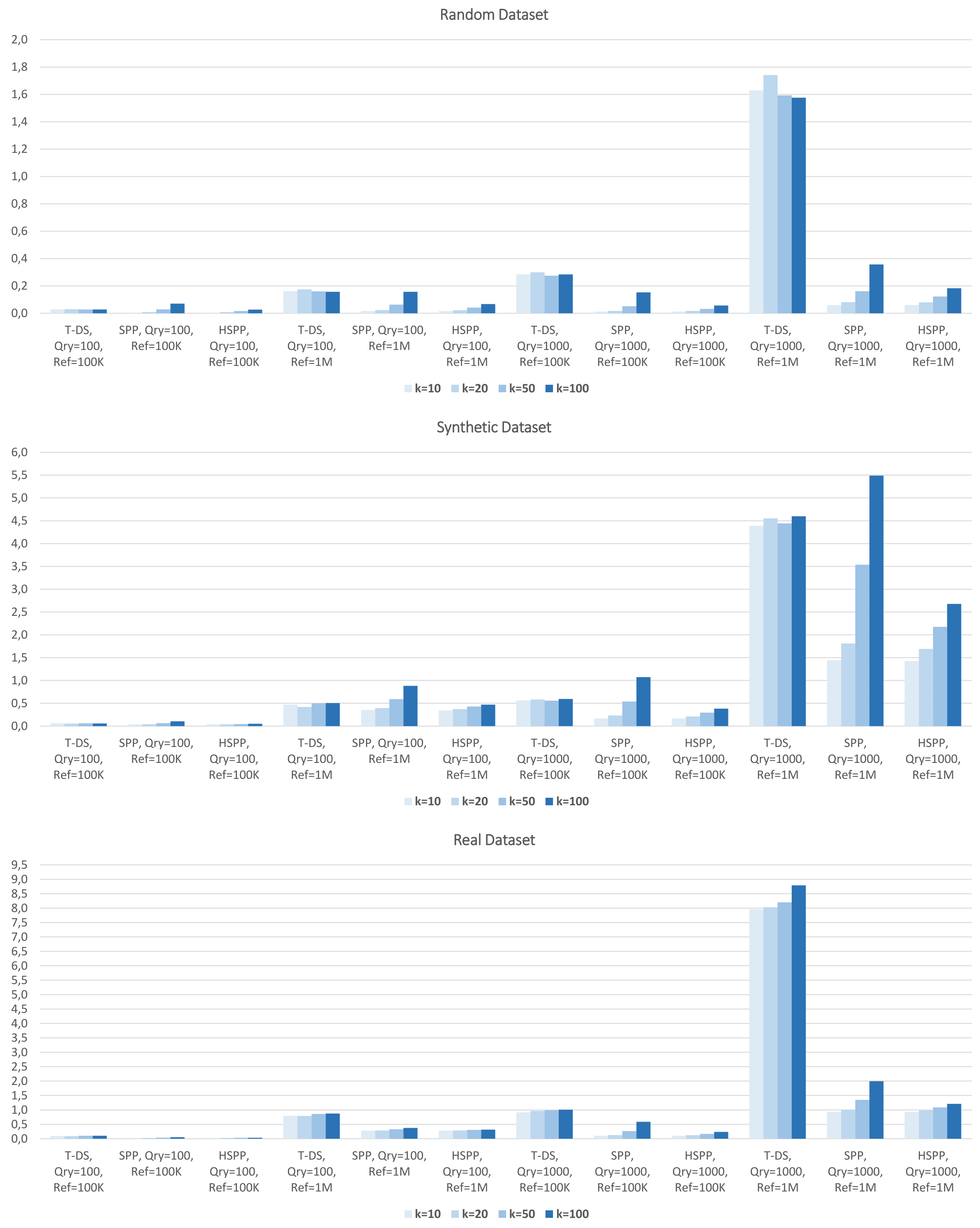


Figure 4.14: Experiment charts. Y-axis is measured in seconds.

			Random				Synthetic				Real			
	Qry	Ref	k=10	k=20	k=50	k=100	k=10	k=20	k=50	k=100	k=10	k=20	k=50	k=100
SPP	100	100K	5,65	3,43	0,97	0,4	1,27	1,19	0,92	0,64	1,69	1,36	0,91	0,54
HSPP	100	100K	5,57	3,43	1,71	1,04	1,28	1,21	1,09	0,98	1,69	1,38	1,28	1,11
SPP	100	1M	9,74	7,4	2,52	1	4,47	3,14	1,31	0,59	1,33	1,07	0,84	0,57
HSPP	100	1M	9,73	7,59	3,88	2,34	4,42	3,18	2,14	1,50	1,37	1,13	1,16	1,07
SPP	1000	100K	24,24	17,54	5,29	1,86	6,69	6,59	4,52	2,59	3,36	2,52	1,04	0,55
HSPP	1000	100K	23,72	17,58	8,61	5,01	6,72	6,58	5,91	4,51	3,4	2,75	1,9	1,56
SPP	1000	1M	27,37	21,5	9,91	4,42	4,4	4,32	4,04	3,51	3,04	2,52	1,26	0,84
HSPP	1000	1M	27,03	21,81	13,05	8,6	4,52	4,4	4,53	4,16	3,07	2,69	2,04	1,72

Table 4.8: Speedup gain of SPP, HSPP vs. T-DS.

noticing that with $k = 100$, HSPP was about 2 times faster than SPP and 4.5 times faster than T-DS. T-DS as we mentioned before, is designed to perform optimal at small groups of query points, thus this difference was expected at larger query datasets. The highest speedup gain of HSPP versus T-DS was 6.72 times (Table 4.8). As a result HSPP outperformed the other ones in all synthetic dataset experiments.

The last set of experiments where against real reference dataset. Using real data resulted again in favor of HSPP. In fact HSPP, once again outperformed T-DS and SPP in all the experiments.

All the experiments resulted to similar conclusions. HSPP is outperforming all the other methods. SPP is better than T-DS but it's performance is deteriorating in a linear way regarding depending on k's increments.

4.4 Conclusions

In this chapter, we presented the first four in-memory algorithms for k-NN query processing in GPUs. These algorithms maximize the utilization of device memory, handling more reference points in the computation. Through an experimental evaluation on synthetic and real datasets, we concluded that T-DS only work faster than existing methods for small groups of query points, SPP outperforms existing methods for larger groups of query points, HSPP further enhances SPP performance for larger k values and all of them scale-up to much larger reference datasets. We validated that T-DS algorithm is faster than T-BS, because of the extra refinement step minimizing the sorting overhead. In terms of memory scaling, SPP and HSPP can compute up to 300M reference points, taking advantage of our KNN-DLB or Max-Heap optimizations. T-DS and T-BF could scale up to 200M and 100M respectively. SPP and HSPP can scale up to 300M points, about 300 times more than other existing algorithms and 100M reference points more than T-DS. HSPP is an overall performance winner, especially for larger k values. The algorithms documented in this chapter were presented in [8, 88, 89].

Chapter 5

$k-$NN Query Processing with GPU and SSD

The GPU device memory is expensive, so it is very important to take advantage of this memory as much as possible and scale-up to larger datasets and avoid the need for distributed processing which suffers from excessive network cost, sometimes overcoming the benefits of distributed parallel execution. 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.

In this Chapter,

- We propose and implement the first (Brute-force and Plane-sweep) GPU-based algorithms for processing the k-NN query on big reference data stored on SSDs.
- We utilize either an array-based, or a max-Heap based buffer for storing the distances of the current k nearest neighbors, which are combined with Brute-force and Plane-sweep techniques, deriving four algorithmic variations.
- Based on 3d synthetic big data, we present an extensive experimental comparison of these algorithmic variations, varying query dataset size, reference dataset size and k and utilizing reference data files which are either presorted in one of the dimensions, or unsorted in all dimensions.
- These experiments highlight that Plane-sweep, combined with either an array or a max-Heap buffer and applied on unsorted reference data, is the performance winner.

The rest of this chapter is organized as follows. In Section 5.1, we review related material and present the motivation for our work. Next, in Section 5.2, we present the new algorithms that we developed for the k-NN GPU-based processing on disk-resident[1] data and in Section 5.3, we present the experimental study that we performed for analyzing the performance of our algorithms and for determining the performance winner among four algorithmic variations tested on presorted and unsorted big reference data. Finally, in Section 5.4, we present the conclusions arising from our work and discuss our future plans.

5.1 Related Work and Motivation

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* (BF) method or by using *indexing data structures*. In the first category, 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. 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. There are different approaches for these two stages and the most representative ones can be found are presented in Section 4.1. Furthermore [88] use heuristics to minimize the reference points near a query point and [8] use of symmetrical partitioning technique with k-NN list buffer. See [8] for a more complete explanation of the Brute-force approaches. In the second category, we can find effective k-NN GPU implementations based on known indexing methods: k-d tree-based [92], grid-based [93], R-tree-based [94], LSH-based [95], etc.

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 [96]. These characteristics include smaller size, lighter weight, lower power consumption, better shock resistance, and faster reads and writes. 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 [50].

[1]We used an SSD and in the rest of the text “SSD” instead of “disk” is used.

To address the necessity of fast nearest neighbor searches on large reference datasets stored in fast secondary devices (SSD), in this chapter, we design and implement efficient k-NN GPU-based algorithms.

5.2 $k-$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 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[2] and loads it into the host memory. The host copies the in-memory partition data into the GPU device memory.

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 4 main steps:

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 describe our new methods. These methods are based on two new main algorithms, "Disk Brute-force" and "Disk Plane-sweep". In both of them we have

[2]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.

implemented two k-NN buffer variations resulting in a total of four new methods (algorithmic variations).

5.2.1 Disk Brute-force Algorithm

The Disk Brute-force algorithm (denoted by DBF) 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 $R = \{r_1, r_2, r_3..r_m\}$ in 3d space and a dataset Q of n query points $Q = \{q_1, q_2, q_3..q_n\}$ 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 (Alg. 7; note that the notation $<<< b, t >>>$ denotes execution using b blocks with t threads each). 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 (Alg. 8) to the query point of the current thread. The first k distances are added to the k-NN buffer of this query point. Every other calculated distance is compared with the current largest one and, if it is smaller, it replaces the current largest one in the k-NN buffer.

The organization and implementation of the k-NN buffer is essential for the effective k-NN calculation, because by using it we elude sorting large distance arrays. The sorting step is extremely demanding, regarding GPU computations. Depending on the algorithm, the CUDA profiler revealed that 90% (or more in large datasets) of the GPU computation may be dedicated to sorting [88]. We will use and compare two alternative k-NN buffer implementations, presented in Sections 4.2.5 and 4.2.6.

5.2.2 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 [97]. The Plane-sweep technique is applied in [98] 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 research) as they are reached by this sweep-line. We can apply this technique for

Algorithm 7 Brute-force Host algorithm

```
Input: NN cardinality=K, Reference filename=RF, Query filename=QF, Partition size=S
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: while not end-of-file RF do
 7:     HostReferencePartition ← readPartition(RF,S);
 8:     DeviceReferencePartition ← HostReferencePartition;
 9:     runKNN<<<(queryPoints-1)/256 +1, 256>>>(DeviceReferencePartition,
        DeviceQueryVector,DeviceKNNBufferVector,K);     // 256 cores assumed
10: HostKNNBufferVector ← DeviceKNNBufferVector;
```

Algorithm 8 Brute-force Device Kernel algorithm (runKNN)

```
Input: NN cardinality=K, Partition Reference array=R, Query array=Q, Partition size=S
Output: Device $k$-NN Buffer array=DKB
1: qIdx ← blockIdx.x*blockDim.x+threadIdx.x;
2: knnBufferOffset ← qIdx*K;
3: for i ← 0 to S-1 do
4:     dist ← $\sqrt[2]{(R[i].x - Q[qIdx].x)^2 + (R[i].y - Q[qIdx].y)^2 + (R[i].z - Q[qIdx].z)^2}$
5:     insertIntoBuffer(DKB,knnBufferOffset,i,qIdx,dist);
6: maxdistance ← calcMaxDistance(DKB);
```

restricting all possible combinations of pairs of objects from the two datasets. The Disk Plane-sweep algorithm (denoted as DSP) 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 $R = \{r_1, r_2, r_3..r_m\}$ in 3d space and a dataset Q of n query points $Q = \{q_1, q_2, q_3..q_n\}$ 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, each bin is transferred to the device memory and sorted by the x-values of its reference points. (Alg. 9). For each partition we apply the k-NN Plane-sweep technique (Fig. 5.1).

Starting from the leftmost reference point of the loaded partition, the sweep-line moves to the right. The sweep-line hops every time to the next reference point until it approaches the x-value of the query point (Fig. 5.1). Using the x-value of the query point, a virtual rectangle is created. This rectangle has a length of $2 * l$, where l is the currently largest k-NN distance in the k-NN buffer of the query point of the current thread.

For every reference point within this rectangle, we calculate the Euclidean distance (Alg.

10) to this query point. The first k distances are added to its k-NN buffer. Every subsequent calculated distance is compared with the largest one in the k-NN buffer and if it is smaller, it replaces the largest one in the k-NN buffer.

In Fig. 5.1, we observe that all the reference points located on the right of the right rectangle limit are not even processed. The reference points located on left of the left rectangle limit are only processed for comparing their x-axis value. The costly Euclidean distance calculation is limited within the rectangle.

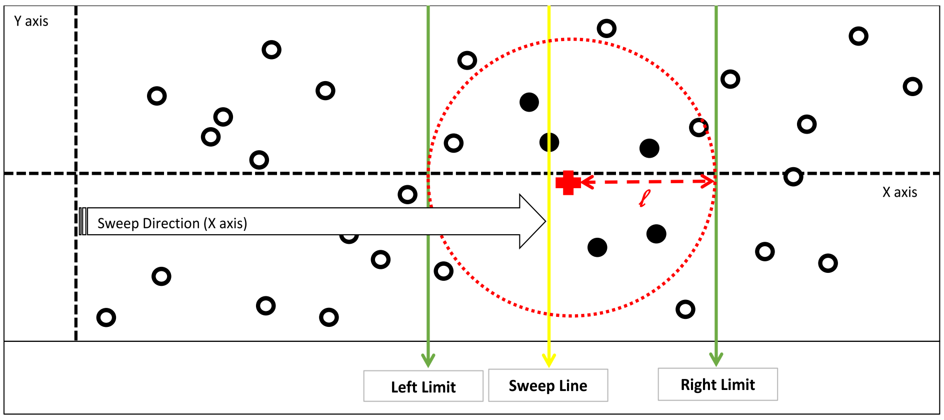


Figure 5.1: Plane-sweep k-NN algorithm. Cross is the Query point, selected reference points in solid circles and not selected reference points in plain circles.

Algorithm 9 Plane-sweep Host algorithm

```
Input: NN cardinality=K, Reference Filename=RF, Query filename=QF, Partition size=S
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: while not end-of-file RF do
7:     HostReferencePartition←readPartition(RF,S);
8:     DeviceReferencePartition←HostReferencePartition;
9:     cudaSort(DeviceReferencePartition);
10:    runKNN<<<(queryPoints-1)/256 +1, 256>>>(DeviceReferencePartition,
       DeviceQueryVector,DeviceKNNBufferVector,K);    // 256 cores assumed
11: HostKNNBufferVector←DeviceKNNBufferVector;
```

Algorithm 10 Plane-sweep Device Kernel algorithm (runKNN)

```
Input: NN cardinality=K, Partition Reference array=R, Query array=Q, Partition size=S, Largest distance in $k$-NN buffer=maxknnDistance
Output: Device $k$-NN Buffer array=DKB
 1: qIdx ← blockIdx.x*blockDim.x+threadIdx.x;
 2: knnBufferOffset ← qIdx*K;
 3: xSweepline ← R[0].x;
 4: i←0;
 5: while xSweepline<Q[qIdx].x and $i < S$ loop do
 6:     inc(i);
 7:     xSweepline ← R[i].x;
 8: leftIdx ← i-1;
 9: while $(R[leftIdx].x - Q[qIdx].x) < maxknnDistance$ and $leftIdx > 0$ do
10:     dec(leftIdx);
11: rightIdx=i;
12: while $(R[rightIdx].x - Q[qIdx].x) < maxknnDistance$ and $leftIdx > 0$ do
13:     inc(rightIdx);
14: for i ← leftIdx to rightIdx do
15:     dist ← $\sqrt[2]{(R[i].x - Q[qIdx].x)^2 + (R[i].y - Q[qIdx].y)^2 + (R[i].z - Q[qIdx].z)^2}$
16:     insertIntoBuffer(DKB,knnBufferOffset,i,qIdx,dist);
```

5.3 Experimental Study

We run a large set of experiments to compare the application of our proposed algorithms. All experiments query at least 500M reference points. We did not include less than 500M reference points because we target reference datasets that do not fit in the device memory. The largest dataset that could fit in device memory in our previous work was 300M [8]. Furthermore, we increased the points accuracy representation from single precision numbers to double precision (Alg. 11) to be able to discriminate among small distance differences.

Algorithm 11 Point Structure

```
   // Point record structure, used in reference datasets. Record size 32 bytes
1: record point_struct begin
2:     id,        // Point ID, type unsigned long long, 8 bytes
3:     x, y, z       // 3 Dimensions, type double, 8 bytes per dimension
4: end;
```

All the datasets were created using the SpiderWeb [99] generator. This generator allows users to choose from a wide range of spatial data distributions and configure the size of the dataset and its distribution parameters. This generator has been successfully used in research work to evaluate index construction, query processing, spatial partitioning, and cost model verification [100].

Table 5.1 lists all the generated datasets. For the reference dataset, we created four datasets using the “Bit” distribution (Fig. 5.2 right), with file sizes ranging from 16GB to 64GB. The reference points dataset size ranges from 500M points to 2G points. For the query points dataset we created one “Uniform” dataset (Fig. 5.2, center) of 10 points and five “Gaussian” datasets (Fig. 5.2, left) ranging from 10K to 50K points.

Table 5.1: SpiderWeb Dataset generator parameters.

Distribution	Size	Seed	File Size	Dataset usage
Bit	500M	1	16GB	Reference
Bit	1G	2	32GB	Reference
Bit	1.5G	3	48GB	Reference
Bit	2G	4	64GB	Reference
Uniform	10	5	32B	Query
Gaussian	10K	6	320KB	Query
Gaussian	20K	7	640KB	Query
Gaussian	30K	8	960KB	Query
Gaussian	40K	9	1,3MB	Query
Gaussian	50K	10	1,6MB	Query

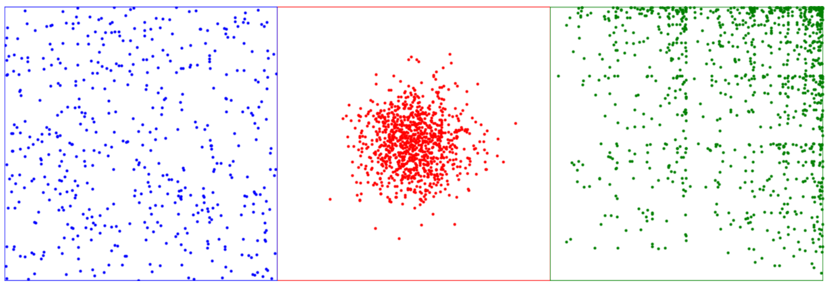


Figure 5.2: Experiment distributions, Left=Uniform, Middle=Gaussian, Right=Bit .

Three different sets of experiments were conducted. In the first one, we scaled the reference dataset size, in the second one we scaled the query dataset size and in the last one we scaled the number of the nearest neighbors, k. For every set, we used presorted and unsorted reference datasets to evaluate their effect on the methods’ performance. We also evaluated the performance of the two alternative list buffers to clarify the pros and cons of using KNN-DLB and max-Heap buffer.

All experiments were performed on a Dell G5 15 laptop, running Ubuntu 20.04, equipped with a six core (12-thread) Intel I7 CPU, 16GB of main memory, a 1TB SSD disk used and a NVIDIA Geforce 2070 (Mobile Max-Q) GPU with 8GB of device memory. CUDA version 11.2 was used.

We run experiments to compare the performance of k-NN queries regarding execution time, as well as memory utilization. We tested a total of four algorithms, listed in the following.

1. DBF, Disk Brute-force using KNN-DLB buffer
2. DBF Heap, Disk Brute-force using max Heap buffer
3. DPS, Plane-sweep using KNN-DLB buffer
4. DPS, Plane-sweep using max Heap buffer

To the best of our knowledge, these are the first methods to address the k-NN query on SSD-resident data.

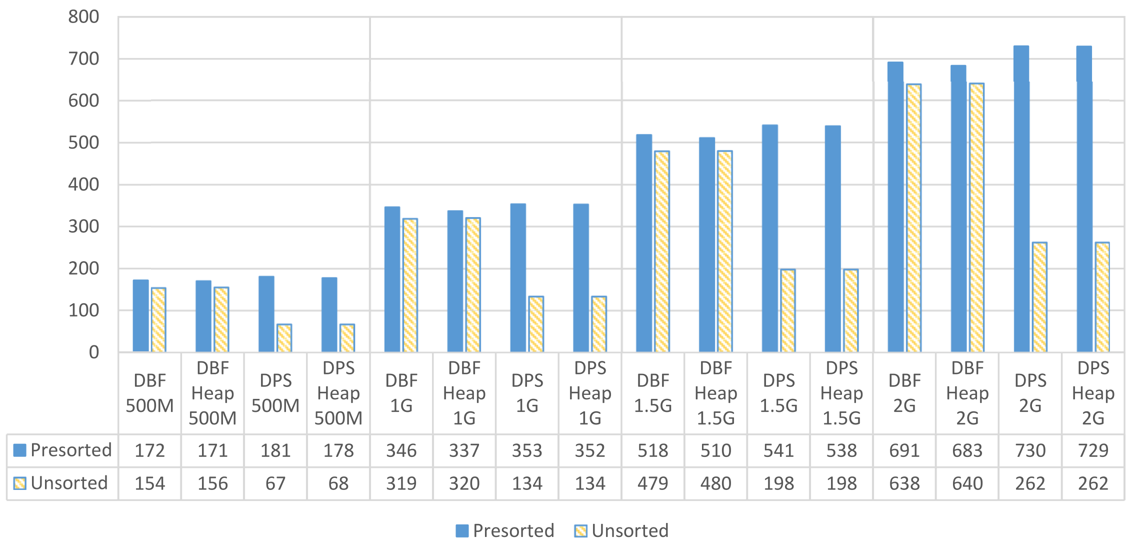
5.3.1 Reference dataset scaling

In our first series of tests, we used the “Bit” distribution synthetic datasets for the reference points. The size of the reference point dataset ranged from 500M points to 2G 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. 5.3, we can see the presorted dataset results in blue and the unsorted dataset results in stripped yellow. In the presorted experiment, we notice that the execution time of all methods is quite similar, for each reference dataset size. For example, for the 500M dataset the execution time is 172 sec. for DBF, 171 second for DBF Heap, 181 for DPS and 178 sec. for DPS Heap. The execution times increase proportionally to the reference dataset size. As expected, we get the slowest execution times for the 2G dataset, 691 sec. for both DBF, 683 sec. for DBF Heap, 730 sec. for DPS and 729 for DPS Heap.

In the unsorted dataset experiments, we observe that all execution times are smaller, especially for the Plane-sweep methods. The Brute-force methods are slightly faster in the unsorted dataset experiments than in the presorted ones. For the 500M unsorted dataset, the execution time for DBF is 154 sec., for DBF Heap is 156 sec., for DPS 67 sec. and for DPS Heap just 68 sec.. Once again, the execution times increase proportionally to the reference dataset size. For the 2G unsorted dataset, we get 638 sec. for DBF, 640 sec. for DBF Heap, 262 sec. for DPS and 262 for DPS Heap.

The reference dataset scaling experiments reveal that all the methods performed better for the unsorted dataset. For the unsorted dataset, the Brute-force methods performed slightly better, but the Plane-sweep methods performed exceptionally better than for the presorted dataset. Furthermore, the Plane-sweep methods were more than 1.7 times faster than Brute-force ones, in the unsorted dataset experiments.



	DBF 500M	DBF Heap 500M	DPS 500M	DPS Heap 500M	DBF 1G	DBF Heap 1G	DPS 1G	DPS Heap 1G	DBF 1.5G	DBF Heap 1.5G	DPS 1.5G	DPS Heap 1.5G	DBF 2G	DBF Heap 2G	DPS 2G	DPS Heap 2G
Presorted	172	171	181	178	346	337	353	352	518	510	541	538	691	683	730	729
Unsorted	154	156	67	68	319	320	134	134	479	480	198	198	638	640	262	262

Figure 5.3: Reference scaling experiment (Y-axis in sec.).

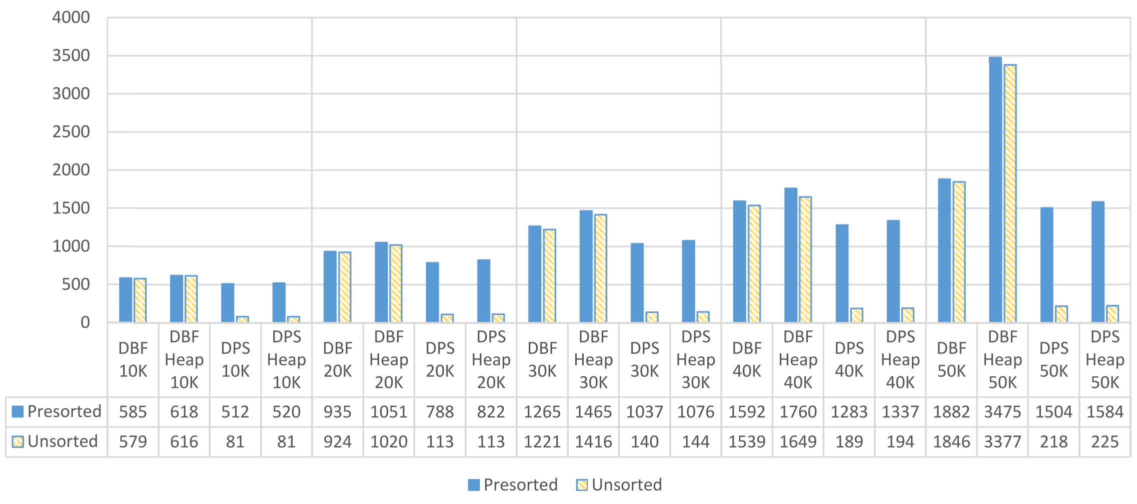
5.3.2 Query dataset scaling

In our second set of experiments, we used between 10K and 50K query points with “Gaussian” distribution. For the reference points we used a 500M “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. 5.4, 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. Depending on the query dataset size, we observe that the execution time gradually increases. For the 10K dataset, the execution time is 585 sec. for DBF, 618 second for DBF Heap, 512 for DPS and 520 sec. for DPS Heap. The slowest execution times were recorded for the 50K dataset, 1882 sec. for DBF, 3475 sec. for DBF Heap, 1504 sec. for DPS and 1584 for DPS Heap.

In the unsorted experiments, we observe once more that all execution times are smaller, especially for the Plane-sweep methods. The Brute-force methods are slightly faster for the unsorted dataset than for the presorted one. For the 10K unsorted query dataset, the execution time for DBF is 579 sec., for DBF Heap is 616 sec., for DPS 81 sec. and for DPS Heap also 81 sec.. Once again, the execution times increase proportionally to the query dataset size. For the 50K unsorted query dataset, we get 1846 sec. for DBF, 3377 sec. for DBF Heap, 218 sec. for DPS and 225 for DPS Heap.

The results of the query dataset scaling experiments conform with the ones of the reference scaling experiments. All the methods performed better with the unsorted dataset. For the unsorted dataset, the Brute-force methods performed slightly better, but the Plane-sweep methods performed once again exceptionally better than for the presorted dataset. Furthermore, the Plane-sweep methods were more than 7 to 15 times faster than Brute-force ones, in the unsorted dataset experiments.



	DBF 10K	DBF Heap 10K	DPS 10K	DPS Heap 10K	DBF 20K	DBF Heap 20K	DPS 20K	DPS Heap 20K	DBF 30K	DBF Heap 30K	DPS 30K	DPS Heap 30K	DBF 40K	DBF Heap 40K	DPS 40K	DPS Heap 40K	DBF 50K	DBF Heap 50K	DPS 50K	DPS Heap 50K
■ Presorted	585	618	512	520	935	1051	788	822	1265	1465	1037	1076	1592	1760	1283	1337	1882	3475	1504	1584
▨ Unsorted	579	616	81	81	924	1020	113	113	1221	1416	140	144	1539	1649	189	194	1846	3377	218	225

Figure 5.4: Query scaling experiment (Y-axis in sec.).

5.3.3 k scaling

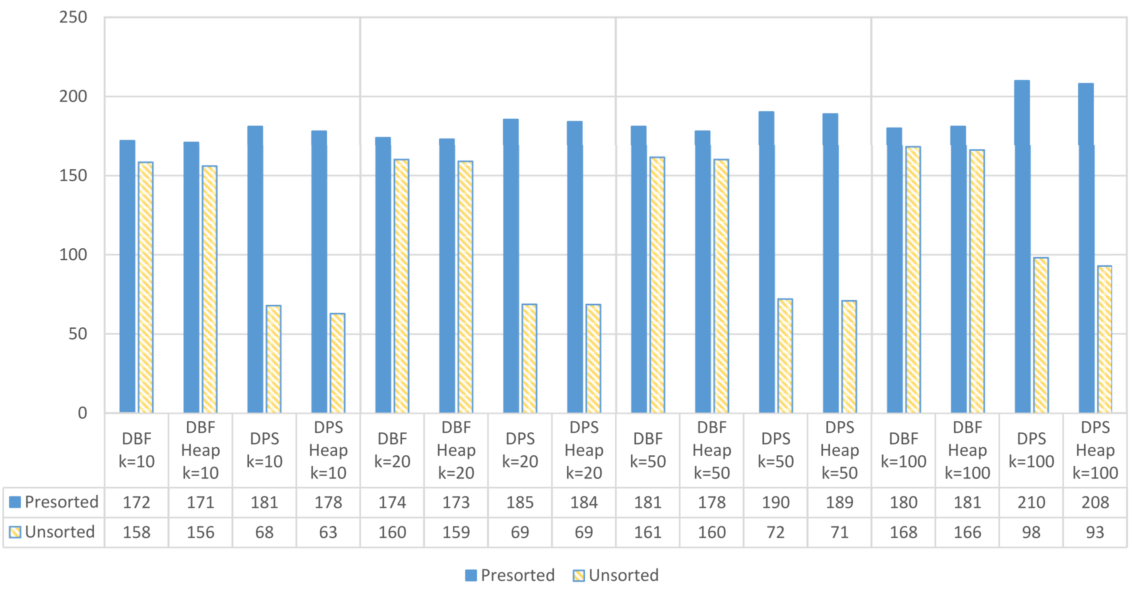
The k scaling is our last set of experiments. In these tests we used k values of 10,20,50 and 100. For the reference points we used the 500M “Bit” distribution synthetic dataset and a small query group of 10 points, with “Uniform” distribution, in order to focus only on the k scaling.

In Fig. 5.5, 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 slightly smaller than the Plane-sweep ones. Depending on the k value, we observe that the execution time increases slightly for larger k values. For k equal to 10 the execution time is 172 sec. for DBF, 171 second for DBF Heap, 181 for DPS and 178 sec. for DPS Heap. The slowest execution times were recorded for k equal to 100, 180 sec. for DBF, 181 sec. for DBF Heap, 210 sec. for DPS and 208 for DPS Heap.

In the unsorted experiments, we observe once more that the execution is faster, especially for the Plane-sweep methods. The Brute-force methods are slightly faster for the unsorted dataset than for the presorted one. For k equal to 10 and the unsorted query dataset, the execution time for DBF is 158 sec., for DBF Heap is 156 sec., for DPS 68 sec. and for DPS Heap also 63 sec.. Once again, the execution times increase proportionally to the k value. For the $k = 100$, we get 168 sec. for DBF, 166 sec. for DBF Heap, 98 sec. for DPS and 93 for DPS Heap.

The results of the k scaling experiments also conform with the results of the previous experiments. All the methods performed better with the unsorted dataset. For the unsorted dataset, the Brute-force methods performed slightly better, but the Plane-sweep methods performed once again exceptionally better than for the presorted dataset. Furthermore, the Plane-sweep methods were about 2 times faster than Brute-force ones, in the unsorted dataset experiments. Although, the two k-NN list buffer methods were shown equal, for even larger k values than the ones studied in this chapter, the k max-Heap list buffer is expected to outperform the KNN-DLB one.



	DBF k=10	DBF Heap k=10	DPS k=10	DPS Heap k=10	DBF k=20	DBF Heap k=20	DPS k=20	DPS Heap k=20	DBF k=50	DBF Heap k=50	DPS k=50	DPS Heap k=50	DBF k=100	DBF Heap k=100	DPS k=100	DPS Heap k=100
■ Presorted	172	171	181	178	174	173	185	184	181	178	190	189	180	181	210	208
▨ Unsorted	158	156	68	63	160	159	69	69	161	160	72	71	168	166	98	93

Figure 5.5: k scaling experiment (Y-axis in sec.).

5.3.4 Interpretation of Results

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 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. 5.6 left), 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 k-th NN found so far (Fig. 5.6 right).

5.4 Conclusions

In this chapter, we presented the first GPU-based algorithms for parallel processing the k-NN query on reference data stored on SSDs, utilizing the Brute-force and Plane-sweep techniques. These algorithms exploit the numerous GPU cores, utilize the device memory as much as possible and take advantage of the speed and storage capacity of SSDs, thus processing efficiently big reference datasets. Through an experimental evaluation on synthetic datasets, we highlighted that Plane-sweep on unsorted reference data (with either an array or a max-Heap buffer for organizing the current k-NNs) is a clear performance winner. The algorithms documented in this chapter were presented in [90].

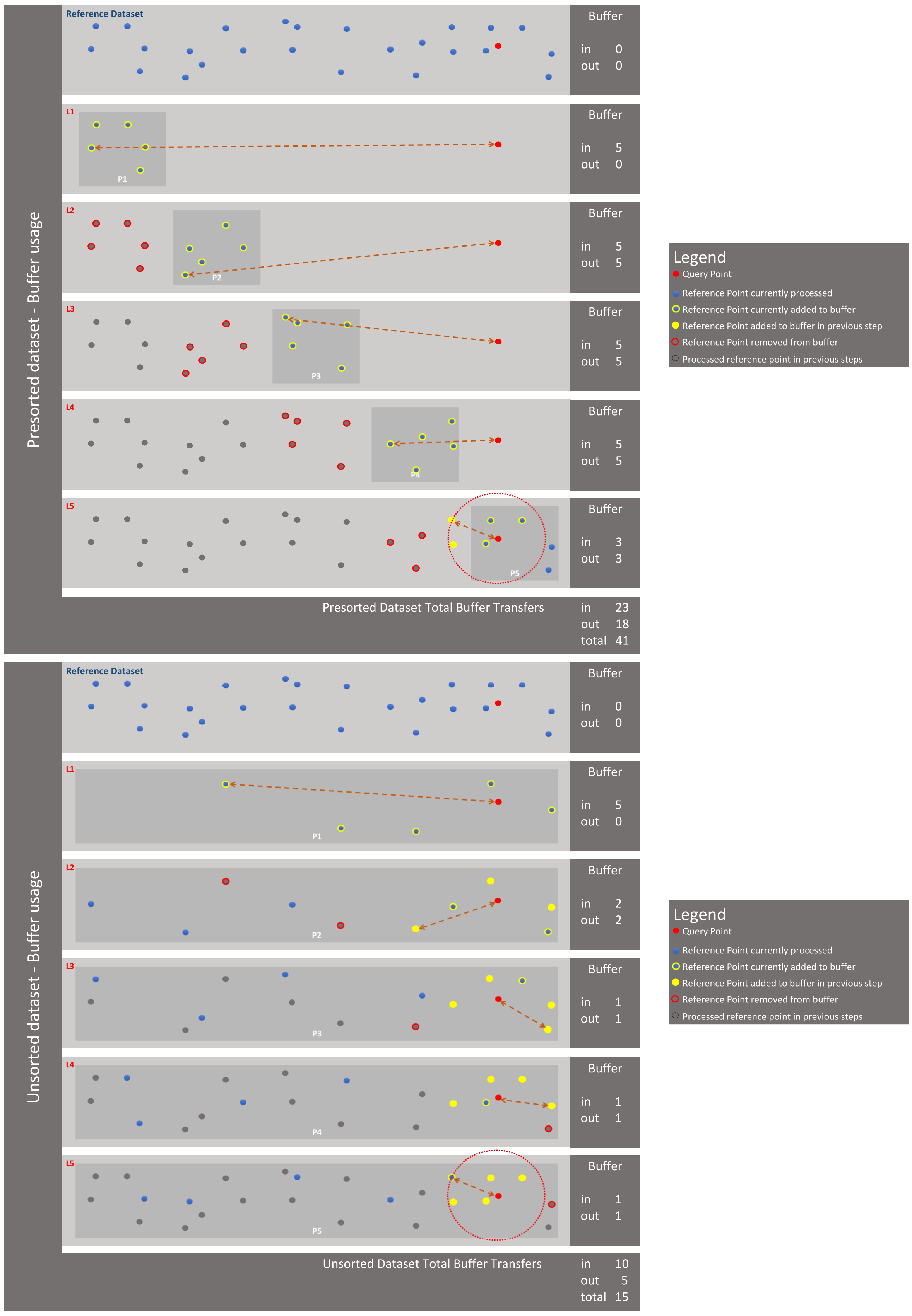

Presorted dataset - Buffer usage
Reference Dataset
Buffer in 0 out 0
L1
P1
Buffer in 5 out 0
L2
P2
Buffer in 5 out 5
L3
P3
Buffer in 5 out 5
L4
P4
Buffer in 5 out 5
L5
P5
Buffer in 3 out 3
Presorted Dataset Total Buffer Transfers
in 23 out 18 total 41
Legend
Query Point
Reference Point currently processed
Reference Point currently added to buffer
Reference Point added to buffer in previous step
Reference Point removed from buffer
Processed reference point in previous steps
Unsorted dataset - Buffer usage
Reference Dataset
Buffer in 0 out 0
L1
P1
Buffer in 5 out 0
L2
P2
Buffer in 2 out 2
L3
P3
Buffer in 1 out 1
L4
P4
Buffer in 1 out 1
L5
P5
Buffer in 1 out 1
Unsorted Dataset Total Buffer Transfers
in 10 out 5 total 15
Legend
Query Point
Reference Point currently processed
Reference Point currently added to buffer
Reference Point added to buffer in previous step
Reference Point removed from buffer
Processed reference point in previous steps

Figure 5.6: Presorted versus unsorted reference dataset buffer update.

Chapter 6

$k-$NN Query Processing with IoT Edge Devices and SSD

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 network's 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 network's 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 during last years [72].

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.

In [8], we presented a new in-memory GPU-based algorithm for the k-NN query, using the CUDA run-time API [73]. 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 chapter:

- 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 [8], 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 chapter has two variations, one which uses the same buffer as [8] and another which uses a max-Heap distance list buffer (as proposed in [89, 90]).

- 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 [101].

- 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 [90] for GPU-based processing of the same query on disk resident data. Our algorithms in [90] 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 chapter is organized as follows. In Section 6.1, we review related material and present the motivation for our work, while in Section 6.2 we present our proposal of the distributed edge-computing environment architecture for processing queries like the k-NN one. Next, in Section 6.3, we briefly present the algorithms of [90], 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 6.4, 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 [90]. Finally, in Section 6.5, we present the conclusions arising from our work and discuss our future plans.

6.1 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.

6.1.1 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 [102]. 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.

[103] 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 [74], 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.

[76] proposed the *CUKNN* algorithm, a CUDA based parallel implementation of k-NN. It used the same approach as [103] and [74] 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 [75], 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 [103] is applied when each column of the distance matrix is sorted.

In [81], 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 [78], 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 [79], the *GPU-FS-kNN* algorithm is presented. It divides the computation of the distance matrix into smaller sub-matrices (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.

[83] 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 [77], a hybrid parallelization approach for Brute-Force computation of multiple k-NN queries on GPUs is proposed. For the matrix computation uses the [74] and [75] 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 [85], 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 [80], 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 [82], 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 [104], 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 [75] and their improved implementations [87]) consume a lot of device memory, since a Cartesian product matrix, containing the distances of reference points to the query points, is stored. In [88], two new algorithms based on GPUs to process k-NN queries on spatial data are proposed, using the Thrust library [72], 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.

6.1.2 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 [105] 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 [106], 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 [92], 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) [107] 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 [95, 108], 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 [109], 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 [110] are promising in speeding up such computing on GPUs; therefore, several papers have been proposed for this purpose. The most significant one is [62], 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 [94], a parallel bottom-up construction of SS-tree [111] 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 [93], 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 [91], the Adaptive Inverse Distance Weighting (AIDW) interpolation algorithm on GPU is presented, where a fast k-NN search approach based on an even grid is used.

Effective spatial data partitioning [112] 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 [8], 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.

6.1.3 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 [96]. 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 appealing 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 [50].

In this chapter, we significantly extend our work in [8] adapting it to reference data stored on SSD storage. We present a new algorithm, called “Disk Symmetric Progression Partitioning” (Section 6.3.3), 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 4.2.5), as in [8], or a max-Heap based buffer (Section 4.2.6), as in [89]. To the best of our knowledge, our recent paper [90] and [113] 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 chapter, 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 ([87, 91]) and algorithms developed by us ([90]). 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: [114], where an evaluation of clustering algorithms on GPU-based modern edge computing platforms is presented; [115, 116], where the computational capability of up-to-date accelerator-based edge devices in the context of scientific computing is evaluated; and [117], where a deep learning benchmark on the latest GPU-accelerated edge devices to measure its performance is proposed.

6.2 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. 6.1. 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. 6.1, 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.

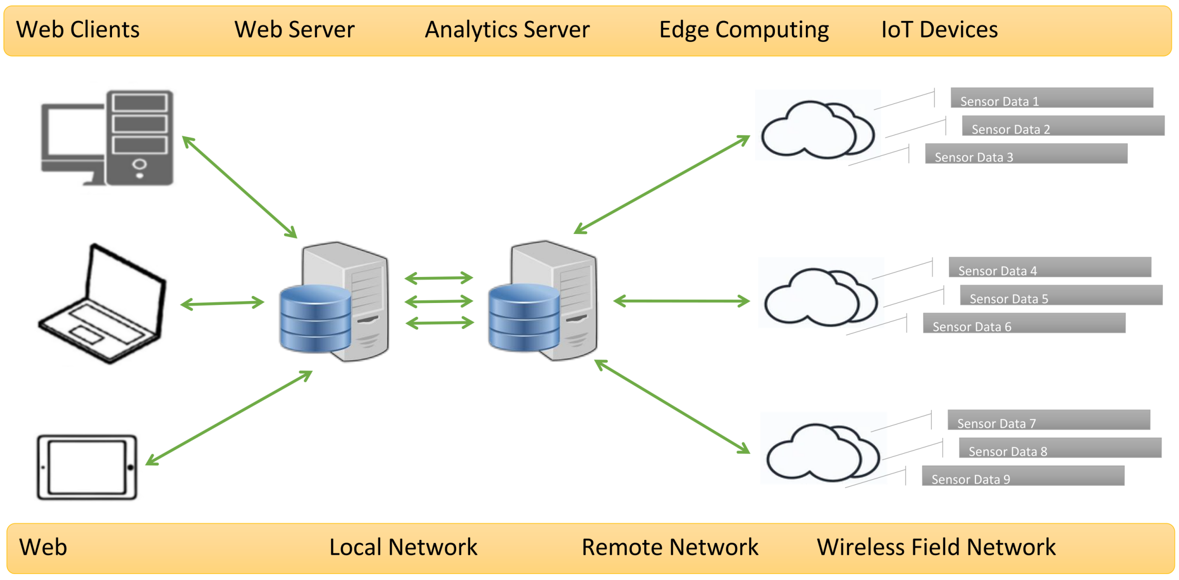


Figure 6.1: Edge Computing Architecture.

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 chapter, 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.

6.3 $k-$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. 6.2).

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. 4.3):

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.

[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.

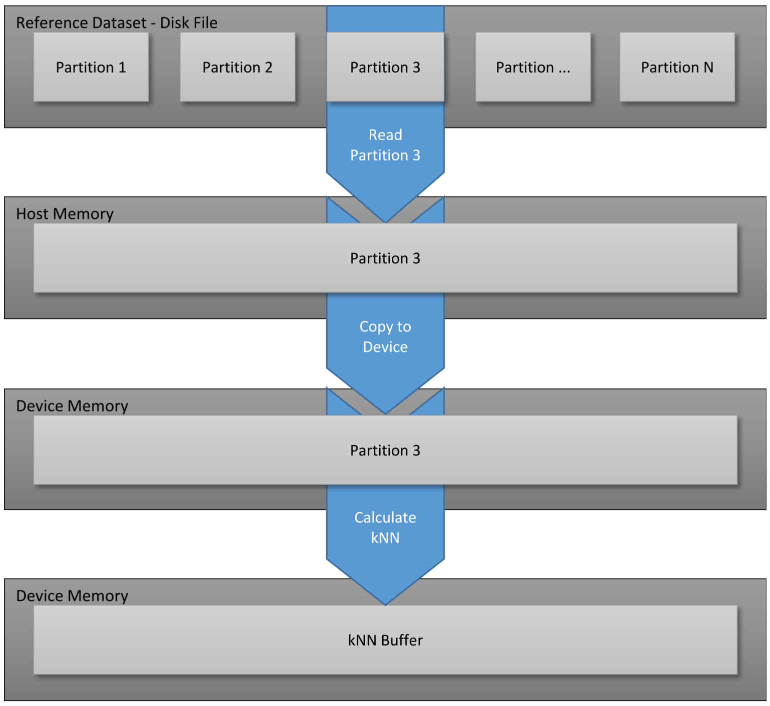


Figure 6.2: Datafile partitioning and loading of partitions into device's memory.

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" [90]. The new one is based on the "Symmetric Progression Partitioning Algorithm" [8]. 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).

6.3.1 Disk Brute-force Algorithm

The Disk Brute-force algorithm (denoted by DBF) [90] 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 $R = \{r_1, r_2, r_3..r_m\}$ in 3d space and a dataset Q of n query points $Q = \{q_1, q_2, q_3..q_n\}$ 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 4.2.5 and 4.2.6.

6.3.2 Disk Plane-sweep Algorithm

An important improvement for join queries is the use of the Plane-sweep technique, which is presented in Section 5.2.2.

Like DBF, DPS accepts as inputs a reference dataset R consisting of m reference points $R = \{r_1, r_2, r_3..r_m\}$ in 3d space and a dataset Q of n query points $Q = \{q_1, q_2, q_3..q_n\}$ 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, 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.

6.3.3 Disk Symmetric Progression Partitioning

We designed and implemented the novel method “Disk Symmetric Progression Partitioning” [113], 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 6.3. 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 DPS, DSPP accepts as inputs a reference dataset R consisting of m reference points $R = \{r_1, r_2, r_3..r_m\}$ in 3d space and a dataset Q of n query points $Q = \{q_1, q_2, q_3..q_n\}$ 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 (Alg. 12). The host 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.

Algorithm 12 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
18: HostKNNBufferVector ← DeviceKNNBufferVector;
```

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. 4.3):

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 (Alg. 13), that the host provided, and checks if its bounding box contains the query point (sub-partition number 5, in our example, Fig. 4.4). 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. 4.4 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.

Algorithm 13 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 then break;
 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;
```

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.

6.4 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 (Alg. 14). Another parameter that we evaluated is the list buffer performance and we will highlight the pros and cons of using KNN-DLB or max Heap buffer.

Algorithm 14 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;
```

All the datasets were created using the SpiderWeb [99, 100] 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 [100]. 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 6.1 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. 6.3), 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.

Table 6.2 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. 7.3), 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.

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

Table 6.1: SpiderWeb Dataset generator parameters, for the existing algorithms comparison experiment.

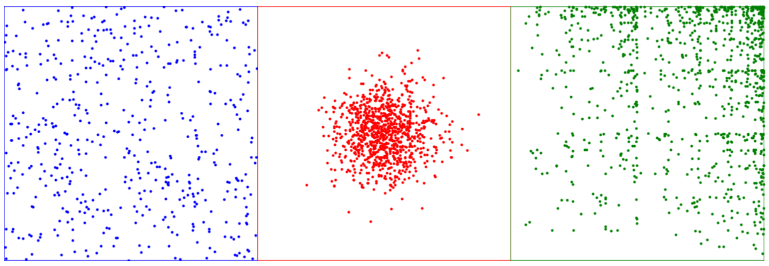


Figure 6.3: Experimental data distributions, Blue=Uniform, Red=Gaussian, Green=Bit.

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 [90] and a new one, presented in this chapter) 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 [90], Disk Brute Force using KNN-DLB buffer.
2. DBF Heap [90], Disk Brute Force using max Heap buffer.
3. DPS [90], Plane Sweep using KNN-DLB buffer.
4. DPS Heap [90], Plane Sweep using max Heap buffer.
5. DSPP, Symmetric Progression Partitioning using KNN-DLB buffer.
6. DSPP Heap, Symmetric Progression Partitioning max Heap buffer.

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 6.2: SpiderWeb Dataset generator parameters, for the new methods scaling experiments.

In order to compare our methods' performance, we will compare them with existing in-memory ones. This is because, apart from the methods of [90], 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 [87], Brute Force algorithm using only the CPU.
2. Garcia GPU BF [87], Brute Force algorithm using the GPU.
3. AIDW [91], 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.

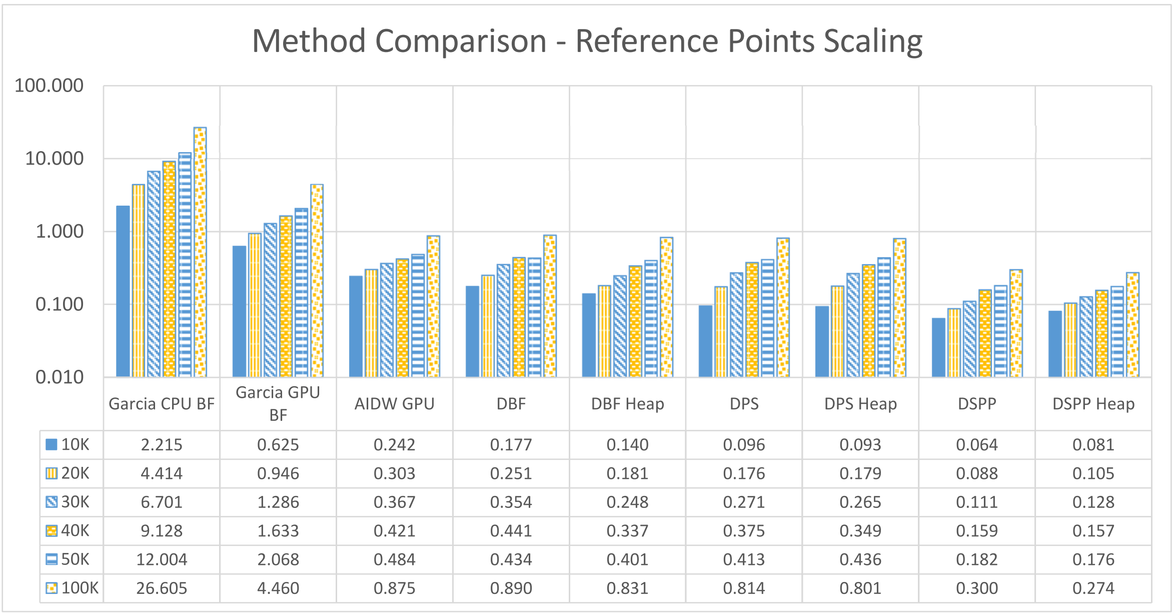
6.4.1 Comparison to existing methods

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 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. 6.4, 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.

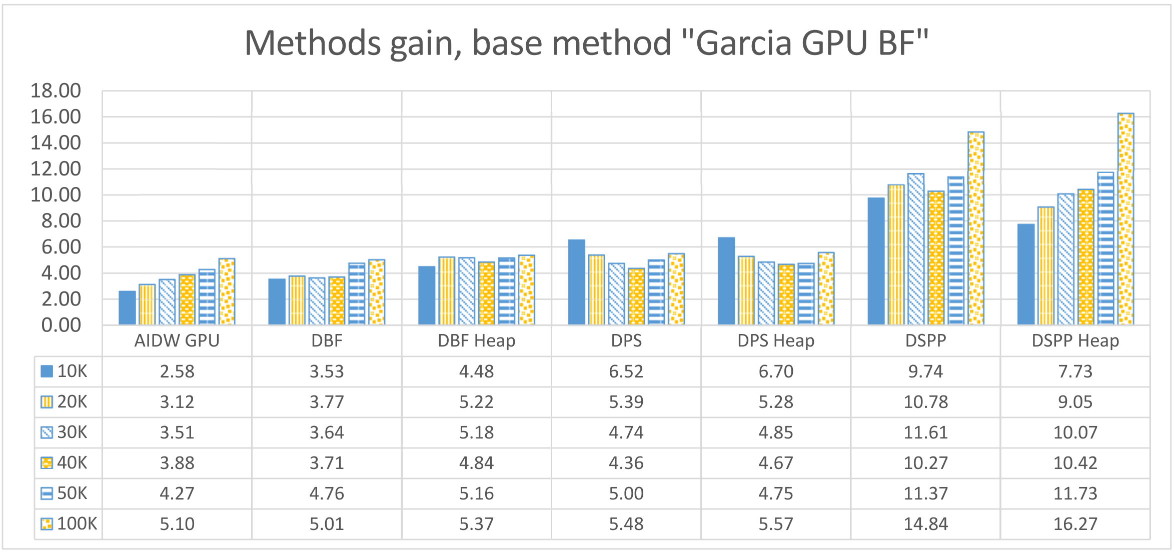
In the rest of this section we will continue by comparing the results of the GPU-based methods only. In Fig. 6.5 we depict the execution time gain of each such algorithm against one (denoted in each relevant figure) which is considered the comparison baseline (for each reference dataset volume, we divide 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 seconds, achieving the fastest execution time in our comparison experiment. DSPP heap follows with a slightly slower time of 0.081 seconds, DPS Heap with 0.093 seconds, DPS with 0.096 seconds, DBF Heap with 0.140 seconds, DBF with 0.177 seconds, AIDW with 0.242 and the slowest one is Garcia GPU BF with 0.625 seconds. As a result DSPP is



	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

Figure 6.4: Experiment Comparison. All values are measured in seconds. The Y axis is in logarithmic scale.

about 9.7 times faster than the Garcia GPU BF method (baseline one).



	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

Figure 6.5: Experiment Comparison, base method DBF.

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 seconds for DSPP Heap, 0.300 seconds for DSPP, 0.801 seconds for DPS Heap, 0.814 seconds for DPS, 0.831 seconds for DBF Heap, 0875 seconds for AIDW GPU, 0.890 for DBF and 4.460 seconds for

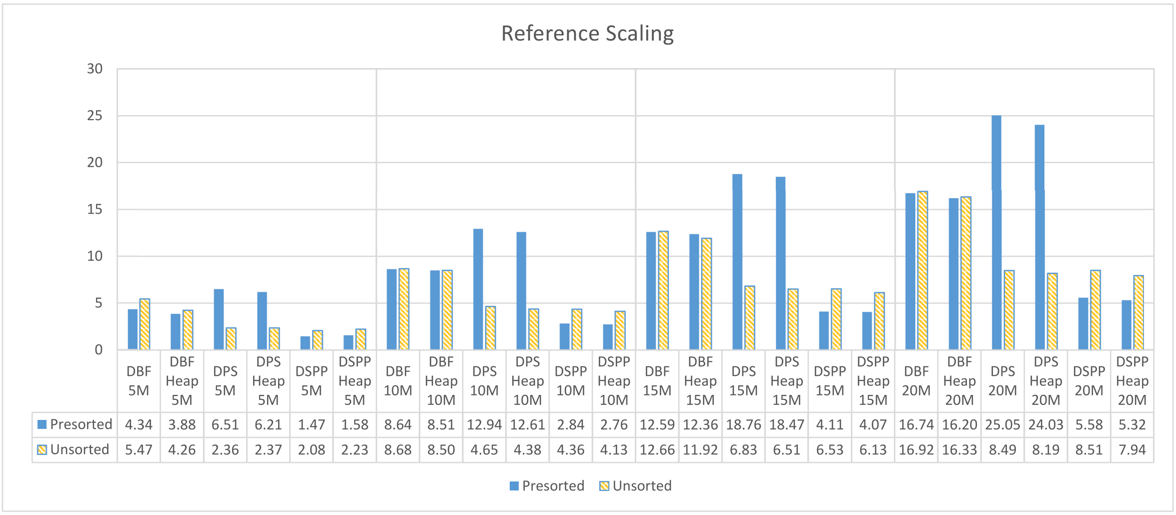
Garcia GPU BF. DSPP Heap is much faster than the other methods, resulting to a 16.27 gain (times faster) when compared to the Garcia GPU BF method.

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.

6.4.2 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. 6.6, 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 seconds for DBF, 3.88 second for DBF Heap, 6.51 for DPS, 6.21 seconds for DPS Heap, 1.47 seconds 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 seconds for DBF, 16.20 for DBF Heap, 25.05 seconds for DPS, 24.03 seconds for DPS Heap, 5.58 seconds for DSPP and 5.32 for DSPP Heap.



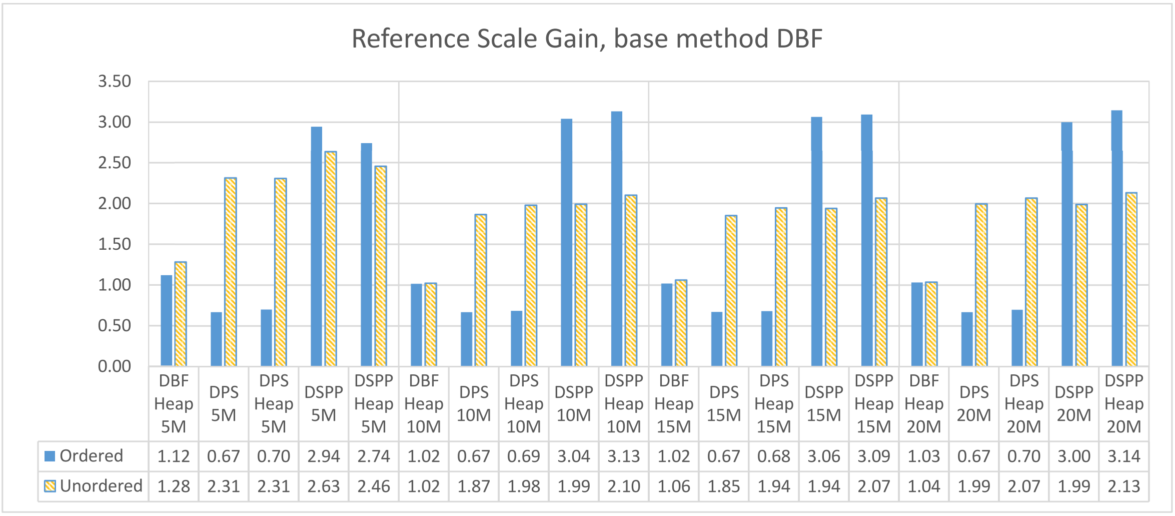
	DBF 5M	DBF Heap 5M	DPS 5M	DPS Heap 5M	DSPP 5M	DSPP Heap 5M	DBF 10M	DBF Heap 10M	DPS 10M	DPS Heap 10M	DSPP 10M	DSPP Heap 10M
■ Presorted	4.34	3.88	6.51	6.21	1.47	1.58	8.64	8.51	12.94	12.61	2.84	2.76
▨ Unsorted	5.47	4.26	2.36	2.37	2.08	2.23	8.68	8.50	4.65	4.38	4.36	4.13

	DBF 15M	DBF Heap 15M	DPS 15M	DPS Heap 15M	DSPP 15M	DSPP Heap 15M	DBF 20M	DBF Heap 20M	DPS 20M	DPS Heap 20M	DSPP 20M	DSPP Heap 20M
■ Presorted	12.59	12.36	18.76	18.47	4.11	4.07	16.74	16.20	25.05	24.03	5.58	5.32
▨ Unsorted	12.66	11.92	6.83	6.51	6.53	6.13	16.92	16.33	8.49	8.19	8.51	7.94

Figure 6.6: Reference scaling experiment (Y-axis in seconds).

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 seconds, for DBF Heap is 4.26 seconds, for DPS 2.36 seconds, 2.37 seconds for DPS Heap, 2.08 from DSPP and 2.23 seconds for DSPP Heap. Once again, the execution times scale analogously to the reference dataset size. For the 20M unsorted dataset, we get 16.92 seconds for DBF, 16.33 seconds for DBF Heap, 8.49 seconds for DPS,8.19 seconds for DPS Heap, 8.51 seconds 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. 6.7) than Brute-force one, in both dataset experiments.



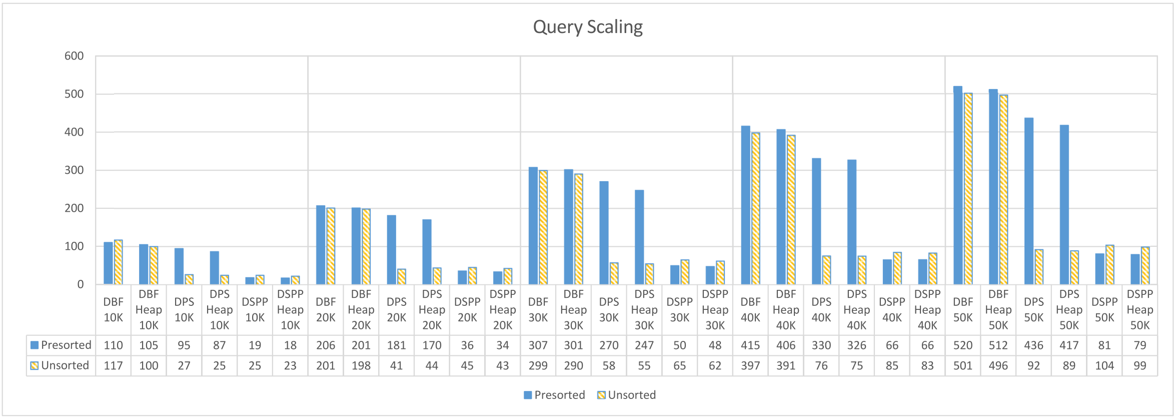
	DBF Heap 5M	DPS 5M	DPS Heap 5M	DSPP 5M	DSPP Heap 5M	DBF Heap 10M	DPS 10M	DPS Heap 10M	DSPP 10M	DSPP Heap 10M	DBF Heap 15M	DPS 15M	DPS Heap 15M	DSPP 15M	DSPP Heap 15M	DBF Heap 20M	DPS 20M	DPS Heap 20M	DSPP 20M	DSPP Heap 20M
■ Ordered	1.12	0.67	0.70	2.94	2.74	1.02	0.67	0.69	3.04	3.13	1.02	0.67	0.68	3.06	3.09	1.03	0.67	0.70	3.00	3.14
▨ Unordered	1.28	2.31	2.31	2.63	2.46	1.02	1.87	1.98	1.99	2.10	1.06	1.85	1.94	1.94	2.07	1.04	1.99	2.07	1.99	2.13

Figure 6.7: Reference scaling experiment gain, base method DBF.

6.4.3 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. 6.8, 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 seconds for DBF, 105 second for DBF Heap, 95 for DPS, 87 seconds for DPS Heap and 19 for DSPP and 18 for DSPP Heap. The slowest execution times were recorded for the 50K query dataset, 520 seconds for DBF, 512 seconds for DBF Heap, 436 seconds for DPS, 417 seconds for DPS Heap, and 81 seconds for both DSS and 79 for DSPP Heap.

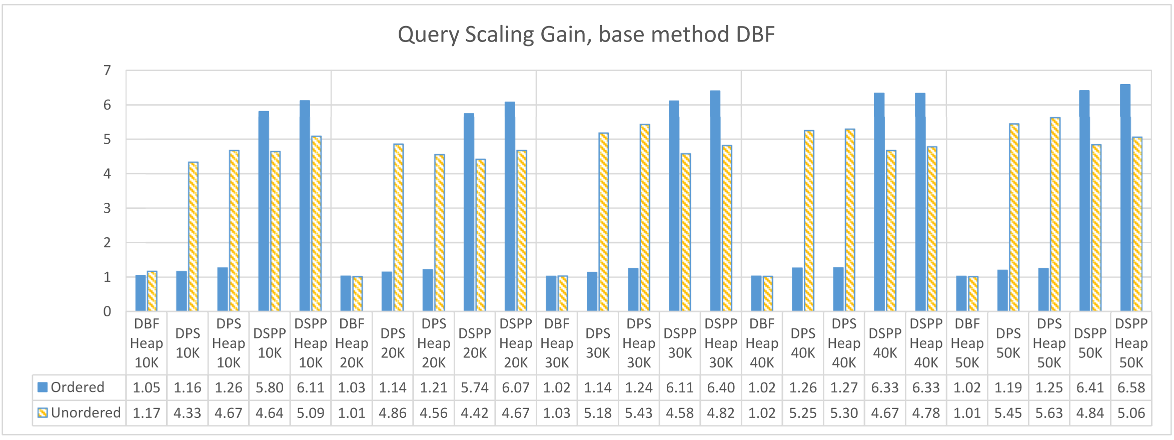


	DBF 10K	DBF Heap 10K	DPS 10K	DPS Heap 10K	DSPP 10K	DSPP Heap 10K	DBF 20K	DBF Heap 20K	DPS 20K	DPS Heap 20K	DSPP 20K	DSPP Heap 20K	DBF 30K	DBF Heap 30K	DPS 30K	DPS Heap 30K	DSPP 30K	DSPP Heap 30K	DBF 40K	DBF Heap 40K	DPS 40K	DPS Heap 40K	DSPP 40K	DSPP Heap 40K	DBF 50K	DBF Heap 50K	DPS 50K	DPS Heap 50K	DSPP 50K	DSPP Heap 50K
■ Presorted	110	105	95	87	19	18	206	201	181	170	36	34	307	301	270	247	50	48	415	406	330	326	66	66	520	512	436	417	81	79
▨ Unsorted	117	100	27	25	25	23	201	198	41	44	45	43	299	290	58	55	65	62	397	391	76	75	85	83	501	496	92	89	104	99

Figure 6.8: Query scaling experiment (Y-axis in seconds).

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 seconds, for DBF Heap is 117 seconds, for DPS 24 seconds, for DPS Heap 25 seconds, for DSPP 25 seconds 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 seconds for DBF, 501 seconds for DBF Heap, 90 seconds for DPS, 94 seconds for DPS Heap, 104 seconds for DSPP and 106 seconds 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. 6.9) than the Brute-force one, in both dataset experiments



	DBF Heap 10K	DPS 10K	DPS Heap 10K	DSPP 10K	DSPP Heap 10K	DBF Heap 20K	DPS 20K	DPS Heap 20K	DSPP 20K	DSPP Heap 20K	DBF Heap 30K	DPS 30K	DPS Heap 30K	DSPP 30K	DSPP Heap 30K	DBF Heap 40K	DPS 40K	DPS Heap 40K	DSPP 40K	DSPP Heap 40K	DBF Heap 50K	DPS 50K	DPS Heap 50K	DSPP 50K	DSPP Heap 50K
■ Ordered	1.05	1.16	1.26	5.80	6.11	1.03	1.14	1.21	5.74	6.07	1.02	1.14	1.24	6.11	6.40	1.02	1.26	1.27	6.33	6.33	1.02	1.19	1.25	6.41	6.58
▨ Unordered	1.17	4.33	4.67	4.64	5.09	1.01	4.86	4.56	4.42	4.67	1.03	5.18	5.43	4.58	4.82	1.02	5.25	5.30	4.67	4.78	1.01	5.45	5.63	4.84	5.06

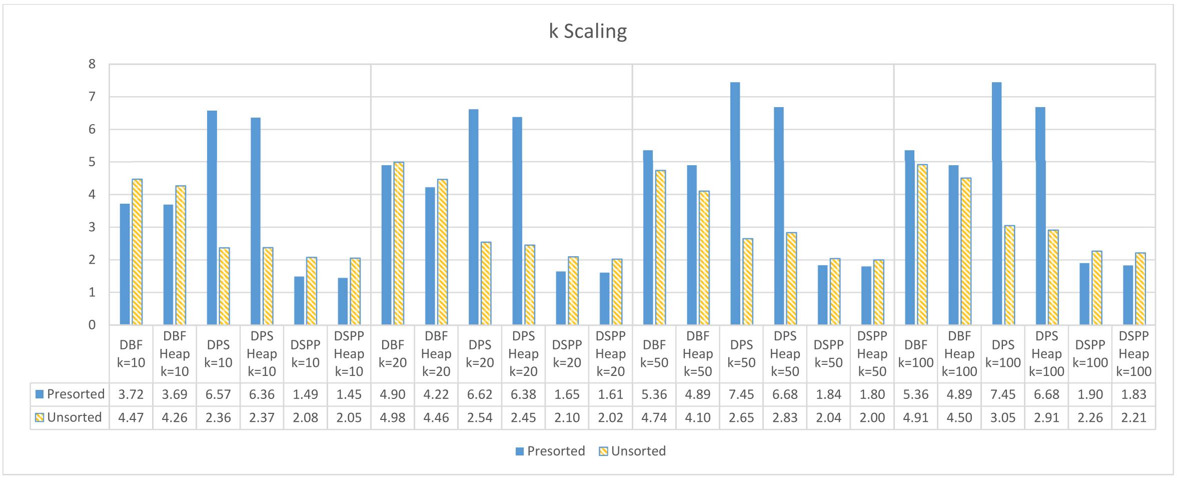
Figure 6.9: Query scaling experiment gain, base method DBF.

6.4.4 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. 6.10, 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 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 seconds for DBF, 3.69 second for DBF Heap, 6.57 for DPS, 6.36 seconds for DPS Heap, 1.49 seconds for DSPP and 1.45 for DSPP Heap. The slowest execution times were recorded for k value of 100, 5.36 seconds for DBF, 4.89 seconds for DBF Heap, 7.45 seconds for DPS, 6.68 for DPS Heap, 1.90 seconds for DSPP and 1.83 for DSPP Heap.

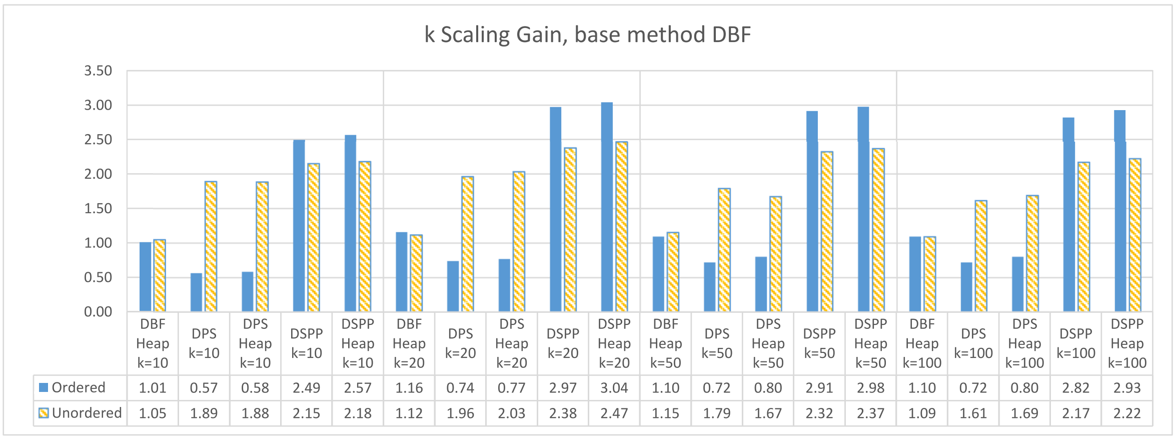
In the unsorted experiments, we observe once more that the execution of DPS is faster than the presorted ones. The Brute-force methods are slightly faster for the unsorted dataset than for the presorted one. The DSPP methods are quicker than the other methods. For k equal to 10 for the unsorted query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



	DBF k=10	DBF Heap k=10	DPS k=10	DPS Heap k=10	DSPP k=10	DSPP Heap k=10	DBF k=20	DBF Heap k=20	DPS k=20	DPS Heap k=20	DSPP k=20	DSPP Heap k=20	DBF k=50	DBF Heap k=50	DPS k=50	DPS Heap k=50	DSPP k=50	DSPP Heap k=50	DBF k=100	DBF Heap k=100	DPS k=100	DPS Heap k=100	DSPP k=100	DSPP Heap k=100
■ Presorted	3.72	3.69	6.57	6.36	1.49	1.45	4.90	4.22	6.62	6.38	1.65	1.61	5.36	4.89	7.45	6.68	1.84	1.80	5.36	4.89	7.45	6.68	1.90	1.83
▨ Unsorted	4.47	4.26	2.36	2.37	2.08	2.05	4.98	4.46	2.54	2.45	2.10	2.02	4.74	4.10	2.65	2.83	2.04	2.00	4.91	4.50	3.05	2.91	2.26	2.21

Figure 6.10: k scaling experiment (Y-axis in seconds).

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. 6.11), in the all dataset experiments. The Heap methods were slightly better than the KNN-DLB ones.



	DBF Heap k=10	DPS k=10	DPS Heap k=10	DSPP k=10	DSPP Heap k=10	DBF Heap k=20	DPS k=20	DPS Heap k=20	DSPP k=20	DSPP Heap k=20	DBF Heap k=50	DPS k=50	DPS Heap k=50	DSPP k=50	DSPP Heap k=50	DBF Heap k=100	DPS k=100	DPS Heap k=100	DSPP k=100	DSPP Heap k=100
■ Ordered	1.01	0.57	0.58	2.49	2.57	1.16	0.74	0.77	2.97	3.04	1.10	0.72	0.80	2.91	2.98	1.10	0.72	0.80	2.82	2.93
▨ Unordered	1.05	1.89	1.88	2.15	2.18	1.12	1.96	2.03	2.38	2.47	1.15	1.79	1.67	2.32	2.37	1.09	1.61	1.69	2.17	2.22

Figure 6.11: k scaling experiment gain, base method DBF.

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

6.4.5 Interpretation of Results

As noted in [90], 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 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. 5.6 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 k-th NN found so far (Fig. 5.6 bottom).

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.

6.5 Conclusions

In this chapter, 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. The algorithms documented in this chapter were published in [113].

Chapter 7

$k-$NN Query Processing with GPU, SSD and Full Dataset Partitioning

As we have already stated in our previous chapters, GPU devices can be utilized for efficient parallel computation of demanding spatial queries, like the k Nearest-Neighbor (k-NN) query, which is widely used for spatial distance classification in many problems areas.

When dealing with big data, the GPU device's memory and/or host main memory may not be able to accommodate an entire, reference and query dataset. As a result, storing this dataset in a fast secondary device, like a Solid State Disk (SSD) is, in many practical cases, a feasible solution.

In this chapter,

- We propose and implement (extending the DSPP algorithm [90]) the first GPU-based algorithms for processing the k-NN query not only on big reference, but also on big query data stored on SSDs.
- We exploit concurrent CUDA kernel execution to enable multiple concurrent CUDA stream k-NN calculations, resulting to better utilization of GPU resources and data transfers/computation overlap.
- We utilize either an array-based, or a max-Heap based buffer for storing the distances of the current k nearest neighbors, which are combined with our new methods, deriving two algorithmic variations.
- Based on 3d synthetic and real big data, we present an extensive experimental comparison of these algorithmic variations, varying query dataset size, reference dataset size

and k. These experiments highlight that the new methods, combined with either an array or a max-Heap buffer are performance winners, especially for very large reference and query datasets and big k values.

The rest of this chapter is organized as follows. In Section 7.1, we review related material and present the motivation for our work. Next, in Section 7.2, we introduce the new algorithm that we developed for the k-NN GPU-based processing on disk-resident[1] data and in Section 7.3, we present the experimental study that we performed for analyzing the performance of all our algorithms and for determining the performance winner among 10 (6 existing and 4 new) algorithmic variations tested on synthetic and real big reference and query data. Finally, in Section 7.4, we present the conclusions arising from our work and discuss our future plans.

7.1 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 *Brute-Force* methods or by using *Spatial Subdivision* techniques. We have reviewed most of these works in Section 6.1 where such mechanisms have been applied to improve the performance on GPUs. Furthermore, *concurrent kernel execution* is an effective method to improve hardware utilization, and it can be used on GPUs to improve resource utilization and system performance, especially when kernels are running together.

7.1.1 Concurrent Kernel Execution

Recent GPUs support *concurrent kernel execution*, that enables different kernels to run simultaneously on the same GPU, sharing the GPU hardware resources. Concurrent kernel execution can improve GPU hardware utilization and system performance. This feature of the current GPU programming models can be used in different scenarios to allow better utilization of GPU resources.

The impact of concurrent kernel execution on performance improvement by funneling all kernels of a multi-threaded host process into a single GPU context was firstly examined in [118]. In the same scenario, a kernel reordering technique is proposed in [119] to improve

[1]We used an SSD and in the rest of the text "SSD" instead of "disk" is used.

GPU performance by taking advantage of concurrent kernel execution focusing on the order in which GPU kernels are invoked on the host side.

In [120], the authors experimentally validate the benefits of using concurrent kernel execution to improve GPU energy-efficiency for computational kernels. For this purpose, they design power-performance models to carefully select the appropriate kernel combinations to be executed concurrently, the relative contributions of the kernels to the thread mix, along with the frequency choices for the cores and the memory to achieve high performance per watt metric.

[121] illustrates that compute-intensive kernels may be starved because other memory-intensive kernels block the memory pipeline on Simultaneous Multitasking (SMK) GPUs. To solve this problem, a dynamic memory instruction limiting method to mitigate the memory pipeline contention and accelerate concurrent kernel execution is proposed. The experimental results show that the proposed approach improves weighted speedup by 27.2% on average over SMK, with minor hardware cost.

In [122], the authors highlight that memory interference can significantly affect the throughput and fairness of concurrent kernel execution. They make a case that even the optimal Cooperative Thread Array (CTA) combination does not eliminate the negative memory interference impact. To address this problem effectively, a coordinated approach for CTA combination and bandwidth partitioning for GPU concurrent kernel execution is proposed. This approach effectively reduces the memory latency for the latency-sensitive kernels. In the meanwhile, the bandwidth utilization is also improved for the bandwidth-intensive kernels.

The performance of compute-intensive kernels is significantly reduced when memory-intensive kernels block memory pipeline and occupy most L1 data cache (L1D) resources, and it is highlighted in [123]. They propose a fair and cache blocking aware warp scheduling (FCBWS) approach for concurrent kernel execution on GPU to ameliorate the contention on data cache and improve system performance. FCBWS adopts kernel aware warp scheduling to provide equal chance of issuing instructions to each kernel. Moreover, for a ready memory instruction to be issued, if it is predicated that this instruction will block the data cache, FCBWS will select and issue another ready instruction of the same kernel; otherwise, this memory instruction will be issued to the memory pipeline. The experiment results indicate that FCBWS has important advantages over spatial multitasking and previous SMK works.

Another context to use concurrent kernel execution is to implement scheduling policies on

GPUs. For example, a software scheduler for GPU applications, called *FlexSched*, has been recently presented in [124], that takes advantage of concurrent kernel execution to implement scheduling policies aimed at maximizing application execution performance, or meeting Quality of Service (QoS) application requirements such as maximum turnaround time. An important feature of FlexSched is the use of a productive on-line profiling, employing a heuristic that compares different co-execution configurations to find a suitable CTA allocation scheme that fulfills the scheduling requirements: throughput or QoS. In a real scheduling scenario, where new applications are launched as soon as GPU resources become available, FlexSched reduces the average overall execution time by a factor of 1.25x with respect to the time obtained when proprietary hardware (HyperQ) is employed.

For interested readers, [125] is a recent survey on GPU multitasking methods, where concurrent kernel execution is studied as a feature of GPUs to support multitasking.

7.1.2 Motivation

Based on our literature research we concluded that there are no methods targeting big reference and query datasets (larger than the available device memory), neither are there methods that explore concurrent kernel execution, for the calculation of k-NN. Furthermore, our previous k-NN method implementations, address only big reference data; they can process only query datasets big enough to fit in device memory.

Based on these facts, there is no work so far that uses the advantages of concurrent kernel execution, to efficiently design and implement k-NN algorithms on GPUs. We will try to investigate and test how the invocation of this feature would increase k-NN calculation performance.

Moreover, we will try to extend our methods implementations, to aim also big query datasets. We will also leverage the trade-offs that arise from new algorithmic overheads and evaluate the effectiveness of our new methods.

7.2 $k-$NN Disk Algorithms

A common practice to handle big data is data partitioning. In order to describe our new algorithms, we first 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 reference points is not divided exactly by 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[2] and loads it into the host memory. The host copies the in-memory partition data into the GPU device memory.[3]

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 in the kernel execution geometry. The thread dispatching consists of 4 main steps:

1. The kernel is invoked with a grid of 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 kernel invocation is populated with the results of all the query points.

In the next sections we will describe our existing and new methods. These methods are based on three main algorithms, "Disk Brute-force", "Disk Plane-sweep" and "Symmetric Progression Partitioning". In all of them we have implemented two k-NN buffer variations resulting in a total of 6 existing and 4 new methods (algorithmic variations).

7.2.1 Disk Brute-force Algorithm

The Disk Brute-force algorithm (denoted by DBF) [90] 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 $R = \{r_1, r_2, r_3, \cdots r_m\}$ in 3d space and a dataset Q of n query points $Q = \{q_1, q_2, q_3, \cdots q_n\}$ also in 3d space. The host reads the query dataset and transfers it in the

[2]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.

[3]The hardware we used does not support GPUDirect storage.

device memory. The reference dataset is transfered to the device memory and is partitioned into equally sized bins. 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 4.2.5 and 4.2.6.

7.2.2 Disk Plane-sweep Algorithm

An important improvement for join queries is the use of the Plane-sweep technique, which is presented in Section 5.2.2.

Like DBF, DPS accepts as inputs a reference dataset R consisting of m reference points $R = \{r_1, r_2, r_3..r_m\}$ in 3d space and a dataset Q of n query points $Q = \{q_1, q_2, q_3..q_n\}$ 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, 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.

7.2.3 Disk Symmetric Progression Partitioning

A more efficient method than the previous ones is the "Disk Symmetric Progression Partitioning" [113], 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 7.2. The second level partitioning takes 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 DPS, DSPP accepts as inputs a reference dataset R consisting of m reference points $R = \{r_1, r_2, r_3, \cdots r_m\}$ in 3d space and a dataset Q of n query points $Q = \{q_1, q_2, q_3, \cdots q_n\}$ also in 3d space. We have presented the DSPP algorithm in Section 6.3.3. We will use and compare two alternative k-NN buffer implementations, presented in Sections 4.2.5 and 4.2.6, thus resulting to two DSPP method variations.

7.2.4 Improved Disk Symmetric Progression Partitioning

One disadvantage of the DSPP method is that it can only process query datasets that fit in device memory. Surely modern GPUs are equipped with abundant memory, however modern big data query datasets can easily surpass GPU memory capacity. To fill this gap we designed and implemented an improved DSPP method, denoted by DSPP+. In our first new method, we incorporated an extra step of query dataset partitioning, just before the device k-NN calculation (Fig. 7.1). This means that the query dataset will be fully read, partition by partition, every time we need to process the next reference partition (Alg. 15). If we partition the reference dataset in N partitions, then the query dataset will be read N times. Taking this under consideration, we expect an execution performance decrease, unless we manage to overlap those transfers with useful computation (which we address in Section 7.2.5) . We must outline that this approach has some extra advantages, apart from processing big query datasets. One advantage is that when we initiate kernel processes the scheduled CUDA blocks query smaller volumes or points resulting in better L2 cache locality. Furthermore, the device data processed are close to each other and the method benefits from coalesced memory transaction, when consecutive threads access consecutive memory addresses.

Like our previous methods, DSPP+ accepts as inputs a reference dataset R consisting of m reference points $R = \{r_1, r_2, r_3, \cdots r_m\}$ in 3d space and a dataset Q of n query points $Q = \{q_1, q_2, q_3, \cdots q_n\}$ also in 3d space. We partition the reference dataset PR consisting of pm reference partitions $PR = \{pr_1, pr_2, pr_3, \cdots pr_{pm}\}$. In analogy, we partition the query dataset QR consisting of qn query partitions $QR = \{qr_1, qr_2, qr_3, \cdots qr_{qn}\}$. For each $PR[i]$ partition, we traverse sequentially all $QR[j]$ partitions and merge the resulting k-NNs to our k-NN buffer (Fig. 7.1).

Like our previous methods, we will use and compare two alternative k-NN buffer implementations, presented in Sections 4.2.5 and 4.2.6, thus resulting to two DSPP+ method variations.

7.2.5 Improved Disk Symmetric Progression Partitioning with pinned memory

The second new method we are presenting, exploits a significant CUDA feature. CUDA streams, which aim to hide the latency of memory copy and kernel launch from different

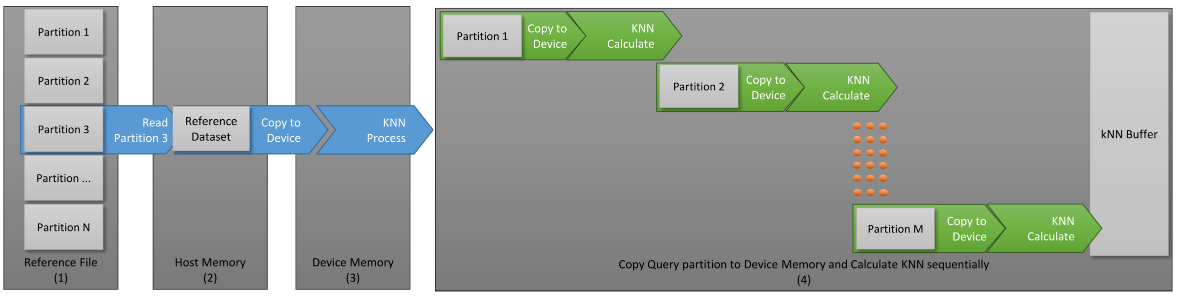


Figure 7.1: DSPP+ Partitioning and execution path.

Algorithm 15 DSPP+ Host algorithm

Input: NN cardinality=K, Reference filename=RF, Query filename=QF, Reference Partition size=S, Reference sub-Partition cardinality=CB, Query Partition size=QS

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:    while not end-of-file QF do
18:        HostQueryPartition ← readPartition(QF,QS);
19:        DeviceQueryPartition ← HostQueryPartition;
20:        runKNN<<<(QS-1)/256 +1, 256>>>(DeviceReferencePartition,
              DeviceQueryPartition, DeviceQueryVector,DeviceKNNBufferVector,K,
              DeviceReferencePartitionIndex);      // 256 cores assumed
21: HostKNNBufferVector ← DeviceKNNBufferVector;
```

independent operations [9], are widely used in computational tasks to increase performance [10].

When CUDA Streams are used, together with pinned memory supporting asynchronous data transfers, we can overlap data transfers with kernel execution, thus effectively hiding data transfer latency. This improves GPU utilization and reduces execution time.

This new method is the Improved Disk Symmetric Progression Partitioning with pinned

Algorithm 16 DSPP+ Device Kernel algorithm (runKNN)

```
Input: NN cardinality=K, Partition Reference array=R, Query array=Q,
       Reference 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 then break;
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;
```

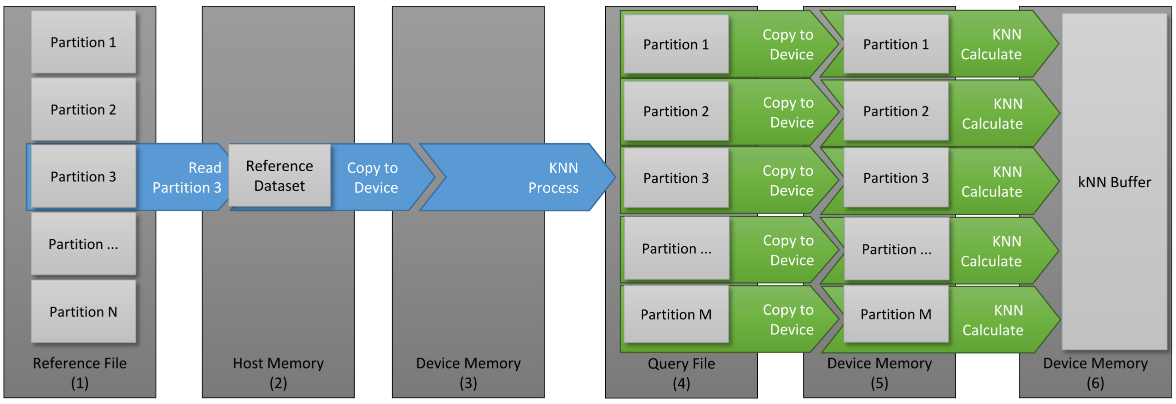


Figure 7.2: DSPP+P Partitioning and execution path.

memory (denoted by DSPP+P).

The DSPP+ algorithm partitions the query dataset and calculates the k-NN for each one. In this second new algorithm, we process every query partition in a new stream. The CUDA kernel executes concurrently and the k-NNs are written to the output buffer (Fig. 7.2).

Like DSPP+, DSPP+P accepts as inputs a reference dataset R consisting of m reference points $R = \{r_1, r_2, r_3, \cdots r_m\}$ in 3d space and a dataset Q of n query points $Q = \{q_1, q_2, q_3, \cdots q_n\}$ also in 3d space. We partition the reference dataset PR consisting of pm reference partitions $PR = \{pr_1, pr_2, pr_3, \cdots pr_{pm}\}$. Analogously we partition the query dataset QR consisting of qn query partitions $QR = \{qr_1, qr_2, qr_3, \cdots qr_{qn}\}$. For each $PR[i]$ partition, we traverse in a parallel and fully asynchronous way, all $QR[j]$ partitions and merge the resulting k-NNs to our k-NN buffer (Fig. 7.2).

Like our previous methods, we will use and compare two alternative k-NN buffer implementations, presented in Sections 4.2.5 and 4.2.6, thus resulting to two DSPP+P method variations.

Algorithm 17 DSPP+P Host algorithm

```
Input: NN cardinality=K, Reference filename=RF, Query filename=QF, Reference Partition size=S, Reference sub-Partition cardinality=CB, Query Partition size=QS, Streaming multiprocessors=SMs
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:     QueryPartitionNumber ← 0;
18:     while not end-of-file QF do
19:         HostQueryPartition ← readPartition(QF,QS);
20:         DeviceQueryPartition ← HostQueryPartition;
21:         runKNN<<<(QS-1)/64 +1, 64, QueryPartitionNumber % SMs >>>
              (DeviceReferencePartition, DeviceQueryPartition,
              DeviceQueryVector,DeviceKNNBufferVector,K,DeviceReferencePartitionIndex);
                // Concurrent kernel execution, 64 cores assumed
22:         QueryPartitionNumber ← QueryPartitionNumber + 1;
23:         if QueryPartitionNumber % SMs==0 then cudaDeviceSynchronize();
24: HostKNNBufferVector ← DeviceKNNBufferVector;
```

7.3 Experimental Study

We run a large set of experiments to quantify the performance of our proposed algorithms. All experiments query a variety of dataset volumes of synthetic and real data. We are using double precision accuracy for the points representation in 3D space (Alg. 11) to be able to discriminate among small distance differences.

Algorithm 18 DSPP+P Device Kernel algorithm (runKNN)

```
Input: NN cardinality=K, Partition Reference array=R, Query array=Q,
       Reference 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 then break;
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;
```

All experiments were performed on a Dell G5 15 laptop, running Ubuntu 20.04, equipped with a six core (12-thread) Intel I7 CPU, 16GB of main memory, a 1TB SSD disk used and a NVIDIA Geforce 2070 (Mobile Max-Q) GPU with 8GB of device memory (as a representative setup for everyday computing). CUDA version 11.2 was used.

We run experiments to compare the performance of k-NN queries regarding execution time, as well as memory utilization. We tested a total of ten algorithms.

1. DBF, Disk Brute-force using KNN-DLB buffer
2. DBF Heap, Disk Brute-force using max-Heap buffer
3. DPS, Disk Plane-sweep using KNN-DLB buffer
4. DPS Heap, Disk Plane-sweep using max-Heap buffer
5. DSPP, Disk Symmetric Progression Partitioning using KNN-DLB buffer
6. DSPP Heap, Disk Symmetric Progression Partitioning using max-Heap buffer
7. DSPP+, Improved Disk Symmetric Progression Partitioning using KNN-DLB buffer
8. DSPP+ Heap, Improved Disk Symmetric Progression Partitioning using max-Heap buffer
9. DSPP+P, Improved Disk Symmetric Progression Partitioning with pinned memory using KNN-DLB buffer

10. DSPP+P Heap, Improved Disk Symmetric Progression Partitioning with pinned memory using max-Heap buffer

To the best of our knowledge, these are the first methods to address the k-NN query on SSD-resident data, that can process big reference and query datasets.

The experimental study is divided in two main subsection. The first one is based on synthetic data and the second one on real data.

7.3.1 Synthetic data experiments

In this section we will evaluate the performance of our methods, based only on synthetic data. All the datasets were created using the SpiderWeb [99] generator. This generator allows users to choose from a wide range of spatial data distributions and configure the size of the dataset and its distribution parameters. This generator has been successfully used in research work to evaluate index construction, query processing, spatial partitioning, and cost model verification [100].

Table 7.1 lists all the generated datasets. For the reference dataset, we created five datasets using the "Bit" distribution (Fig. 7.3 right), with file sizes ranging from 32MB to 160MB. The reference points dataset size ranges from 1M points to 5M points. For the query points dataset we created five "Uniform" datasets (Fig. 7.3, left) ranging from 100K to 500K points.

Distribution	Size	Seed	File Size	Dataset usage
Bit	1M	1	32MB	Reference
Bit	2M	2	64MB	Reference
Bit	3M	3	96MB	Reference
Bit	4M	4	128MB	Reference
Bit	5M	4	160MB	Reference
Uniform	100K	6	3.2MB	Query
Uniform	200K	7	6.4MB	Query
Uniform	300K	8	9.6MB	Query
Uniform	400K	9	12.8MB	Query
Uniform	500K	10	16MB	Query

Table 7.1: SpiderWeb Dataset generator parameters.

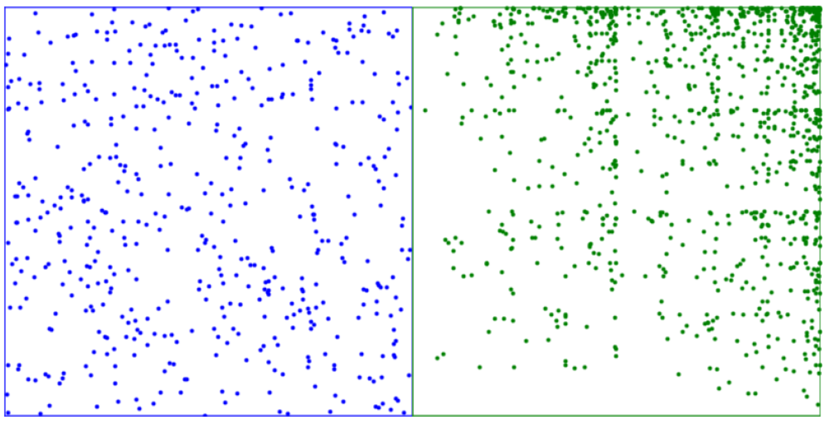


Figure 7.3: Experiment distributions, Left=Uniform, Right=Bit .

Three different sets of experiments on synthetic data were conducted. In the first one, we scaled the reference dataset size, in the second one we scaled the query dataset size and in the third one we scaled the number of the nearest neighbors, k. We also evaluated the performance of the two alternative list buffers to clarify the pros and cons of using KNN-DLB and max-Heap buffer.

7.3.1.1 Reference dataset scaling

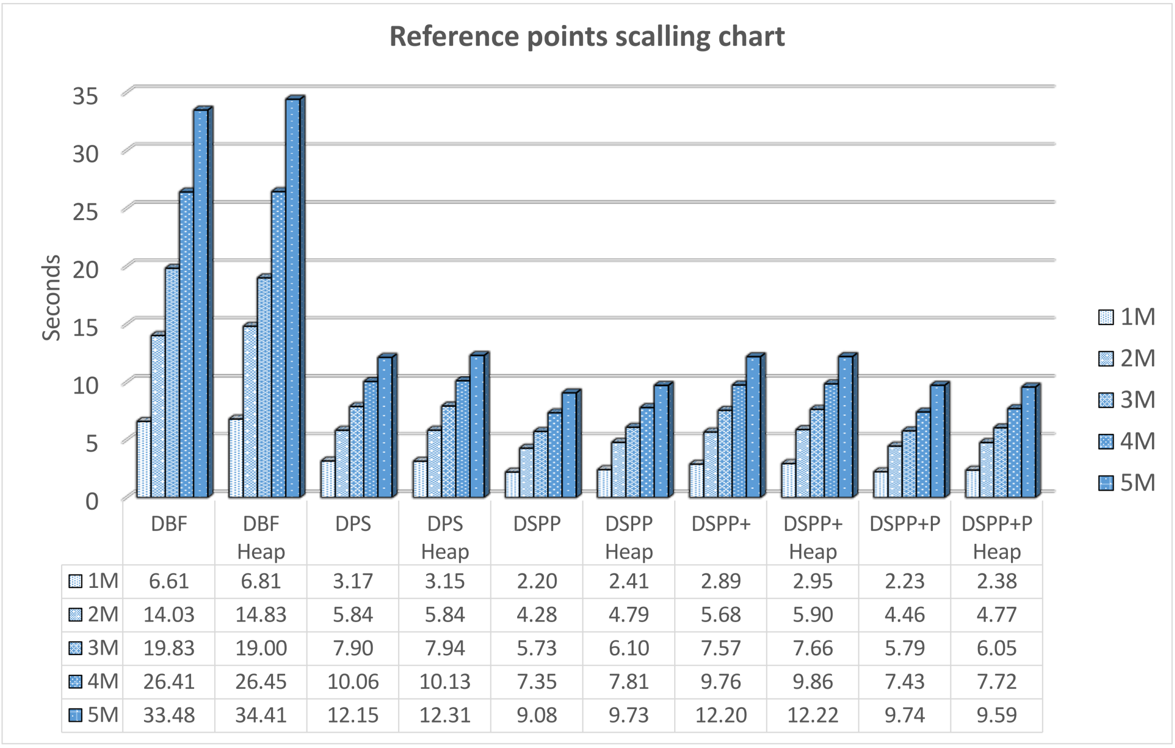
In our first series of tests, we used the “Bit” distribution synthetic datasets for the reference points. The size of the reference point dataset ranged from 1M points to 5M points. Furthermore, we used a fixed query dataset of 100K points, with “Uniform” distribution and a k value of 20, in order to focus only on the reference dataset scaling.

In Fig. 7.4, we can see the experiment results chart. In these results, we notice that the execution time of the Brute-force methods is larger than the rest of the methods. Apart from the Brute-force methods, all other methods have quite similar executions times, for each reference dataset size. For example, for the 1M dataset the execution times range from 6.81 seconds for DBF Heap (slowest) to 2.20 seconds for the DSPP (fastest). The execution times increase proportionally to the reference dataset size. As expected, we get the slowest execution times for the 5M dataset, ranging from 34.41 seconds for DBF Heap to 9.08 for the DSPP Heap method.

In order to compare the performance of our methods in Table 7.2, we present the execution speedup gain, using the slowest method (DFB) as the baseline. Every number in this table represents the method gain relative to the base method DBF (how many times faster than DBF). As expected, the DBF Heap method gain is close to 1, meaning that the DBF and

DBF Heap are performing equally in all reference datasets. The execution speedup gain of the other methods ranges from 2.08 (times faster) in DPS for the 1M reference dataset, to 3.69 (times faster) in DSPP for the 5M reference dataset. We should notice that the method DSPP+P, is the second fastest method, performing slightly worse than DSPP, and achieved a speedup of 3.56 for the 4M reference dataset.

The reference dataset scaling experiments reveal that the Brute-force methods did not perform well. This behaviour is expected because of the naive Brute-force algorithm of these methods. The DSPP method is faster than the DPS one, confirming our results from our previous publications [8, 90]. The interesting part of this experiment is that our new methods DSPP+ and DSPP+P performed about equally to DSPP, despite their query partitioning algorithm adding an overall overhead by repeatedly reading the query dataset. Especially DSPP+P performance is equivalent to DSPP. As we experimentally validate, this overhead was leveraged by the concurrent kernel execution invocation of DSPP+P. Furthermore, we can observe that the two k-NN distance list buffers, KNN-DLB and max-Heap, perform equally in all reference datasets. The value of k=20 is explicitly selected to be small enough so that the buffer usage does not affect the experimental results. The difference between the two buffers will be quantified in the k scaling experiment later on.



	DBF	DBF Heap	DPS	DPS Heap	DSPP	DSPP Heap	DSPP+	DSPP+ Heap	DSPP+P	DSPP+P Heap
1M	6.61	6.81	3.17	3.15	2.20	2.41	2.89	2.95	2.23	2.38
2M	14.03	14.83	5.84	5.84	4.28	4.79	5.68	5.90	4.46	4.77
3M	19.83	19.00	7.90	7.94	5.73	6.10	7.57	7.66	5.79	6.05
4M	26.41	26.45	10.06	10.13	7.35	7.81	9.76	9.86	7.43	7.72
5M	33.48	34.41	12.15	12.31	9.08	9.73	12.20	12.22	9.74	9.59

Figure 7.4: Reference scaling experiment (Y-axis in sec.).

Method	**1M**	**2M**	**3M**	**4M**	**5M**
DBF Heap	0.97	0.95	1.04	1.00	0.97
DPS	2.08	2.40	2.51	2.62	2.76
DPS Heap	2.10	2.40	2.50	2.61	2.72
DSPP	**3.00**	**3.28**	**3.46**	**3.59**	**3.69**
DSPP Heap	2.74	2.93	3.25	3.38	3.44
DSPP+	2.29	2.47	2.62	2.71	2.74
DSPP+ Heap	2.24	2.38	2.59	2.68	2.74
DSPP+P	2.96	3.15	3.43	3.56	3.44
DSPP+P Heap	2.78	2.94	3.28	3.42	3.49

Table 7.2: Reference scale gain, base method DBF.

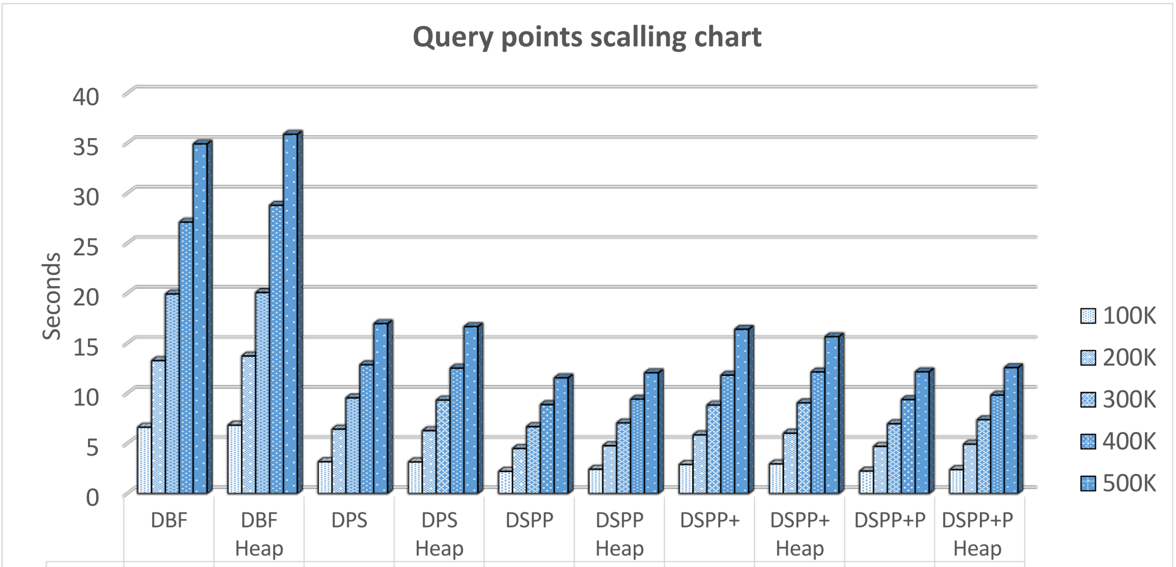
7.3.1.2 Query dataset scaling

In our second set of experiments, we also used the “Uniform” distribution synthetic datasets for the query points. The size of the query point dataset ranged from 100K points to 500K points. Furthermore, we used a fixed reference dataset of 1M points, with “Bit” distribution and a k value of 20, in order to focus only on the query dataset scaling.

In Fig. 7.5, we can see the experiment results chart. In these results, we see similar results as in our previous experiment. The execution time of the Brute-force methods is higher than the rest of our methods. All the other methods have comparable executions times, for each reference dataset size. For example, for the 200K query dataset the execution times range from 13.72 seconds for DBF Heap (slowest) to 4.28 seconds for DSPP (fastest). The execution times increase proportionally to the query dataset size. We get the highest execution times for the 500K dataset, ranging from 35.87 seconds for DBF Heap to 11.54 for the DSPP method.

To compare the performance of our methods, in Table 7.3 we present the speedup, using the slowest method (DFB) as the baseline. Similarly to the previous experiment, the two Brute-force methods are again performing identically for all query datasets. The speedup of the other methods ranges from 2.07 (times faster) in DPS for the 200K dataset, to 3.06 (times faster) in DSPP for the 400K query dataset. As was also the case for our reference scale experiment, we notice that the method DSPP+P, is the second fastest method, performing slightly worse than DSPP, and achieved a speedup gain of 2.96 for the 100K query dataset.

The query dataset scaling experiment revealed once again that the Brute-force methods did not perform well for the same reasons as in our first experiment. The DSPP method is faster than the DPS one, confirming our results from our previous publications. Our new methods DSPP+ and DSPP+P performed about equally to DSPP, even if their query partitioning algorithm is adding an overall overhead by repeatedly reading the query dataset. Especially DSPP+P performance is equivalent to DSPP. We confirm again in this experiment that this overhead was leveraged by the concurrent kernel execution invocation of DSPP+P.



	DBF	DBF Heap	DPS	DPS Heap	DSPP	DSPP Heap	DSPP+	DSPP+ Heap	DSPP+P	DSPP+P Heap
▨ 100K	6.61	6.81	3.17	3.15	2.20	2.41	2.89	2.95	2.23	2.38
▨ 200K	13.26	13.72	6.42	6.26	4.48	4.76	5.84	6.01	4.69	4.92
▨ 300K	19.92	20.07	9.53	9.31	6.66	7.04	8.82	9.04	6.95	7.35
▨ 400K	27.10	28.78	12.84	12.50	8.86	9.41	11.81	12.12	9.36	9.81
▨ 500K	34.92	35.87	16.95	16.65	11.54	12.04	16.39	15.64	12.14	12.55

Figure 7.5: Query scaling experiment (Y-axis in sec.).

7.3.1.3 k scaling

The k scaling is our third experiment. In these tests we used k values of 20, 40, 60, 80 and 100. For the reference points we used the 1M "Bit" distribution synthetic dataset and for the query dataset 100K points, with "Uniform" distribution.

In Fig. 7.6, we can see the experiment results chart. The results in this experiment are quite different than the previous experiments. The execution time of the Brute-force methods is higher than the rest of our methods and we can see that the execution time of all KNN-DLB methods (DBF, DPS, DSPP, DSPP+ and DSPP+P), tend to increase in exponential way for larger k values. On the other hand, the execution time of all heap methods are increasing in a linear way. For k=100 the execution times range from 10.79 seconds for DBF (slowest) to

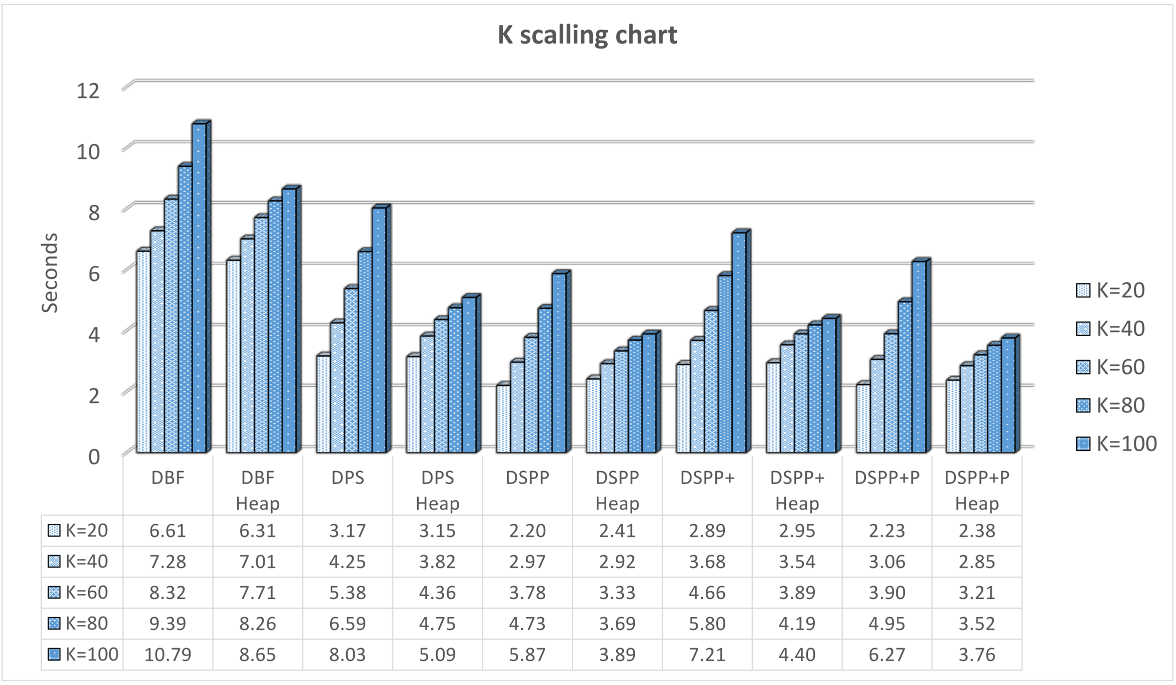
Method	**100K**	**200K**	**300K**	**400K**	**500K**
DBF Heap	0.97	0.97	0.99	0.94	0.97
DPS	2.08	2.07	2.09	2.11	2.06
DPS Heap	2.10	2.12	2.14	2.17	2.10
DSPP	**3.00**	**2.96**	**2.99**	**3.06**	**3.03**
DSPP Heap	2.74	2.78	2.83	2.88	2.90
DSPP+	2.29	2.27	2.26	2.30	2.13
DSPP+ Heap	2.24	2.21	2.20	2.24	2.23
DSPP+P	2.96	2.82	2.87	2.89	2.88
DSPP+P Heap	2.78	2.70	2.71	2.76	2.78

Table 7.3: Query scale gain, base method DBF.

3.76 seconds for DSPP+P (fastest).

In Table 7.4, we compare the execution performance of our methods, we present the execution speed gain, based on our slowest method DFB. In contrary to our previous experiments the two Brute-force methods are not performing identically; the DBF Heap method is faster for larger k values. Generally all heap methods perform clearly faster for larger k values. The execution gain of our methods for k=100, ranges from 1.25 (times faster) in DBF Heap, to 2.87 (times faster) in DSPP+P Heap. In this experiment the DSPP+P method is a clear winner. It is better than DSP in all k values except the smallest one, k=20.

In the k scaling experiment the heap methods stand out. For larger k values the max-Heap buffer is a much faster algorithm, because of its O(log(n)) complexity. Another interesting result is that DSPP+P Heap is overtaking even the DSPP Heap method. When we increase the k value, the k-NN calculation is even more computationally bound. The use of CUDA streams and the associated data transfers/kernel execution overlap, further accelerates this GPU costly operation, resulting in lower execution times, even if the DSPP+P methods repeatedly read the query dataset and transfer it to device memory. We confirm again in this experiment that the read overhead was successfully leveraged by the concurrent kernel execution invocation of DSPP+P.



	DBF	DBF Heap	DPS	DPS Heap	DSPP	DSPP Heap	DSPP+	DSPP+ Heap	DSPP+P	DSPP+P Heap
K=20	6.61	6.31	3.17	3.15	2.20	2.41	2.89	2.95	2.23	2.38
K=40	7.28	7.01	4.25	3.82	2.97	2.92	3.68	3.54	3.06	2.85
K=60	8.32	7.71	5.38	4.36	3.78	3.33	4.66	3.89	3.90	3.21
K=80	9.39	8.26	6.59	4.75	4.73	3.69	5.80	4.19	4.95	3.52
K=100	10.79	8.65	8.03	5.09	5.87	3.89	7.21	4.40	6.27	3.76

Figure 7.6: k scaling experiment (Y-axis in sec.).

Method	K=20	K=40	K=60	K=80	K=100
DBF Heap	1.05	1.04	1.08	1.14	1.25
DPS	2.08	1.71	1.55	1.43	1.34
DPS Heap	2.10	1.90	1.91	1.98	2.12
DSPP	**3.00**	2.45	2.20	1.98	1.84
DSPP Heap	2.74	2.50	2.49	2.55	2.77
DSPP+	2.29	1.98	1.78	1.62	1.50
DSPP+ Heap	2.24	2.06	2.14	2.24	2.45
DSPP+P	2.96	2.38	2.13	1.90	1.72
DSPP+P Heap	2.78	**2.55**	**2.59**	**2.67**	**2.87**

Table 7.4: K scale gain, base method DBF.

7.3.2 Real data experiments

In this section we will present three real data experiments, using the real datasets documented in Section 7.3. We used three big real datasets [1], which represent water resources of North America (Water Dataset) consisting of 5.8M line-segments and world parks or green areas (Parks Dataset) consisting of 11.5M polygons and world buildings (Buildings Dataset) consisting of 114.7M polygons (Table 7.5). To create sets of points, we used the centers of the line-segment MBRs from Water and the centroids of polygons from Park and Buildings. For all datasets the 3-dimensional data space is normalized to have unit length (values [0, 1] in each axis).

Description	Size	File Size	Dataset usage
Water	5M	186.8MB	Reference or Query
Parks	11.5M	368.1MB	Reference or Query
Buildings	114.5M	2.8GB	Reference or Query

Table 7.5: Real Datasets.

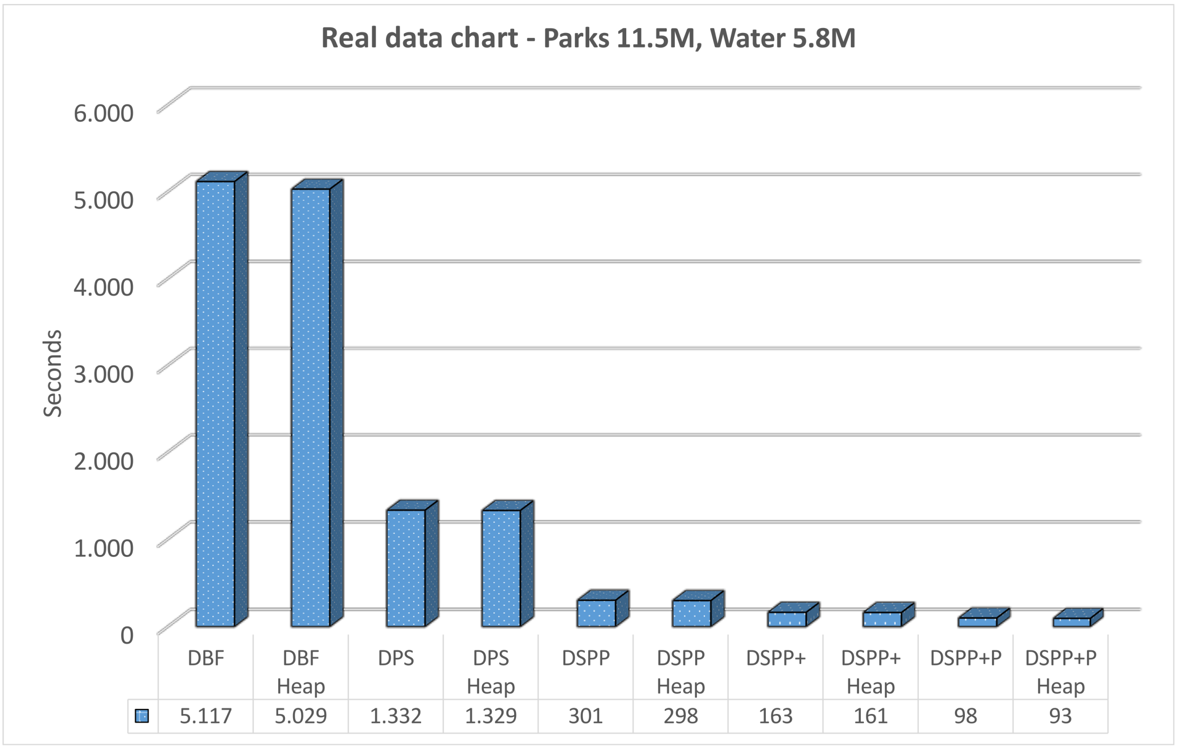
These experiments will evaluate our new methods performance when targeting real life data. The first of them queries only the smallest datasets, in order to compare our new methods performance with our previous ones. The second and third experiments query the larger real datasets, which our previous methods couldn’t target.

7.3.2.1 Real experiment 1 - Parks 11.5M, Water 5.8M

In Fig. 7.7, we can see the first real data experiment results chart. The results in this experiment are similar to the reference and query scaling experiments. The execution time of the Brute-force methods is much larger than the rest of our methods. The execution times range from 5,117 seconds for DBF (slowest method) to just 93 seconds for DSPP+P Heap (fastest method). When data volumes increase, the streaming kernel execution implementation in the DSPP+P methods, clearly outperforms all other methods, even the DSPP ones. In Table 7.6 we observe that DSPP+P was 57.23 times faster than DBF.

7.3.2.2 Real experiment 2 - Buildings 114.7M, Water 5.8M

The next real data experiment is presented in Fig. 7.8. In this experiment we evaluated only our new methods; our previous ones could not target a query dataset so large, because its footprint exceeds device memory capacity. The execution times range from 1,500 seconds for DSPP+ (slowest method) to 807 seconds for DSPP+P Heap (fastest method). Once again, when data volumes increase, the streaming kernel execution implementation in the DSPP+P methods, clearly outperforms all other methods.



	DBF	DBF Heap	DPS	DPS Heap	DSPP	DSPP Heap	DSPP+	DSPP+ Heap	DSPP+P	DSPP+P Heap
▨	5.117	5.029	1.332	1.329	301	298	163	161	98	93

Figure 7.7: Real data experiment (Y-axis in sec.).

Method	**Gain**
DBF Heap	0.94
DPS	3.84
DPS Heap	3.77
DSPP	17.00
DSPP Heap	16.63
DSPP+	31.42
DSPP+ Heap	31.85
DSPP+P	**57.23**
DSPP+P Heap	54.79

Table 7.6: Real data gain, base method DBF.

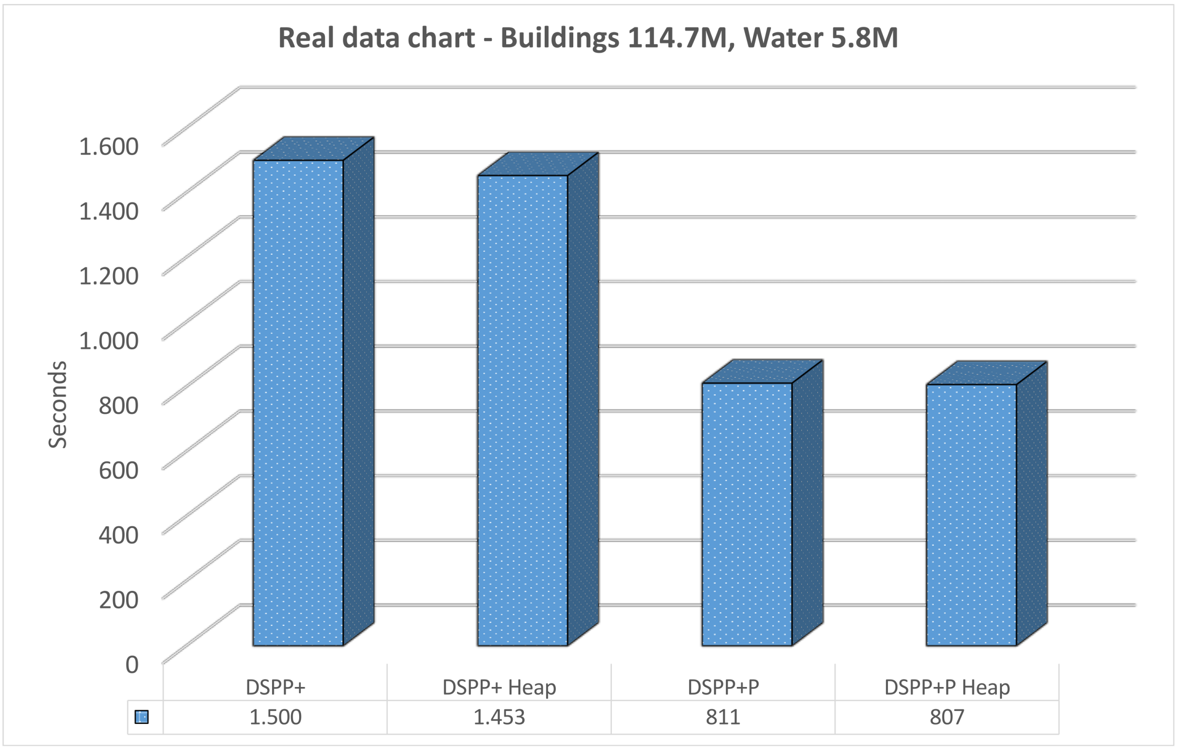


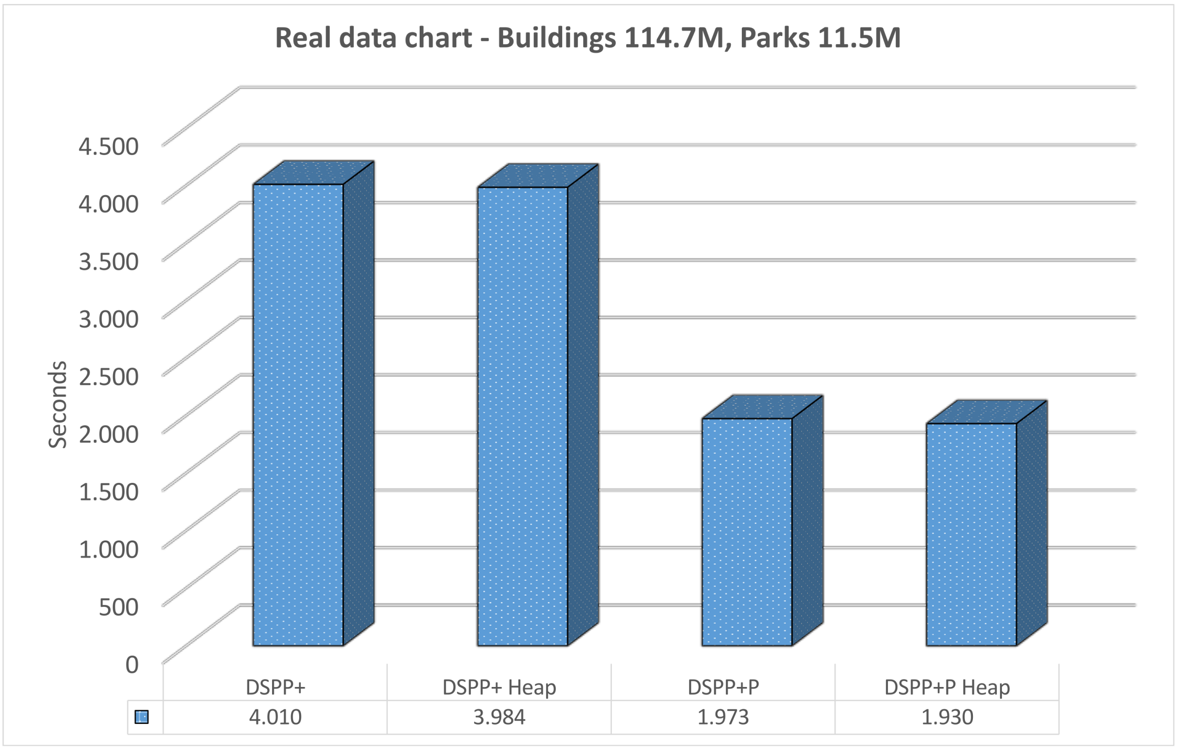
Figure 7.8: Real data experiment (Y-axis in sec.).

7.3.2.3 Real experiment 3 - Buildings 114.7M, Parks 11.5M

The last real data experiment is presented in Fig. 7.9. In this experiment we evaluated again only our new methods. The new methods successfully processed these large datasets, which were the largest in our experiments. The execution times range from 4,010 seconds for DSPP+ (slowest method) to 1,930 seconds for DSPP+P Heap (fastest method). Our results are once more validated, when data volumes increase, the streaming kernel execution implementation in the DSPP+P methods, is the performance winner.

7.4 Conclusions

In this chapter, we introduced the first GPU-based algorithms for parallel processing the k-NN query on reference and query big data stored on SSDs, utilizing the Symmetric Progression Partitioning technique. Our new algorithms exploit the manycore GPU architecture, the concurrent kernel execution feature of Nvidia GPUs, utilize the device memory efficiently, take advantage of the speed and storage capacity of SSDs and, thus, process efficiently big reference and query datasets. Through an extensive experimental evaluation on synthetic and



	DSPP+	DSPP+ Heap	DSPP+P	DSPP+P Heap
■	4.010	3.984	1.973	1.930

Figure 7.9: Real data experiment (Y-axis in sec.).

real datasets, we highlighted that the DSPP+P algorithm and especially its Heap variation, when using large k values and/or larger dataset volumes, is a clear performance winner. The algorithms documented in this chapter are submitted for possible publication.

Chapter 8

Conclusions and Future Directions

In this thesis we presented new and improved existing algorithms for processing several spatial queries in parallel in CPU and GPU environments. Some of these algorithms were the first ones in the literature to solve such queries. All the algorithms were presented in detail, graphically and using pseudocode and furthermore they were extensively tested against a variety of real world and synthetic datasets in 2 and 3 dimensions, using popular APIs such as OpenMP and CUDA. Existing implementations were also tested against our GPU methods. Several scaling experiments were tested for each GPU algorithm, including reference points scaling, query points scaling and k value scaling. Whenever these algorithms were tested against other popular ones from the literature, they always performed better.

8.1 Conclusions

The numerous experiments conducted for each algorithm using combinations of different APIs, datasets and tuning parameters have led to many interesting conclusions which will be presented separately for CPU and GPU, each query type.

8.1.1 Query Processing with CPU and SSD conclusions

In Chapter 3, we extended the algorithms presented in [46], for the first time in the literature, we present algorithms for common spatial batch queries on single datasets, using xBR$^+$-trees in SSDs that take advantage of multiple cores. Processing of spatial queries in SSDs has not received considerable attention in the literature, so far. Even more, the utilization of multiple cores, based on a combination of breadth-first and depth-first tree traversals, is a new approach that further accelerates processing. The algorithms proposed in [46] exploit the massive I/O advantages of SSDs and outperform the repetitive application of existing algorithms by exploiting the massive I/O advantages of SSDs, both regarding actual disk access and execution time, even if the I/O of existing algorithms are assisted by LRU buffering. The parallel extensions of these algorithms clearly outperform the ones of [46], although the three queries studied are I/O bound. The new algorithms can be applied to a parallel and distributed environment and deal with very big data.

8.1.2 Query Processing with GPU conclusions

8.1.2.1 k-NN Query Processing with GPU and RAM

In Chapter 4, we presented the first four in-memory algorithms for k-NN query processing in GPUs. These algorithms maximize the utilization of device memory, handling more reference points in the computation. Through an experimental evaluation on synthetic and real datasets, we concluded that T-DS only work faster than existing methods for small groups of query points, SPP outperforms existing methods for larger groups of query points, HSPP further enhances SPP performance for larger k values and all of them scale-up to much larger reference datasets. We validated that T-DS algorithm is faster than T-BS, because of the extra refinement step minimizing the sorting overhead. In terms of memory scaling, SPP and HSPP can compute up to 300M reference points, taking advantage of our KNN-DLB or Max-Heap optimizations. T-DS and T-BF could scale up to 200M and 100M respectively. SPP and HSPP can scale up to 300M points, about 300 times more than other existing algorithms and 100M reference points more than T-DS. HSPP is an overall performance winner, especially for larger k values.

8.1.2.2 k-NN Query Processing with GPU and SSD

In Chapter 5, we presented the first GPU-based algorithms for parallel processing the k-NN query on reference data stored on SSDs, utilizing the Brute-force and Plane-sweep techniques. These algorithms exploit the numerous GPU cores, utilize the device memory as much as possible and take advantage of the speed and storage capacity of SSDs, thus processing efficiently big reference datasets. Through an experimental evaluation on synthetic datasets, we highlighted that Plane-sweep on unsorted reference data (with either an array or a max-Heap buffer for organizing the current k-NNs) is a clear performance winner.

8.1.2.3 k-NN Query Processing with IoT Edge Devices and SSD

In Chapter 6, 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.

8.1.2.4 k-NN Query Processing with GPU, SSD and full dataset partitioning

In Chapter 7, we introduced the first GPU-based algorithms for parallel processing the k-NN query on reference and query big data stored on SSDs, utilizing the Symmetric Progression Partitioning technique. Our new algorithms exploit the manycore GPU architecture, the concurrent kernel execution feature of Nvidia GPUs, utilize the device memory efficiently, take advantage of the speed and storage capacity of SSDs and, thus, process efficiently big reference and query datasets. Through an extensive experimental evaluation on synthetic and real datasets, we highlighted that the DSPP+P algorithm and especially its Heap variation, when using large k values and/or larger dataset volumes, is a clear performance winner.

8.2 Future directions

Although, there are many future research directions that could build on the work presented in the thesis, the most mature ones are summarized in the following:

- Developing algorithms for parallel and distributed systems. Many of the spatio - temporal data analytics systems we presented in Section 2.1, already incorporate spatial indexing methods and distributes data to their nodes. In [67, 68], spatial query processing techniques have been added to SpatialHadoop. We could build on [67, 68] to embed into Spatiahadoop SSD-based xBR$^+$-trees that process spatial queries, taking advantage of multiple cores.

- Developing GPU algorithms for other demanding queries and/or queries also addressing the temporal dimension of data for the same computing architecture. Such algorithms can be the spatio-temporal polygon range query (STPRQ), which aims to find all records from a polygonal location in a time interval and the spatio-temporal k nearest neighbors query (STkNNQ), which directly searches the query point's k closest neighbors [126]. These algorithms can be executed in GPU devices, in a massively parallel way.

- Examining the utilization of indexes for speeding up GPU processing even further. Introduce Concurrent GPU Data Structures as presented in [127] and [128] and exploit such indexes. The usage of these indexes and the utilization of SSDs will accelerate the spatial query execution.

- Introducing even more advanced partitioning in DSPP algorithm, to further improve its performance. One technique that can be implemented is 3 dimensional grid based partitioning, such as Adaptive grid-based forest-like clustering algorithm [129] or context based partitioning for big data [130].

- Developing and implementing the architecture presented in Section 6.2. We will try to research the scaling potential of such architecture, using load balancing techniques and locality based acceleration. We will try to exploit the distributed edge computers, using communication based on MQTT technology or ad-hoc transmission techniques.

- implementation of join queries (like k-closest pairs [131]), based on techniques utilized in this dissemination.

Publications

[1] Polychronis Velentzas, Michael Vassilakopoulos, Antonio Corral, and Christos Antonopoulos. Gpu-based algorithms for processing the k nearest-neighbor query using partitioning and concurrent kernel execution. *Submitted for possible publication*.

[2] Polychronis Velentzas, Michael Vassilakopoulos, and Antonio Corral. Gpu-aided edge computing for processing the k nearest-neighbor query on ssd-resident data. *Internet of Things*, 15:100428, 2021.

[3] Polychronis Velentzas, Antonio Corral, and Michael Vassilakopoulos. Big spatial and spatio-temporal data analytics systems. *Transactions on Large Scale Data Knowledge Centered Systems*, 47:155–180, 2021.

[4] Polychronis Velentzas, Michael Vassilakopoulos, and Antonio Corral. Gpu-based algorithms for processing the k nearest-neighbor query on disk-resident data. In *Model and Data Engineering - 10th International Conference, MEDI 2021, Tallinn, Estonia, June 21-23, 2021, Proceedings*, volume 12732 of *Lecture Notes in Computer Science*, pages 264–278. Springer, 2021.

[5] George Roumelis, Polychronis Velentzas, Michael Vassilakopoulos, Antonio Corral, Athanasios Fevgas, and Yannis Manolopoulos. Parallel processing of spatial batch-queries using xbr^{+}-trees in solid-state drives. *Cluster Computing*, 23(3):1555–1575, 2020.

[6] Polychronis Velentzas, Michael Vassilakopoulos, and Antonio Corral. In-memory k nearest neighbor gpu-based query processing. In *Proceedings of the 6th International Conference on Geographical Information Systems Theory, Applications and Management, GISTAM 2020, Prague, Czech Republic, May 7-9, 2020*, pages 310–317. SCITEPRESS, 2020.

[7] Polychronis Velentzas, Michael Vassilakopoulos, and Antonio Corral. A partitioning gpu-based algorithm for processing the k nearest-neighbor query. In *MEDES '20: 12th International Conference on Management of Digital EcoSystems, Virtual Event, United Arab Emirates, 2-4 November, 2020*, pages 2–9. ACM, 2020 (2nd place in Best Papers Awards: `https://medes.sigappfr.org/20/best-paper-awards`).

[8] Polychronis Velentzas, Panagiotis Moutafis, and George Mavrommatis. An improved gpu-based algorithm for processing the k nearest neighbor query. In *PCI 2020: 24th Pan-Hellenic Conference on Informatics, Athens, Greece, 20-22 November, 2020*, pages 372–375. ACM, 2020.

References

[1] Ahmed Eldawy and Mohamed F. Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *ICDE*, pages 1352–1363. IEEE Computer Society, 2015.

[2] Francisco García-García, Antonio Corral, Luis Iribarne, Michael Vassilakopoulos, and Yannis Manolopoulos. Efficient large-scale distance-based join queries in spatialhadoop. *GeoInformatica*, 22(2):171–209, 2018.

[3] Jia Yu, Zongsi Zhang, and Mohamed Sarwat. Spatial data management in apache spark: the geospark perspective and beyond. *GeoInformatica*, 23(1):37–78, 2019.

[4] Louai Alarabi, Mohamed F. Mokbel, and Mashaal Musleh. St-hadoop: A mapreduce framework for spatio-temporal data. In *SSTD*, volume 10411 of *Lecture Notes in Computer Science*, pages 84–104. Springer, 2017.

[5] Louai Alarabi, Mohamed F. Mokbel, and Mashaal Musleh. St-hadoop: a mapreduce framework for spatio-temporal data. *GeoInformatica*, 22(4):785–813, 2018.

[6] Stefan Hagedorn, Philipp Götze, and Kai-Uwe Sattler. The STARK framework for spatio-temporal data analytics on spark. In *BTW*, volume P-265 of *LNI*, pages 123–142. GI, 2017.

[7] Stefan Hagedorn, Oliver Birli, and Kai-Uwe Sattler. Processing large raster and vector data in apache spark. In *BTW*, volume P-289 of *LNI*, pages 551–554. Gesellschaft für Informatik, Bonn, 2019.

[8] Polychronis Velentzas, Michael Vassilakopoulos, and Antonio Corral. A partitioning gpu-based algorithm for processing the k nearest-neighbor query. In *MEDES*, pages 2–9. ACM, 2020 (2nd place in Best Papers Awards: `https://medes.sigappfr.org/20/best-paper-awards`).

[9] Cuda 7 streams simplify concurrency. `https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/`. Last accessed: 2022-11-02.

[10] Husheng Zhou, Soroush Bateni, and Cong Liu. S^3dnn: Supervised streaming and scheduling for gpu-accelerated real-time DNN workloads. In *RTAS*, pages 190–201, 2018.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150. USENIX Association, 2004.

[12] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The performance of mapreduce: An in-depth study. *Proc. VLDB Endow.*, 3(1):472–483, 2010.

[13] Ahmed Eldawy and Mohamed F. Mokbel. The ecosystem of spatialhadoop. *ACM SIGSPATIAL Special*, 6(3):3–10, 2014.

[14] Apachehadoop. `http://hadoop.apache.org/`. Last accessed: 2022-09-16.

[15] Ahmed Eldawy, Louai Alarabi, and Mohamed F. Mokbel. Spatial partitioning techniques in spatial hadoop. *Proc. VLDB Endow.*, 8(12):1602–1605, 2015.

[16] Ahmed Eldawy, Yuan Li, Mohamed F. Mokbel, and Ravi Janardan. Cg_hadoop: computational geometry in mapreduce. In *SIGSPATIAL/GIS*, pages 284–293. ACM, 2013.

[17] Ahmed Eldawy and Mohamed F. Mokbel. Pigeon: A spatial mapreduce language. In *ICDE*, pages 1242–1245. IEEE Computer Society, 2014.

[18] Mohamed F. Mokbel, Louai Alarabi, Jie Bao, Ahmed Eldawy, Amr Magdy, Mohamed Sarwat, Ethan Waytas, and Steven Yackel. MNTG: an extensible web-based traffic generator. In *SSTD*, volume 8098 of *Lecture Notes in Computer Science*, pages 38–55. Springer, 2013.

[19] Louai Alarabi, Ahmed Eldawy, Rami Alghamdi, and Mohamed F. Mokbel. TAREEG: a mapreduce-based web service for extracting spatial data from openstreetmap. In *SIGMOD*, pages 897–900. ACM, 2014.

[20] Amr Magdy, Louai Alarabi, Saif Al-Harthi, Mashaal Musleh, Thanaa M. Ghanem, Sohaib Ghani, and Mohamed F. Mokbel. Taghreed: a system for querying, analyzing, and visualizing geotagged microblogs. In *SIGSPATIAL/GIS*, pages 163–172. ACM, 2014.

[21] Ahmed Eldawy, Mohamed F. Mokbel, Saif Al-Harthi, Abdulhadi Alzaidy, Kareem Tarek, and Sohaib Ghani. SHAHED: A mapreduce-based system for querying and visualizing spatio-temporal satellite data. In *ICDE*, pages 1585–1596. IEEE Computer Society, 2015.

[22] Ahmed Eldawy, Mohamed F. Mokbel, and Christopher Jonathan. Hadoopviz: A mapreduce framework for extensible visualization of big spatial data. In *ICDE*, pages 601–612. IEEE Computer Society, 2016.

[23] Louai Alarabi and Mohamed F. Mokbel. A demonstration of st-hadoop: A mapreduce framework for big spatio-temporal data. *Proc. VLDB Endow.*, 10(12):1961–1964, 2017.

[24] Louai Alarabi. Summit: a scalable system for massive trajectory data management. *ACM SIGSPATIAL Special*, 10(3):2–3, 2018.

[25] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. Query processing on heterogeneous cpu/gpu systems. *ACM Comput. Surv.*, 55(1), 2022.

[26] David A Patterson Hennessy. Computer architecture: A quantitative approach by john l. *Hennessy, David A. Patterson*, 2017.

[27] David W. Wall. Limits of instruction-level parallelism. In *ASPLOS*, pages 176–188. ACM Press, 1991.

[28] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, may 2011.

[29] Nvidia a100 tensor core gpu architecture. `https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf`. Last accessed: 2022-09-08.

[30] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53, mar 2008.

[31] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Comput. Sci. Eng.*, 12(3):66–73, 2010.

[32] Using shared memory in cuda c/c++. `https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/`. Last accessed: 2022-09-08.

[33] Faster parallel reductions on kepler. `https://developer.nvidia.com/blog/faster-parallel-reductions-kepler/`. Last accessed: 2022-09-08.

[34] Mikhail Khalilov and Alexey Timoveev. Performance analysis of CUDA, OpenACC and OpenMP programming models on TESLA v100 GPU. *Journal of Physics: Conference Series*, 1740(1):012056, jan 2021.

[35] S.B. Imandoust and Mohammad Bolandraftar. Application of k-nearest neighbor (knn) approach for predicting economic events theoretical background. *Int J Eng Res Appl*, 3:605–610, 01 2013.

[36] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.

[37] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, 1984.

[38] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[39] Michael Vassilakopoulos and Yannis Manolopoulos. External balanced regular (x-BR) trees: New structures for very large spatial databases. In *Advances in Informatics (Proc. HCI'99)*, pages 324–333. World Scientific, 2000.

[40] George Roumelis, Michael Vassilakopoulos, Thanasis Loukopoulos, Antonio Corral, and Yannis Manolopoulos. The xbr$^+$-tree: An efficient access method for points. In *DEXA*, volume 9261 of *Lecture Notes in Computer Science*, pages 43–58. Springer, 2015.

[41] George Roumelis, Michael Vassilakopoulos, Antonio Corral, and Yannis Manolopoulos. Efficient query processing on large spatial databases: A performance study. *J. Syst. Softw.*, 132:165–185, 2017.

[42] Frank T. Hady, Annie P. Foong, Bryan Veal, and Dan Williams. Platform storage performance with 3d xpoint technology. *Proc. IEEE*, 105(9):1822–1833, 2017.

[43] Michael Cornwell. Anatomy of a solid-state drive. *Commun. ACM*, 55(12):59–63, 2012.

[44] Sangyeun Cho, Sanghoan Chang, and Insoon Jo. The solid-state drive technology, today and tomorrow. In *ICDE*, pages 1520–1522. IEEE Computer Society, 2015.

[45] Hongchan Roh, Sungho Kim, Daewook Lee, and Sanghyun Park. AS b-tree: A study of an efficient b+-tree for ssds. *J. Inf. Sci. Eng.*, 30(1):85–106, 2014.

[46] George Roumelis, Michael Vassilakopoulos, Antonio Corral, Athanasios Fevgas, and Yannis Manolopoulos. Spatial batch-queries processing using xbr^{+} -trees in solid-state drives. In *MEDI*, volume 11163 of *Lecture Notes in Computer Science*, pages 301–317. Springer, 2018.

[47] George Roumelis, Michael Vassilakopoulos, and Antonio Corral. Performance comparison of xbr-trees and r*-trees for single dataset spatial queries. In *ADBIS*, volume 6909 of *Lecture Notes in Computer Science*, pages 228–242. Springer, 2011.

[48] George Roumelis, Michael Vassilakopoulos, Antonio Corral, and Yannis Manolopoulos. Bulk-loading xbr^{+}-trees. In *MEDI*, volume 9893 of *Lecture Notes in Computer Science*, pages 57–71. Springer, 2016.

[49] George Roumelis, Michael Vassilakopoulos, Antonio Corral, and Yannis Manolopoulos. Bulk insertions into xbr^{+}-trees. In *MEDI*, volume 10563 of *Lecture Notes in Computer Science*, pages 185–199. Springer, 2017.

[50] Hongchan Roh, Sanghyun Park, Sungho Kim, Mincheol Shin, and Sang-Won Lee. B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives. *Proc. VLDB Endow.*, 5(4):286–297, 2011.

[51] Chin-Hsien Wu, Li-Pin Chang, and Tei-Wei Kuo. An efficient r-tree implementation over flash-memory storage systems. In *ACM-GIS*, pages 17–24. ACM, 2003.

[52] Maciej Pawlik and Wojciech Macyna. Implementation of the aggregated r-tree over flash memory. In *DASFAA*, volume 7240 of *Lecture Notes in Computer Science*, pages 65–72. Springer, 2012.

[53] Yanfei Lv, Jing Li, Bin Cui, and Xuexuan Chen. Log-compact r-tree: An efficient spatial index for SSD. In *DASFAA*, volume 6637 of *Lecture Notes in Computer Science*, pages 202–213. Springer, 2011.

[54] Peiquan Jin, Xike Xie, Na Wang, and Lihua Yue. Optimizing r-tree for flash memory. *Expert Syst. Appl.*, 42(10):4676–4686, 2015.

[55] Guohui Li, Pei Zhao, Ling Yuan, and Sheng Gao. Efficient implementation of a multi-dimensional index structure over flash memory storage systems. *J. Supercomput.*, 64(3):1055–1074, 2013.

[56] Song Lin, Demetrios Zeinalipour-Yazti, Vana Kalogeraki, Dimitrios Gunopulos, and Walid A. Najjar. Efficient indexing data structures for flash-based sensor devices. *ACM Trans. Storage*, 2(4):468–503, 2006.

[57] Athanasios Fevgas and Panayiotis Bozanis. Grid-file: Towards to a flash efficient multi-dimensional index. In *DEXA*, volume 9262 of *Lecture Notes in Computer Science*, pages 285–294. Springer, 2015.

[58] Mohamed Sarwat, Mohamed F. Mokbel, Xun Zhou, and Suman Nath. FAST: A generic framework for flash-aware spatial trees. In *SSTD*, volume 6849 of *Lecture Notes in Computer Science*, pages 149–167. Springer, 2011.

[59] Anderson Chaves Carniel, Ricardo Rodrigues Ciferri, and Cristina Dutra de Aguiar Ciferri. A generic and efficient framework for spatial indexing on flash-based solid state drives. In *ADBIS*, volume 10509 of *Lecture Notes in Computer Science*, pages 229–243. Springer, 2017.

[60] Hongchan Roh, Sanghyun Park, Mincheol Shin, and Sang-Won Lee. Mpsearch: Multi-path search for tree-based indexes to exploit internal parallelism of flash ssds. *IEEE Data Eng. Bull.*, 37(2):3–11, 2014.

[61] Jianting Zhang and Simin You. Large-scale geospatial processing on multi-core and many-core processors: Evaluations on cpus, gpus and mics. *CoRR*, abs/1403.0802, 2014.

[62] Simin You, Jianting Zhang, and Le Gruenwald. Parallel spatial query processing on gpus using r-trees. In *ACM SIGSPATIAL*, pages 23–31. ACM, 2013.

[63] Jianting Zhang and Simin You. Speeding up large-scale point-in-polygon test based spatial join on gpus. In *ACM SIGSPATIAL*, pages 23–32. ACM, 2012.

[64] Jianting Zhang, Simin You, and Le Gruenwald. Parallel online spatial and temporal aggregations on multi-core cpus and many-core gpus. *Inf. Syst.*, 44:134–154, 2014.

[65] Mark McKenney and Tynan McGuire. A parallel plane sweep algorithm for multi-core systems. In *ACM SIGSPATIAL*, pages 392–395. ACM, 2009.

[66] Mark McKenney, Roger Frye, Mathew Dellamano, Kevin Anderson, and Jeremy Harris. Multi-core parallelism for plane sweep algorithms as a foundation for GIS operations. *GeoInformatica*, 21(1):151–174, 2017.

[67] Francisco García-García, Antonio Corral, Luis Iribarne, Michael Vassilakopoulos, and Yannis Manolopoulos. Efficient large-scale distance-based join queries in spatialhadoop. *GeoInformatica*, 22(2):171–209, 2018.

[68] Francisco García-García, Antonio Corral, Luis Iribarne, and Michael Vassilakopoulos. Voronoi-diagram based partitioning for distance join query processing in spatialhadoop. In *MEDI*, volume 11163 of *Lecture Notes in Computer Science*, pages 251–267. Springer, 2018.

[69] George Roumelis, Polychronis Velentzas, Michael Vassilakopoulos, Antonio Corral, Athanasios Fevgas, and Yannis Manolopoulos. Parallel processing of spatial batch-queries using xbr^{+}-trees in solid-state drives. *Cluster Computing*, 23(3):1555–1575, 2020.

[70] Hui-Ling Chen, Bo Yang, Gang Wang, Jie Liu, Xin Xu, Sujing Wang, and Dayou Liu. A novel bankruptcy prediction model based on an adaptive fuzzy k-nearest neighbor method. *Knowl. Based Syst.*, 24(8):1348–1359, 2011.

[71] Ching-Hsue Cheng, Chia-Pang Chan, and Yu-Jheng Sheu. A novel purity-based k nearest neighbors imputation method and its application in financial distress prediction. *Eng. Appl. Artif. Intell.*, 81:283–299, 2019.

[72] Gerassimos Barlas. *Multicore and GPU Programming: An Integrated Approach*. Morgan Kaufmann, 1st edition, 2014.

[73] Nvidia cuda runtime api. `https://docs.nvidia.com/cuda/cuda-runtime-api/index.html`. Last accessed: 2020-01-12.

[74] Quansheng Kuang and Lei Zhao. A practical gpu based knn algorithm. In *ISCSCI*, pages 151–155, 2009.

[75] Vincent Garcia, Eric Debreuve, Frank Nielsen, and Michel Barlaud. K-nearest neighbor search: Fast gpu-based implementations and application to high-dimensional feature matching. In *ICIP*, pages 3757–3760. IEEE, 2010.

[76] Shenshen Liang, Cheng Wang, Ying Liu, and Liheng Jian. Cuknn: A parallel implementation of k-nearest neighbor on cuda-enabled gpu. In *YC-ICT*, pages 415–418, 2009.

[77] Ivan Komarov, Ali Dashti, and Roshan M. D'Souza. Fast k-nng construction with gpu-based quick multi-select. *PloS ONE*, 9(5):1–9, 2014.

[78] Nikos Sismanis, Nikos Pitsianis, and Xiaobai Sun. Parallel search of k-nearest neighbors with synchronous operations. In *HPEC*, pages 1–6. IEEE, 2012.

[79] Ahmed Shamsul Arefin, Carlos Riveros, Regina Berretta, and Pablo Moscato. Gpu-fs-knn: A software tool for fast and scalable knn computation using gpus. *PloS ONE*, 7(8):1–13, 2012.

[80] Pablo David Gutiérrez, Miguel Lastra, Jaume Bacardit, José Manuel Benítez, and Francisco Herrera. Gpu-sme-knn: Scalable and memory efficient knn and lazy learning using gpus. *Inf. Sci.*, 373:165–182, 2016.

[81] Ricardo J. Barrientos, José Ignacio Gómez, Christian Tenllado, Manuel Prieto-Matías, and Mauricio Marín. knn query processing in metric spaces using gpus. In *Euro-Par*, volume 6852 of *Lecture Notes in Computer Science*, pages 380–392. Springer, 2011.

[82] Ricardo J. Barrientos, Fabricio Millaguir, José L. Sánchez, and Enrique Arias. Gpu-based exhaustive algorithms processing knn queries. *The Journal of Supercomputing*, 73(10):4611–4634, 2017.

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

[84] Jan Masek, Radim Burget, Jan Karasek, Václav Uher, and Malay Kishore Dutta. Multi-gpu implementation of k-nearest neighbor algorithm. In *TSP*, pages 764–767. IEEE, 2015.

[85] Shengren Li and Nina Amenta. Brute-force k-nearest neighbors search on the GPU. In *SISAP*, volume 9371 of *Lecture Notes in Computer Science*, pages 259–270. Springer, 2015.

[86] Amreek Singh, Kusum Deep, and Pallavi Grover. A novel approach to accelerate calibration process of a k-nearest neighbours classifier using GPU. *J. Parallel Distributed Comput.*, 104:114–129, 2017.

[87] Fast k nearest neighbor search using gpu. `http://vincentfpgarcia.github.io/kNN-CUDA/`. Last accessed: 2020-01-12.

[88] Polychronis Velentzas, Michael Vassilakopoulos, and Antonio Corral. In-memory k nearest neighbor gpu-based query processing. In *GISTAM*, pages 310–317. SCITEPRESS, 2020.

[89] Polychronis Velentzas, Panagiotis Moutafis, and George Mavrommatis. An improved gpu-based algorithm for processing the k nearest neighbor query. In *PCI*, pages 372–375. ACM, 2020.

[90] Polychronis Velentzas, Michael Vassilakopoulos, and Antonio Corral. Gpu-based algorithms for processing the k nearest-neighbor query on disk-resident data. In *MEDI*, volume 12732 of *Lecture Notes in Computer Science*, pages 264–278. Springer, 2021.

[91] Gang Mei, Nengxiong Xu, and Liangliang Xu. Improving gpu-accelerated adaptive IDW interpolation algorithm using fast knn search. *CoRR*, abs/1601.05904, 2016.

[92] Fabian Gieseke, Justin Heinermann, Cosmin E. Oancea, and Christian Igel. Buffer k-d trees: Processing massive nearest neighbor queries on gpus. In *ICML*, pages 172–180. JMLR.org, 2014.

[93] Pedro Jose Silva Leite, João Marcelo X. N. Teixeira, Thiago S. M. C. de Farias, Bernardo Reis, Veronica Teichrieb, and Judith Kelner. Nearest neighbor searches on the GPU - A massively parallel approach for dynamic point clouds. *Int. J. Parallel Program.*, 40(3):313–330, 2012.

[94] Moohyeon Nam, Jinwoong Kim, and Beomseok Nam. Parallel tree traversal for nearest neighbor query on the GPU. In *ICPP*, pages 113–122. IEEE, 2016.

[95] Jia Pan, Christian Lauterbach, and Dinesh Manocha. Efficient nearest-neighbor computation for gpu-based motion planning. In *IROS*, pages 2243–2248. IEEE, 2010.

[96] Sparsh Mittal and Jeffrey S. Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Trans. Parallel Distributed Syst.*, 27(5):1537–1550, 2016.

[97] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry - An Introduction*. Texts and Monographs in Computer Science. Springer, 1985.

[98] Klaus H. Hinrichs, Jürg Nievergelt, and Peter Schorn. Plane-sweep solves the closest pair problem elegantly. *Inf. Process. Lett.*, 26(5):255–261, 1988.

[99] Puloma Katiyar, Tin Vu, Ahmed Eldawy, Sara Migliorini, and Alberto Belussi. Spider-web: A spatial data generator on the web. In *SIGSPATIAL/GIS*, pages 465–468. ACM, 2020.

[100] Tin Vu, Sara Migliorini, Ahmed Eldawy, and Alberto Belussi. Spatial data generators. In *SIGSPATIAL/SpatialGems*, pages 1–7. ACM, 2019.

[101] Sparsh Mittal. A survey on optimized implementation of deep learning models on the NVIDIA jetson platform. *J. Syst. Archit.*, 97:428–442, 2019.

[102] Dhirendra Pratap Singh, Ishan Joshi, and Jaytrilok Choudhary. Survey of gpu based sorting algorithms. *International Journal of Parallel Programming*, 46(6):1017–1034, 2018.

[103] Vincent Garcia, Eric Debreuve, and Michel Barlaud. Fast k nearest neighbor search using GPU. In *CVPR Workshops*, pages 1–6. IEEE, 2008.

[104] Javier A. Riquelme, Ricardo J. Barrientos, Ruber Hernández-García, and Cristóbal A. Navarro. An exhaustive algorithm based on GPU to process a knn query. In *SCCC*, pages 1–8. IEEE, 2020.

[105] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[106] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics*, 27(5):126, 2008.

[107] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, pages 604–613. ACM, 1998.

[108] Jia Pan and Dinesh Manocha. Fast gpu-based locality sensitive hashing for k-nearest neighbor computation. In *SIGSPATIAL/GIS*, pages 211–220. ACM, 2011.

[109] Patrick Wieschollek, Oliver Wang, Alexander Sorkine-Hornung, and Hendrik P. A. Lensch. Efficient large-scale approximate nearest neighbor search on the GPU. *CoRR*, abs/1702.05911, 2017.

[110] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57. ACM, 1984.

[111] David A. White and Ramesh C. Jain. Similarity indexing with the ss-tree. In *ICDE*, pages 516–523. IEEE, 1996.

[112] Ablimit Aji, Hoang Vo, and Fusheng Wang. Effective spatial data partitioning for scalable query processing. *CoRR*, abs/1509.00910:1–12, 2015.

[113] Polychronis Velentzas, Michael Vassilakopoulos, and Antonio Corral. Gpu-aided edge computing for processing the k nearest-neighbor query on ssd-resident data. *Internet of Things*, 15:100428, 2021.

[114] José M. Cecilia, Juan-Carlos Cano, Juan Morales-García, Antonio Llanes, and Baldomero Imbernón. Evaluation of clustering algorithms on gpu-based edge computing platforms. *Sensors*, 20(21):6335, 2020.

[115] Pilsung Kang and Sungmin Lim. A taste of scientific computing on the gpu-accelerated edge device. *IEEE Access*, 8:208337–208347, 2020.

[116] Sungmin Lim and Pilsung Kang. Implementing scientific simulations on gpu-accelerated edge devices. In *ICOIN*, pages 756–760. IEEE, 2020.

[117] Jongmin Jo, Sucheol Jeong, and Pilsung Kang. Benchmarking gpu-accelerated edge devices. In *BigComp*, pages 117–120. IEEE, 2020.

[118] Lingyuan Wang, Miaoqing Huang, and Tarek A. El-Ghazawi. Exploiting concurrent kernel execution on graphic processing units. In *HPCS*, pages 24–32. IEEE, 2011.

[119] Florian Wende, Frank Cordes, and Thomas Steinke. On improving the performance of multi-threaded cuda applications with concurrent kernel execution by kernel reordering. In *SAAHPC*, pages 74–83. IEEE, 2012.

[120] Qing Jiao, Mian Lu, Huynh Phung Huynh, and Tulika Mitra. Improving gpgpu energy-efficiency through concurrent kernel execution and dvfs. In *CGO*, pages 1–11. IEEE, 2015.

[121] Hongwen Dai, Zhen Lin, Chao Li, Chen Zhao, Fei Wang, Nanning Zheng, and Huiyang Zhou. Accelerate gpu concurrent kernel execution by mitigating memory pipeline stalls. In *HPCA*, pages 208–220. IEEE, 2018.

[122] Zhen Lin, Hongwen Dai, Michael Mantor, and Huiyang Zhou. Coordinated cta combination and bandwidth partitioning for gpu concurrent kernel execution. *ACM Transactions on Architecture and Code Optimization*, 16(3):23:1–23:27, 2019.

[123] Chen Zhao, Wu Gao, Feiping Nie, Fei Wang, and Huiyang Zhou. Fair and cache blocking aware warp scheduling for concurrent kernel execution on gpu. *Future Generation Computer Systems*, 112:1093–1105, 2020.

[124] Bernabé López-Albelda, Francisco M. Castro, José María González-Linares, and Nicolás Guil. Flexsched: Efficient scheduling techniques for concurrent kernel execution on gpus. *The Journal of Supercomputing*, 78(1):43–71, 2022.

[125] Chen Zhao, Wu Gao, Feiping Nie, and Huiyang Zhou. A survey of gpu multitasking methods supported by hardware architecture. *IEEE Transactions on Parallel and Distributed Systems*, 33(6):1451–1463, 2022.

[126] Xin Li, Huayan Yu, Ligang Yuan, and Xiaolin Qin. Query optimization for distributed spatio-temporal sensing data processing. *Sensors*, 22(5), 2022.

[127] Muhammad Abdelghaffar Awad. *Fully Concurrent GPU Data Structures*. PhD thesis, University of California, Davis, 2022.

[128] George Roumelis, Polychronis Velentzas, Michael Vassilakopoulos, Antonio Corral, Athanasios Fevgas, and Yannis Manolopoulos. Parallel processing of spatial batch-queries using xbr$^+$-trees in solid-state drives. *Cluster Computing*, 23(3):1555–1575, 2020.

[129] Mingchang Cheng, Tiefeng Ma, Lin Ma, Jian Yuan, and Qijing Yan. Adaptive grid-based forest-like clustering algorithm. *Neurocomputing*, 481:168–181, 2022.

[130] Sara Migliorini, Alberto Belussi, Elisa Quintarelli, and Damiano Carra. A context-based approach for partitioning big data. In *EDBT*, pages 431–434. OpenProceedings.org, 2020.

[131] Antonio Corral, Yannis Manolopoulos, Yannis Theodoridis, and Michael Vassilakopoulos. Closest pair queries in spatial databases. In *SIGMOD*, pages 189–200. ACM, 2000.