Università
Ca'Foscari
Venezia

Scuola Dottorale di Ateneo
Graduate School

Dottorato di ricerca
in Informatica
Ciclo XXVII
Anno di discussione 2014

Analysis and Fast Querying of Mobility Data

SETTORE SCIENTIFICO DISCIPLINARE DI AFFERENZA: INF/01
Tesi di Dottorato di Francesco Lettich, matricola 819700

Coordinatore del Dottorato	Tutore del Dottorando
Prof. Riccardo Focardi	Prof. Salvatore Orlando

Università Ca' Foscari di Venezia

Dottorato di Ricerca in Informatica, XXVII ciclo

Ph.D. Thesis

Analysis and Fast Querying of Mobility Data

Francesco Lettich

Supervisor

Salvatore Orlando

PhD Coordinator

Riccardo Focardi

Referee

Gianluigi Greco

Referee

Yannis Theodoridis

Author's Web Page: http://informatica.dais.unive.it/~lettich

Author's e-mail: lettich@dais.unive.it

Author's address:

Dipartimento di Informatica
Università Ca' Foscari di Venezia
Via Torino, 155
30172 Venezia Mestre – Italia
tel. +39 041 2348411
fax. +39 041 2348419
web: http://www.dsi.unive.it

To my father, Giulio, and my mother, Rosa.

Abstract

Mobility data represents a widely used term to indicate sources of information, possibly structured in many different ways - depending on technologies and formats used - describing the localization or the movement, in time and space, of sets of entities. Regardless of specific ways through which mobility data is represented, this class of information is nowadays pervasive since it is massively produced, processed and analyzed for many different purposes. The importance of mobility data is going to increase even further in the future, given the ever growing diffusion of the *internet of devices* and the progressive introduction of the *internet of things* paradigm. In this thesis we contribute to mobility data research by addressing separately two distinct problems.

The first one is related to the on-line processing of streams of mobility data coming from massive amounts of moving objects, where such streams contain location updates and some kind of queries continuously and periodically issued by the objects. This problem is frequently met, nowadays, in the context of *Location-Based Services* (LBS) or *Location-Based Social Networking* applications (LBSN), even if one has to observe that the nature of the problem allows it to be possibly found in quite diverse domains, such as massively multiplayer online games, anti-collision detection systems, behavioural simulations and so on. More precisely, we focus on the problem of computing massive *range* and *k-nearest neighbour* queries, which represents the time-dominant phase of the whole processing. In order to tackle effectively the problem we exploit the remarkable - yet cheap - computational power of modern GPUs by introducing novel algorithms and data structures, and we prove the effectiveness of our solutions through an extensive series of experiments.

The second problem relates to the domain of mobility data mining. In this context the main goal is to devise novel, off-line analyses able to extract previously unknown and interesting patterns from raw mobility data. This kind of research is, in general, very interesting since it allows to gain new insights on mobility data. We address the problem of detecting *avoidance behaviours between moving objects* from historical movement traces. To this end, we first introduce a framework which formally defines what is an avoidance behaviour between moving objects; subsequently, on the basis of such framework we provide an algorithm which is able to extract these patterns. Finally, we experimentally prove the effectiveness of our solution with real-world datasets.

Acknowledgments

I would like to thank my family and Letizia, since they fully supported me along this path...I will always be grateful to you.

I would also like to thank my supervisor, Salvatore Orlando, along with Claudio Silvestri and Alessandra Raffaetà, since their guidance and patience were fundamental during these three, long years; I'm also grateful to Christian Jensen for the beautiful stay I had at the computer science department in Aarhus.

I thank Gianluigi Greco and Yannis Theodoridis for having accepted to review my thesis and provide useful comments, suggestions and corrections.

Finally, I would like to thank my colleagues, as well as the fantastic folks I met in Aarhus, with whom I shared many beautiful moments.

Contents

Introduction **1**
I.1 Thesis content . . . 2
I.2 Thesis contributions . . . 3

I First part **5**

1 Processing repeated range and k-NN queries over massive moving objects **7**
1.1 Problem Setting and Statement . . . 9
1.1.1 Problem Setting . . . 9
1.1.2 Batch Processing . . . 11
1.1.3 Query Semantics . . . 11
1.1.4 Quality of Service - Query Latency . . . 12
1.1.5 Problem Statement . . . 13
1.2 Graphics Processing Units . . . 14
1.2.1 Main algorithmic design issues . . . 15
1.3 Related work . . . 15
1.3.1 Processing repeated range queries over massive moving objects observations . . . 15
1.3.2 Processing repeated k-NN queries over massive moving objects observations . . . 17

2 GPU-Based processing of repeated range queries **21**
2.1 Spatial indexing and data structures . . . 22
2.1.1 Design considerations . . . 23
2.1.2 Overview of the methods . . . 23
2.1.3 Space partitioning and indexing . . . 24
2.1.4 Data structures . . . 26
2.2 Query processing pipeline . . . 28
2.2.1 Pipeline description . . . 28
2.2.2 Index creation and indexing in UG and $\mathsf{UG}_{Baseline}$. . . . 29
2.2.3 Index Creation and Indexing in QUAD. . . . 31
2.2.4 Filtering . . . 38
2.2.5 Bitmap decoding . . . 44
2.2.6 Optimizations . . . 46
2.3 Experimental Setup . . . 47

2.4 Experimental Evaluation . . . 48
2.4.1 Analysis on the benefits coming from the usage of bitmaps (S1) 50
2.4.2 Covering subqueries optimization (S2) . . . 51
2.4.3 Task scheduling policy (S3) . . . 53
2.4.4 Data skewness and optimal grid coarseness for UG (S4) . . . 56
2.4.5 Data skewness and optimal cell size for QUAD (S5) . . . 58
2.4.6 Impact of spatial distribution skewness on the performance (S6) 59
2.4.7 Performance analysis for different spatial distributions, amount of objects, and query areas (S7) . . . 61
2.4.8 Bandwidth analysis (S8) . . . 63

3 GPU-Based processing of repeated k-NN queries 69
3.1 K-NN_{GPU} overview . . . 70
3.1.1 Motivating challenges . . . 70
3.1.2 Relevant data structures . . . 71
3.2 Processing Pipeline . . . 73
3.2.1 Index Creation and Moving Objects Indexing. . . . 73
3.2.2 Iterative k-NN queries computation . . . 75
3.3 Experimental Setup . . . 92
3.4 Experimental Evaluation . . . 93
3.4.1 (S1) Tree height, neighbours list size, query rate and spatial skewness impacts on K-NN_{GPU}'s performance . . . 95
3.4.2 (S2) K-NN_{GPU} vs K-$NN_{BASELINE}$ . . . 98
3.4.3 (S3) K-NN_{GPU} vs K-NN_{CPU} . . . 99
3.4.4 (S4) Bandwidth analysis . . . 105

II Second part 109

4 Detecting avoidance behaviours between moving objects 111
4.1 Introduction and Motivation . . . 111
4.2 Related Work . . . 113
4.3 Preliminaries . . . 115
4.4 Avoidance . . . 116
4.4.1 Avoidance Classification . . . 119
4.4.2 Problem Statement . . . 121
4.5 Algorithmic Framework . . . 123
4.5.1 An Algorithm for Avoidance Detection . . . 123
4.5.2 Avoidance Detectors . . . 125
4.6 Experimental Evaluation . . . 126
4.6.1 Experimental Setup . . . 127
4.6.2 Analysis of the Ground Truth Dataset . . . 130
4.6.3 Analysis of a Real World Unannotated Dataset . . . 134

Conclusions 141

Bibliography 143

List of Figures

1.1 Moving objects, issued queries q and position updates u. 7
1.2 Example of a set of moving objects issuing k-NN queries. 8
1.3 Timeline 11

2.1 Simple mapping example over a quadtree-induced grid. 26
2.2 Example of a structure of vectors used on a set of objects described by 3-tuples. 27
2.3 A simple example of a GPU-based sorting, based on the structure of vectors representation, of 8 entities according to their Morton codes. The discontinuities among the codes (thicker lines) determine the set of entities belonging to each cell. 31
2.4 Example of quadtree construction with 7 objects, $th_{quad} = 1$ and $l_{max} = 2$. 34
2.5 Example of the mapping established by z_{map} between the quadtree-induced grid $\mathcal{C}$ (left side) and the uniform grid $\mathcal{C}^{l_{deep}}$ (right side) related to the quadtree deepest level. 36
2.6 Query indexing example with QUAD. 37
2.7 Bitmap layouts used during the processing. 39
2.8 Collective creation of a interlaced bitmap by a block of GPU threads. 41
2.9 Collective linearization of a group of columns by a warp of GPU threads. 42
2.10 UG vs. $\text{UG}_{Baseline}$ time analysis for uniform datasets by varying the number of objects in [100K,1000K]. The histograms of the UG filtering and decoding phases are time-stacked for clarity purposes. 51
2.11 Gaussian datasets, 500K objects, query area $(400u)^2$, varying hotspots in [50,1000], query rate 100%, Covering ON vs OFF. 53
2.12 Gaussian datasets with 50 hotspots, 500K objects, query area varied in $[(200u)^2, (400u)^2]$, query rate 100%, Covering ON vs. OFF. 54
2.13 Analysis on the performances and workload redistribution among the GPU streaming multiprocessor with and without the static task list reordering - gaussian datasets, 500K objects, query area $(400u)^2$, query rate 100%, varying amount of hotspots. The top plot refers to the execution times observed while the bottom one refers to the profiling data collected during the filtering phase (decoding phase data is analogous). 55
2.14 Gaussian dataset, 200K objects, query area $(400u)^2$, query rate 100%, 150 hotspots. The optimal value is equal to 110. Logscale on the y-axis is conveniently used to magnify small differences in the filtering execution times. 57

2.15 Gaussian dataset, 200K objects, query area $(400u)^2$, query rate 100%, 20 hotspots. The optimal value is equal to 95. Logscale on the y-axis is conveniently used to magnify small differences in the filtering execution times. 57
2.16 Performance analysis with different QUAD th_{quad} values when varying the skewness degree. 58
2.17 Performance analysis with different QUAD th_{quad} values when considering different query areas. 59
2.18 Gaussian datasets, 500K objects, query area $(400u)^2$, amount of hotspots varied in [10,200], average running times per tick and speedup against CPU-ST. 60
2.19 Gaussian datasets, 500K objects, query area $(400u)^2$, amount of hotspots varied in [10,200], mean and dispersion index over the grid active cells. 61
2.20 Varying the number of objects: average running time per tick and speedup versus CPU-ST. From top to bottom: uniform datasets, gaussian datasets with 25 hotspots, and San Francisco Network datasets. 62
2.21 Varying the query area: average running time per tick and speedup versus CPU-ST. From top to bottom: uniform datasets, gaussian datasets with 25 hotspots, and San Francisco Network datasets. . . . 64
2.22 Variably sized queries: average running time per tick and speedup versus CPU-ST. Uniform datasets 65
2.23 Variably sized queries: average running time per tick and speedup versus CPU-ST. Gaussian datasets. 65
2.24 Variably sized queries: average running time per tick and speedup versus CPU-ST. Network datasets. 66
2.25 System bandwidth analysis when varying the amount of moving objects or the dataset skewness. 67
2.26 System bandwidth analysis when varying the query area or the query rate. 67

3.1 Interlaced and linear result set layouts. In the Figure we denote the j-th result of query q_i as r_i^j. 72
3.2 Toy example of a 1-NN query for which we have to analyze the content of neighbouring quadrants in order to compute the final, correct result set. 84
3.3 Left and right sub-visits, quadtree nodes coverage example. 85
3.4 Relationship between the tree height (indirectly controlled through th_{quad}) and k, and its repercussions on K-NN$_{\mathsf{GPU}}$'s performance. 96
3.5 Skewness repercussions on th_{quad}'s optimality. 97

3.6 Query rate influence on K-NN$_{GPU}$'s performance. The amount of objects (500K), the neighbours list size ($k = 32$) and the maximum amount of objects per quadtree leaf ($th_{quad} = 384$) are all fixed across the experiments. . . . 97
3.7 K-NN$_{GPU}$ vs K-NN$_{BASELINE}$, variable amount of moving objects, $k = 32$. 98
3.8 K-NN$_{GPU}$ vs K-NN$_{BASELINE}$, variable nearest neighbours list size k. . . . 99
3.9 K-NN$_{GPU}$ vs K-NN$_{CPU}$, variable amount of moving objects, uniform datasets. . . . 100
3.10 K-NN$_{GPU}$ vs K-NN$_{CPU}$, variable amount of moving objects, gaussian datasets, 25 hotspots. . . . 100
3.11 K-NN$_{GPU}$ vs K-NN$_{CPU}$, variable amount of moving objects, network-based (San Francisco) datasets. . . . 101
3.12 K-NN$_{GPU}$ vs K-NN$_{CPU}$, variable neighbours list size k, uniform dataset (1M objects). . . . 102
3.13 K-NN$_{GPU}$ vs K-NN$_{CPU}$, variable neighbours list size k, gaussian dataset (25 hotspots, 1M objects). . . . 102
3.14 K-NN$_{GPU}$ vs K-NN$_{CPU}$, variable neighbours list size k, network dataset (1M objects). . . . 103
3.15 K-NN$_{GPU}^{COALESCE}$ vs K-NN$_{GPU}^{CACHE}$ vs K-NN$_{CPU}$, variable neighbours list size k, uniform dataset (1M objects). . . . 103
3.16 K-NN$_{GPU}^{COALESCE}$ vs K-NN$_{GPU}^{CACHE}$ vs K-NN$_{CPU}$, variable neighbours list size k, gaussian dataset (25 hotspots, 1M objects). . . . 104
3.17 K-NN$_{GPU}^{COALESCE}$ vs K-NN$_{GPU}^{CACHE}$ vs K-NN$_{CPU}$, variable neighbours list size k, network dataset (1M objects). . . . 104
3.18 System bandwidth analysis when varying the amount of moving objects or the dataset skewness. . . . 106
3.19 System bandwidth analysis when varying the query rate or the neighbours list size k. . . . 106

4.1 Different kinds of avoidance behaviors: avoidance with respect to a static object (a), and avoidance between moving objects: *individual* (b), *mutual* (c), and *individual* induced by a change in speed (d). . . . 112
4.2 *A meets B* (the distance in t' is less than δ) during $[t_1, t_2]$ but not during $[t_2, t_3]$. . . . 116
4.3 Different kinds of predictors based on several interpolations and on movement constraints (road network). . . . 117
4.4 According to the forecasts, *A will meet B* (the distance will be less than δ) at some time t' in the time interval $[t_1, t_2]$. . . . 118
4.5 *A avoids B* during the time interval $[t_1, t_2]$: meet is expected according to the forecast computed at time t_1 but no actual meet happened during the given time interval. . . . 119

4.6 Trajectory A *changes behavior*: at some time t' during $[t_1, t_2]$ the distance of the actual position from the forecast position is greater than δ. . . . 120
4.7 On the left, both A and B change behavior to avoid each other (*mutual* avoidance). On the right, according to the forecast A and B should meet but this does not happen even if both did not significantly change behavior (*weak* avoidance). . . . 121
4.8 Examples of three visual inspections performed on three different avoidances returned by the algorithm. . . . 131
4.9 Decision problem with simple detector: F-Measure analysis. . . . 131
4.10 Decision problem with fused detector: F-Measure analysis. In X-axis the values are the maximums of the thresholds δ used during the fusion operation, e.g., 9 is the maximum for the set $\{3, 6, 9\}$. . . . 133
4.11 Search problem with simple detector: Q-Measure analysis. . . . 133
4.12 Search problem with fused detector: Q-Measure analysis. In X-axis the values are the maximums of the thresholds δ used during the fusion operation, e.g., 9 is the maximum for the set $\{3, 6, 9\}$. . . . 134
4.13 Example of a paired movement event involving the frequent ship 228051000. The ships are moving from bottom to top. . . . 136
4.14 Subtrajectories related to avoidance behaviors detected in a time interval spanning two months ([20/04/2009, 20/06/2009]). . . . 137

List of Tables

2.1 Data and workload generation parameters. 49

3.1 Data and workload generation parameters. 94

4.1 Confusion matrix. 128
4.2 *Frequent ships* details. 135

List of Algorithms

1 GPU-based PR-quadtree construction 32
2 QUAD and UG filtering phase. 41
3 $\text{UG}_{Baseline}$ filtering phase 43
4 Decoding phase 45

5 $distComp(\overline{C}, Q, P, k)$ 77
6 $findKDist(q, c, dist_{min}, dist_{max}, k, numBins)$ 79
7 $distCompPhase1(\overline{C}, Q, P, k)$ 80
8 $distCompPhase2(\overline{C}, Q, P, MAXDIST, NUMRES, k)$ 81
9 K-NN_{GPU} – Schema used for subsequent iterations 86
10 $navigateTree(Q_{process}, \mathcal{C}, z_{map}, k, MAXDIST, NUMRES)$ 89

11 Avoidance behavior detection 124

Introduction

Mobility data represents a widely used term to indicate sources of information, possibly structured in many different ways - depending on technologies and formats used - describing the localization or the movement, in time and space, of sets of entities. Regardless of specific ways through which mobility data is represented, this class of information is nowadays pervasive since it is massively produced, processed and analyzed for many different purposes.

Starting from the mid-nineties, when a wide-scale adoption of GPS localization systems occurred in many contexts and GPS data started to be fairly accurate, researchers and institutions began to exploit mobility data in order to extract useful and interesting patterns from it, as well as synthetic indicators.

However only in recent years, as a consequence of the tremendous expansion of the so-called *internet of devices* [1], the processing and analysis of mobility data have sparked huge interests and efforts in academic and private sectors. Indeed, considered the plethora of smarthpones or, more in general, modern mobile devices connected to the internet - each one usually equipped with a GPS antenna and a deluge of actuators and sensors - and the related layer of applications and services offered to end users, the production of mobility data reached unprecedented levels, thus raising the importance of processing and analyzing such kind of information; this claim is even more true whenever we consider that many companies make lots of profits out of it nowadays.

The importance of mobility data is going to increase even further in the foreseeable future, considered the progressive diffusion of the *internet of things* [1], a paradigm depicting scenarios where almost each object in the world will be connected to the internet by means of miniaturized mobile devices, giving life to new problems and opportunities.

This thesis is related to mobility data in two different, specific ways. The first one, which also represents the first part of the thesis (Chapters 1, 2 and 3), deals with the on-line processing of streams of mobility data coming from massive amounts of moving objects, where such streams contain location updates and (some kind of) queries continuously and periodically issued by objects. This is a problem nowadays met quite frequently in the context of *Location-Based Services* (LBS) or *Location-Based Social Networking* applications (LBSN) [2]. As a side note, considering its flexible nature this problem can be met in quite diverse domains apart from those related to mobility data, such as massively multiplayer on-line games, anti-collision detection systems, behavioural simulations, and so on. More specifically, we target scenarios where massive amounts of moving objects (which represent the users of such applications) continuously issue *updates* - in order to notify their current lo-

cation - and queries - in order to be notified about possible moving objects inside their interaction area - to some centralized infrastructure. In such applications the main challenge is to process queries quickly and efficiently, since the query processing represents the time-dominant phase of the whole processing, usually subject to some Quality of Service constraints as well. If we conveniently adopt the taxonomy introduced in [2, Ch.1] used to classify *location-aware* applications, applications conforming to such schema are classified as *mobile-mobile*, a research area of particular interest considering the multitude of smartphone applications (few examples are Google's Waze, the now-defunct Google's Latitude, social-network based applications, chat services, dating services, etc.) whose success depends on solving effectively problems similar to the one considered here.

The second part of the thesis situates itself in the ample and well-established family of off-line analyses used to extract useful and interesting patterns from mobility data. Given the tremendous amount of varied mobility data available nowadays, devising appropriate and novel analyses is more and more important since these allow to discover patterns which may bring far-reaching benefits on everyday life, or give interesting and useful insights on human mobility.

In recent literature many works focus on the discovery and extraction of patterns from trajectories, such as T-patterns, flocks, meets, periodic movements, chasing, etc. [3, 4], thus making this area of research quite active. In this thesis we contribute to such branch of research by introducing a novel pattern, trying to characterize formally what is an *avoidance behaviour*, punctually in space and time, between pairs of trajectories. More specifically, we try to model events in which pairs of trajectories, individually or mutually, change their movements, in terms of speed or direction, in order to *avoid* being "too close". We also provide an algorithm, based on the aforementioned formal characterization, able to detect such events. This contribution is presented in Chapter 4.

I.1 Thesis content

The thesis is structured as follows:

- Chapter 1 introduces the problem of processing repeated range or k-NN queries with massive moving objects observations. Subsequently, we introduce an unified framework used for modelling such processing and give an overview on modern Graphics Processing Units (GPUs), also presenting the major challenges to be tackled when considering the usage of the General Purpose Computing on GPU (GPGPU) paradigm. The chapter finally concludes with an extensive overview about related work regarding the problems introduced, especially when seen from the GPGPU perspective.

- Chapter 2 focuses specifically on the problem of processing repeated range queries over moving objects observations, presenting the approach we use to

efficiently solve the problem, namely the *Quadtree* approach.

- Chapter 3 focuses specifically on the problem of processing repeated k-NN queries over moving objects observations, presenting the approach we use to efficiently solve the problem, namely K-NN_{GPU}.

- Chapter 4 presents the work related to the formal charachterization and detection of avoidance behaviours between trajectories.

- Finally, the thesis ends by providing the final conclusions.

I.2 Thesis contributions

For what concerns the GPU-based processing of repeated, massive range or k-NN queries over massive moving objects observations, the main, relevant contributions made (also) by the author are the following:

- We introduce a formal framework which models the processing of repeated, massive amounts of range or k-NN queries over massive moving objects observations, keeping into consideration Quality of Service constraints as well.

- We introduce two query processing pipelines based on the aforementioned framework, one for range queries and the other one for k-NN queries, coupled with a PR-quadtree based spatial index - used to distribute workloads among the GPU streaming multiprocessors efficiently - and lock-free data structures used to manage intermediate query results on GPU with the aim of boosting the overall GPU memory throughput.

For what is related to research contributions, a simpler version of the framework, as well as a much simpler range query processing pipeline, are presented in [5]. In [6] we present the framework as well as the range query processing pipeline presented in this thesis as well.

Passing to the characterization and detection of avoidance behaviours between moving objects, the contributions presented in this thesis can be summarized as follows:

- We formally introduce a framework which characterizes what is an avoidance behaviour, in space and time, between moving objects.

- We formally define whenever an avoidance behaviour is *individual*, *mutual* or *weak*, based on factual evidence emerging from data.

- On the basis of the aforementioned framework, we introduce an algorithm capable of detecting avoidance behaviours between moving objects, as well as capable of classifying these events according to the above taxonomy.

For what is related to research contributions, all the aforementioned contributions are presented in [7].

I

First part

1

Processing repeated range and k-NN queries over massive moving objects

An increasing number of applications need to process massive spatial workloads. Specifically, we consider applications in settings where spatial data is continuously produced over time and needs to be processed rapidly, e.g., scenarios involving services or applications mainly tailored for mobile device infrastructures - such as *Location-Based Services* (LBS) or *Location-Based Social Networking* applications (LBSN) [2] - Massively Multiplayer Online Games (MMOG), anti-collision detection systems, behavioural simulations where the behaviours of agents may affect other agents within a given range, and so on. In these applications, very large populations of continuously *moving objects* frequently update their positions and issue some kind of queries in order to look for other objects within their interaction area. The resulting massive workloads pose new challenges to data management techniques.

In this context we consider separately two specific kinds of queries which are of particular interest: rectangular range queries, i.e., queries represented by a fixed rectangular area for which we have to determine the set of objects falling inside; and k-NN queries, globular queries whose spatial extent is dictated by the dislocation of the k nearest objects with respect to their centers.

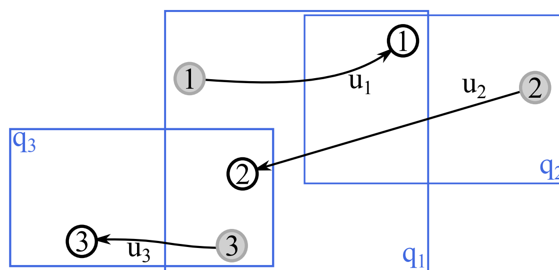


Figure 1.1: Moving objects, issued queries q and position updates u.

Figure 1.1 depicts a toy example involving range queries, where three objects, namely 1, 2, and 3, continuously issue position updates and queries. Object positions are represented as circles with object identifiers inside, while updates are depicted as arrows (labeled u_1, u_2, u_3) connecting previous positions (represented by gray

circles) with current positions. Range queries are shown as rectangles and labeled as q_1, q_2, q_3. The result set of query q_3, when executed after u_2, is $\{2\}$ (excluding the issuing object from the result set).

On the other hand, Figure 1.2 depicts a toy example involving k-NN queries. Here three objects, namely $2, 3$ and 5, issue a position update, while 1 issue a 2-NN query in order to know the set of the two nearest objects with respect to its location. The result set of such query, whenever executed after the updates, is $\{2, 3\}$.

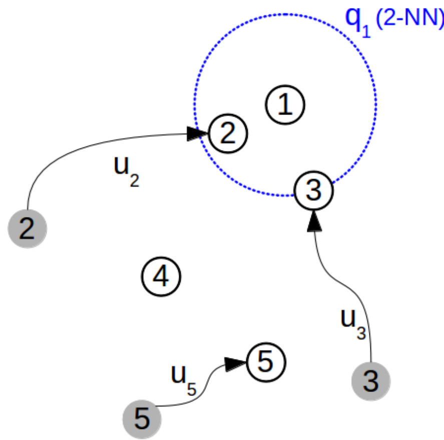


Figure 1.2: Example of a set of moving objects issuing k-NN queries.

To enable parallel processing and optimizations, and thus manage the targeted workloads in a scalable manner, we recur to an approach based on time discretization. In this sense we partition the time in intervals (or *ticks*), assign location updates and queries to the ticks in which they occur, and process the updates and queries in the resulting batches such that the query results are reported after the end of each tick. This approach has the effect of replacing the processing of a large number of independent and asynchronous queries with the *iterated processing of spatial joins* between the last known positions of all the moving objects at the end of a tick and the queries issued during the tick. In other words we trade (slightly) delayed processing of queries for increased throughput, and therefore care is needed to ensure acceptable delays.

In order to achieve high performance and scalability we also want to exploit a platform encompassing an off-the-shelf general-purpose microprocessor (CPU) coupled with a *Graphics Processing Unit* (GPU) that features hundreds of processing cores. To benefit from GPUs exploitation, limitations and peculiarities of these architectures must be carefully taken into account. Specifically, individual GPU cores are slower than those of a typical CPU, while some memory access patterns may cause serious performance degradation due to contention and serialization of memory accesses. Effective query processing techniques must address these limitations: multiple cores must work together to efficiently process queries in parallel for most of the time, and must coordinate their activities to ensure high memory bandwidth.

In light of the scenarios and goals considered, in Section 1.1 we first provide a unified framework, based on time discretization, for processing repeated range or k-NN queries over massive moving objects observations. Such framework will be used later on in Chapters 2 and 3 as a common ground to devise hybrid CPU/GPU query processing pipelines addressing the two kinds of query considered. In Section 1.2 we then provide a brief overview about modern graphics processing units (GPUs), so to show their main architectural peculiarities and limitations one has to take into consideration when designing hybrid processing pipelines. Finally, in Section 1.3 we provide an overview about works in literature relevant to the scenarios and problem setting considered.

1.1 Problem Setting and Statement

In this section we provide definitions that capture the problem setting and the problems we aim to solve.

1.1.1 Problem Setting

We consider a set of points $O = \{o_1, \ldots, o_n\}$ moving in two-dimensional Euclidean space $\mathbb{R}^2$, where the position of object o_i is given by the function $pos_i : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^2$ mapping time instants into spatial positions.

These points model objects that issue position updates and queries (range or k-NN, depending on the scenario considered) as they move. Let $\mathcal{P}_i = \langle p_i^{t_0}, \ldots, p_i^{t_k}, \ldots \rangle$, $t_j < t_{j+1}$, be the time-ordered sequence position updates issued by o_i, where $p_i^{t_j} = pos_i(t_j)$ is a position update. At this point we need to define the two typologies of queries considered.

Definition 1. *[Range query]*
We define a range query issued by object o_i at time t by

$$q_i^t = (x^a, x^b, y^a, y^b),$$

where (x^a, y^a) and (x^b, y^b) represent respectively the lower left and upper right corners of a rectangle.

Definition 2. *[k-nearest neighbours (k-NN) query]*
We define a k-NN query issued by object o_i at time t as

$$q_i^t = (x, y, k),$$

where (x, y) represents the query center while k the amount of nearest neighbours required.

Regardless of the query typology considered, we denote by $\mathcal{Q}_i = \langle q_i^{t_0}, \ldots, q_i^{t_m}, \ldots \rangle$, $t_m < t_{m+1}$, the time-ordered sequence of queries issued by o_i.

Given the above, the most recently known position of o_i before time t, $t \geq t_0$, is denoted as $\hat{p}_i^t$ and defined as follows:

$$\hat{p}_i^t = p_i^{t_m} \in \mathcal{P}_i \text{ if } t_m < t \leq t_{m+1}$$

Similarly, the most recent query issued by o_i before time t, $t \geq t_0$, is $\hat{q}_i^t$:

$$\hat{q}_i^t = q_i^{t_m} \in \mathcal{Q}_i \text{ if } t_m < t \leq t_{m+1}$$

This notation helps to identify which updates and queries must be processed during any tick, since later on (Section 1.1.2) we assume that the processing of a query can be delayed up to a certain extent; this is done in order to optimize the overall system throughput while processing queries using the most up-to-date information available. At this point we can formally define the result set of a range query and the result set of a k-NN query.

Definition 3. *[Result set of a range query]*
The result of a range query q_i^t when computed at time t', $t_0 \leq t \leq t'$, is denoted by $res(q_i^t, t')$ and is defined as follows:

$$res(q_i^t, t') = \{o_j \in O \mid \hat{p}_j^{t'} \in_s q_i^t \wedge i \neq j\},$$

where $\hat{p}_j^{t'} \in_s q_i^t$ denotes that $\hat{p}_j^{t'} = (x, y)$ belongs to the query rectangle q_i^t, i.e., $x^a \leq x \leq x^b$ and $y^a \leq y \leq y^b$.

Assuming that updates $u_1, \ldots, u_3$ in Figure 1.1 are the most recent ones before t', we have $res(q_3^t, t') = \{2\}$.

Definition 4. *[Result set of a k-NN query]*
Let $pos_i(t)$ be the center of a k-NN query issued by o_i at time t. Let also $B_i = (pos_i(t), \Delta)$ be the ball centered on $pos_i(t)$ having radius Δ.

Then, the result set of a k-NN query q_i^t computed at time t', $t_0 \leq t \leq t'$ is denoted by $res(q_i^t, t')$ and is defined as follows:

$$res(q_i^t, t') = \{o_j \in O \mid \hat{p}_j^{t'} \in_r q_i^t \wedge i \neq j\},$$

where $\hat{p}_j^{t'} \in_r q_i^t$ denotes that $\hat{p}_j^{t'} = (x, y)$ belongs to the ball $B_i = (pos_i(t), \Delta)$ such that Δ makes the cardinality of the result set being equal to

$$|res(q_i^t, t')| = k.$$

Assuming that updates $\{u_2, u_3, u_5\}$ in Figure 1.2 are the most recent ones before t', we have $res(q_1^t, t') = \{2, 3\}$.

1.1.2 Batch Processing

To obtain high throughput when facing massive workloads due to frequent updates and queries issued by huge populations of moving objects, we quantize time into *ticks* (time intervals) with the objective of processing updates and queries in batches on a per-tick basis. Assuming that the initial time is 0 and the tick duration is Δt, the k-th time tick τ_k is the time interval $[k \cdot \Delta t, \quad (k+1) \cdot \Delta t)$. Specifically, we aim to collect object position updates and queries that arrive during a tick, and process them at the end of the tick. If an object submits more than one update and query during a time tick, only the most recent ones are processed.

Let $P^{\tau_k} = \{\hat{p}_1^{(k+1)\cdot\Delta t}, \ldots, \hat{p}_n^{(k+1)\cdot\Delta t}\}$ be the last known positions of all objects at the beginning of the $(k+1)$-th tick, and $Q^{\tau_k} = \{q_1^{\tau_k}, \ldots, q_n^{\tau_k}\}$ be the most recent queries issued during the k-th tick, where:

$$q_i^{\tau_k} = \begin{cases} \hat{q}_i^{(k+1)\cdot\Delta t} & \text{If object } o_i \text{ issues any query during the } k\text{-th tick.} \\ \perp & \text{Otherwise.} \end{cases}$$

Note that if object o_i does not issue any query during the tick, then $q_i^{\tau_k} = \perp$.

Figure 1.3 captures the temporal aspects of the previous example. The timeline is partitioned into ticks τ_1, τ_2, ... of duration Δt. Objects issue updates and queries

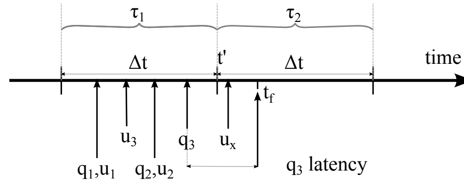


Figure 1.3: Timeline

independently and asynchronously. For example object o_3 sends a query and an update separately. Incoming updates and queries are batched based on the ticks. For example, update u_x belongs to the batch of τ_2. The batches are processed at the beginning of the next tick. Thus, at time t' (at the beginning of τ_2) we start processing all updates and queries arrived during τ_1. We complete the processing of the batch, thus making available the query results, at time t_f, hopefully before the end of τ_2.

1.1.3 Query Semantics

The procedure for computing queries as described above ensures serializable query processing and implements the timeslice query semantics, where query results are consistent with the database state at a given time, usually when we start processing

the query. This is a popular choice in traditional databases and is commonly adopted in related work [8, 9, 10, 11].

For example, in Figure 1.3 the computation results are returned at time t_f for the first batch. Since the object update u_x occurs at time t_u, $t_u > t'$, i.e., after we start processing q_3, u_x is not considered even if it arrives before the results are returned. In this way, the query result is consistent with the database state at time t', when we start processing query q_3.

1.1.4 Quality of Service - Query Latency

On the one hand, the processing of updates and queries on large batches can be expected to improve system throughput. On the other hand, we assume that some applications, e.g., MMOG applications, are sensitive to the delays with which query results are returned. Thus, it is important to be able to assess the *latency* of query processing, which is affected by the number of queries and updates that arrive during a tick, the tick duration, and the computational capabilities of the system.

Definition 5. *[Latency, Queueing, Computation Time]*
Assume that the processing of query q_i^t, issued at time t, starts at time t' and completes at time t_f. We define the following durations: $latency_time(q_i^t) = t_f - t$, $queueing_time(q_i^t) = t' - t$, and $processing_time(q_i^t) = t_f - t'$, so that $latency_time(q_i^t) = queueing_time(q_i^t) + processing_time(q_i^t)$.

We can now generalize the concept of latency to all the queries issued during a tick, all executed in batch at the beginning of the next tick.

Definition 6. *[Tick Latency]*
The latency of the queries Q^{τ_k} arrived during the k-th tick is defined as follows:

$$Tick_Latency(Q^{\tau_k}) = \max_{i \in \{1,\dots,n\},\ \ q_i^{\tau_k} \neq \perp} latency_time(q_i^{\tau_k})$$

Given an application-dependent *maximum latency threshold* λ, a system satisfies the application's *QoS Latency Requirement* if, for each tick k, $Tick_Latency(Q^{\tau_k}) \leq \lambda$ holds.

The tick duration Δt should be chosen such that even queries issued at the beginning of a tick are answered within time duration λ. Since query processing is delayed till the beginning of the next tick, the *worst-case latency* for a query q_i^t takes place when q_i^t is issued at the beginning of a tick: in this case, the latency is the sum of $\Delta t = queueing_time(q_i^t)$ and $processing_time(q_i^t)$. The following lemma states a simple, sufficient criteria to select Δt or to verify whether a given execution time satisfies the latency requirement.

Lemma 1. *Let Δt_{exe}^k be the time to process all queries in the k-th batch. Given a tick duration Δt and a latency requirement λ, then if $\Delta t + \Delta t_{exe}^k \leq \lambda$, the execution*

satisfies the latency requirements, i.e., Tick_Latency(Q^{τ_k}) $\leq \lambda$. From the above we can derive the following sufficient condition for Tick_Latency(Q^{τ_k}) $\leq \lambda$:

$$\Delta t > \lambda - \Delta t \geq \Delta t^k_{exe} \tag{1.1}$$

Above, we have that $\Delta t > \lambda - \Delta t$ because the processing of the queries accumulated during a tick is assumed to be completed before the end of the next tick.

The computational capabilities of a system influences the choice of the tick duration and the fulfillment of the latency requirement in Lemma 1.

Lemma 2. *Let β be the system bandwidth, expressed in terms of the number of queries processed per time unit, and let Q_{max} be the maximum number of queries that can occur during a tick. Then a sufficient condition for the system to meet the QoS latency requirement (based on threshold λ) is:*

$$\beta \geq \frac{Q_{max}}{\lambda - \Delta t} \tag{1.2}$$

Proof. According to Equation 1.1, in order to respect the timeliness, we have to process all queries in a time Δt^k_{exe} such that $\Delta t^k_{exe} \leq \lambda - \Delta t < \Delta t$. Given the bandwidth β, the maximum execution time to process all queries of a tick is $\frac{Q_{max}}{\beta}$. Hence, $\frac{Q_{max}}{\beta} \leq \lambda - \Delta t$ must hold, from which the lemma follows. □

For a given latency requirement λ, if we increase the tick duration Δt, this increases Q_{max} and decreases $\lambda - \Delta t$. So, if we increase Δt, in order to satisfy Equation 1.2, we have to compute more queries in less time, and thus it may happen that bandwidth β becomes insufficient to support the requested workload respecting the given latency threshold, i.e., $\beta < \frac{Q_{max}}{\lambda - \Delta t}$.

1.1.5 Problem Statement

We can finally state the problem of computing repeated range or k-NN queries over massive streams of moving objects observations, by discretizing the time in intervals (ticks), synchronizing query processing according to these ticks, and iteratively computing queries in batch mode.

Given *(i)* a set of n objects O, *(ii)* a partitioning of the time domain into ticks $[\tau_k]_{k \in \mathbb{N}}$ of duration Δt, *(iii)* a query latency requirement λ, and *(iv)* a sequence of pairs $[(P_{\tau_k}, Q_{\tau_k})]_{k \in \mathbb{N}}$, where P_{τ_k} is the up-to-date object positions at the end of τ_k, and Q_{τ_k} is the set of the last issued queries during τ_k, we have that the **iterated batch processing** of queries Q_{τ_k} over the corresponding P_{τ_k}, $k \in \mathbb{N}$, **yields** $[R_{\tau_k}]_{k \in \mathbb{N}}$, i.e., a **sequence of pairs**, each composed of a query and the list of the corresponding results:

$$R_{\tau_k} = \{(q_i^{\tau_k},\ res(q_i^{\tau_k},\ (k+1) \cdot \Delta t))\ \mid\ q_i^{\tau_k} \neq \perp\ \wedge\ q_i^{\tau_k} \in Q_{\tau_k}\}.$$

The **processing time** of each batch of queries Q_{τ_k} must be **upperbounded** as follows, to satisfy the query latency requirement λ:

$$processing_time(Q_{\tau_k}) \leq (\lambda - \Delta t) < \Delta t$$

1.2 Graphics Processing Units

GPUs are based on massively parallel computing architectures that feature thousands of *cores* grouped in *streaming multiprocessors* [1] (hereinafter denoted by SMs for brevity) coupled with several gigabytes of high-bandwidth RAM. In recent years these devices sparked a consistent interest due to their ability in performing general purpose computations, thus offering possible substantial performance gains when compared to the performance of traditional CPUs.

Due to the architecture of these devices, exploiting effectively their computational power is far from trivial. Specifically, each GPU processing core is slower than a typical CPU and has limitations on its access to device memory, resulting in potential contentions unless specific conditions are satisfied [12]. Moreover, GPU cores have to coordinate their actions, which is usually a complex issue considered their architectural organization.

Proper algorithms, designed with the architectures of the GPUs in mind, are needed in order to maximize the performances and obtain significant gains with respect to CPU-based algorithms, a goal which is not always possible to pursue [13] depending on the characteristics of the targeted problem.

In order to exploit effectively the computational power of a GPU, memory accesses should generally have high spatial and temporal locality. In addition, we have to ensure that all the cores of an SM profit from memory block transfers, by forcing coalescing of parallel data transfers: this avoids serial memory accesses and consequent performance degradation due to a wrong usage of memory hierarchies.

Moreover, the GPUs feature several types of memories ranging from private thread registers and fast shared memory, which are both shared among the core groups of each SM, to global memory, which has a lower throughput but it is of significant size and represents the contact point with the CPU host. To achieve consistent performances a programmer has to be aware of this complex memory hierarchy by orchestrating and managing explicitly memory transfers between different memories.

Workload partitioning is paramount when designing GPU algorithms since unbalances may create inactivity bubbles across the streaming multiprocessors and seriously cripple the performance.

A GPU consists of an array of n_{SM} multithreaded SMs, each with n_{core} cores, yielding a total number of $n_{SM} \cdot n_{core}$ cores. Each SM is able to run *blocks* of *threads*,

[1] We use the NVIDIA CUDA terminology, throughout the thesis, to refer GPUs architectural features and peculiarities, as well as to describe software targeted to GPUs, since CUDA represents the dominant framework in the context of general purpose computing on GPUs.

namely *data-parallel tasks*, with the threads in a block running concurrently on the cores of an SM. Since a block typically has many more threads than the cores available in a single SM, only a subset of the threads, called *warp*, can run in parallel at a given time instant. Each warp consists of sz_{warp} *synchronous, data parallel threads*, executed by an SM according to a SIMD paradigm [12, 14]. Due to this behavior, it is important to avoid branching inside the same block of threads. It is worth remarking that at warp level no synchronization mechanisms are needed to guarantee data dependencies among threads, thanks to the underlying scheduling. Finally, a function designed to be executed on GPU is called *kernel*.

1.2.1 Main algorithmic design issues

Considering the specificities of the problem described in Section 1.1.5, five main design issues shall drive the design of the hybrid CPU/GPU pipeline in charge of the query processing. First, we have to find a proper way to distribute the workload evenly among the GPU streaming multiprocessors, since unbalances typically create inactivity bubbles. Second, we need to avoid contention/serialization when accessing the GPU device memory, in order to favour spatial locality, thus properly taking advantage from the complex GPU memory hierarcies. Third, in the *range queries* case we should compress the data the GPU has to send back to the CPU during the processing, since the output is typically much larger than the input. Fourth, expensive synchronization mechanisms among concurrent threads should be avoided, since these are typically very costly in terms of performance. Finally, for each pipeline task executed on the GPU the unit of parallelization (either objects or queries, or partitions of queries) should be carefully chosen according to task specificities.

1.3 Related work

In this section we provide an overview about related works on the scenarios and problem setting considered, with a particular focus on works targeting equivalent or very similar problems by means of hybrid CPU/GPU approaches.

1.3.1 Processing repeated range queries over massive moving objects observations

The idea of using the abundant and cheap computational power offered by GPUs in order to boost spatial joins computations dates back to the era when GPUs did not offer real general purpose computing capabilities and the use of OpenGL or DirectX APIs were needed in order to have access to their resources [15]. The potential of GPUs was clearly understood, but the scenarios, the problems, and the approaches considered at that time were quite different from those covered in this work.

As pointed out in an extensive review [16], the need for managing continuously incoming and evolving spatial data can be addressed by using simple, light-weight and, in many cases, throwaway data structures. However, it is crucial that data structures and algorithms contend effectively with skewed data and avoid redundant spatial joins and bad workload distributions as much as possible. In such review the authors claims Synchronous Traversal to be the top performer across several datasets. However, when considering its multi-threaded variant the speedup yielded by Synchronous Traversal (up to 6× with 12 cores) does not follow a linear behaviour as the amount of cores increases, due to inter-thread dependencies and challenges related to finding a proper way to partition the workload among the cores. Indeed, these are serious challenges which we try to tackle in the present work.

Recent studies [10, 17] show how uniform grid-based solutions are particularly attractive when managing continuously incoming and evolving spatial data in main-memory multi-core settings. Even if these works do not consider the architectural peculiarities and limitations of the GPUs, they nonetheless highlight how regular grids, in general, represent a natural basis for GPU parallelization strategies thanks to their structural regularity [18].

Other works consider the problem of building R-Trees (and possible derivations) from scratch [16], even recurring to hybrid approaches based on the combined use of CPU and GPU for range queries computation [19, 20]. While the goals of some of these works are different with respect from the ones of the present work, it is interesting to notice how solving certain problems is particularly recurrent and challenging when processing massive spatial data by using massively parallel architectures, i.e., (i) finding a solution able to distribute the workload in the most uniform way (depending also on the spatial data distribution), (ii) arranging spatial data by using proper GPU-friendly light-weight regular data structures which allow to use the GPUs features effectively, and (iii) exploiting spatial locality as much as possible.

In a recent work [21] in the context of collision detection in computer graphics, the the author focuses on extremely fast and efficient GPU-based construction and lookup algorithms for binary radix trees when performing real-time collision detection between 3D objects (thus addressing a similar problem with respect to the one addressed in this work). While the algorithms proposed are able to handle elegantly the skewness possibly characterizing the data, the work doesn't consider the problems of detecting and having to write out huge amounts of results in very short time intervals. The first problem increases remarkably the amount of lookups and traversals in the trees needed to compute a query, while the second problem entails serious issues mainly related to memory throughput maximization and how to avoid memory access contention.

Previous works [18] already pointed out the advantages of using point-region quadtrees for partitioning a low dimensional space when using the GPUs, thanks to the direct relationship between the quadtrees structural properties and the Morton codes [22, Ch. 2][23]. Indeed, quadtrees fit extremely well the GPUs architectural features, hence allowing to devise fast and efficient algorithms.

We are unaware of existing studies tackling the problem of repeatedly computing sets of range queries over continuously moving objects by means of an hybrid CPU/GPU approach. The most closely related work is focused on point-in-polygon joins [24]. This work considers scenarios characterized by massive amounts (possibly millions) of static entities, represented by points, and sets of polygons (in the order of few thousands) potentially covering the entities: the goal is to speed up the point-in-polygon tests by exploiting the computational power of GPUs using a novel approach stemming from the traditional *filtering and refinement* schema. This work is similar to the present work in that it exploits point-region quadtrees in order to index the points, and thus improve the workload distribution when determining which point-in-polygon tests have to be computed. However, relevant differences separate the two works: (i) the entities are static, (ii) joins are computed between huge amounts of points and limited sets of polygons (instead of huge sets of range queries issued by the same entities) and (iii) polygons are indexed (through their bounding boxes) by means of a uniform grid and subsequently paired (for the final refinement phase) with sets of potentially overlapped points. This is in turn achieved by indexing each quadtree quadrant minimum bounding rectangle (enclosing the quadrant points) by means of the same uniform grid. In light of this, we deem that a comparison with our proposals would be not interesting, since the other work tackles different scenarios and consequently does not consider a set of relevant issues having far-reaching consequences, above all the issues related to the continuous management of huge sets of queries and results.

1.3.2 Processing repeated k-NN queries over massive moving objects observations

The problem of computing a given set of k nearest neighbours (k-NN) queries over a given set of points in some d-dimensional space is a well-renowned problem found in many theoretical and practical fields. The general schema used to tackle this problem usually follows a two step approach: first, some *spatial index* is built over the set of points in order to reduce the amount of computations per query with a subsequent *recursive search* phase, which is performed over the index in order to compute the query results. In general, the effectiveness of a solution depends on the underlying spatial index and on the relaxation of some problem constraints related to the quality of results.

In the following we provide a brief overview about the main works in literature; moreover, since we are considering the aforementioned the problem in $\mathbb{R}^2$, in Section 1.3.2.2 we exclusively focus on those works tackling the problem of computing k-NN queries in low dimensional spaces by means of an hybrid CPU/GPU approach.

1.3.2.1 General overview.

The literature can be substantially divided into two macro-families. The first one tackles the problem when processing k-NN queries with data in low dimensional spaces, while the second family tries to solve the same problem with high dimensional spaces.

For what relates to the first family - which also covers the scenario and the problem setting introduced in this work - the vast majority of approaches are based on the usage of kd-trees [25, 26], as shown in extensive surveys such as [27, Ch. 63] or [28, Sec. 5]. Other minor solutions are based on R-Trees [29].

The second family focuses on high dimensional data (e.g., images, documents, feature vectors describing some kind of entities, etc.). High dimensional data poses serious problems when designing or choosing spatial indices due to the so-called *curse of dimensionality*, which negatively affects solutions tailored for low dimensional spaces in terms of time and space complexity (on this matter the reader may refer, for instance, to [30]). In order to tackle effectively such problem many approaches recur to the use of *random projections*, exploiting the intuition for which, in many cases, high-dimensional data tend to form clusters in subspaces having much lower dimensionality than the original space. Also, such approaches trade inaccuracy (for what relates to queries results) for performance - at least up to some extent. Among these works, worth of mention are solutions based on locality sensitive hash (LSH) functions. Since this interesting branch of research goes out of the scope of the present work, the reader may refer to the following references: [31, Ch.3], [32, 30, 33, 34].

1.3.2.2 Computing k-NN queries on GPU.

To date, the first work tackling the problem of computing k-NN queries by means of an hybrid CPU/GPU approach is [35]. In this work the authors propose a brute-force quadratic approach: first, for each k-NN query all the distances between its center and the dataset points are computed. Subsequently, distances are sorted in ascending order so that the first k ones represent the final query result set. This approach is quite simple and straightforward, yet it fits quite well the GPUs architectures and proves to be quite effective with small/medium sized datasets, especially when processing data in spaces having moderate to high dimensionality.

Focusing specifically on works related to low dimensional spaces, many of these actually tackle the problem of computing k-NN queries in static scenarios and when the problem is part of a bigger problem. Indeed, computing k-NN queries is strikingly recurrent in many fields, such as computer graphics, physics, astronomy, etc. (see for instance [36, 37, 38, 39]). Such works almost always rely on a kd-tree based index whose construction usually happens on CPU. The index is then subsequently used to perform the k-NN search, where the tree has to be navigated for each query: such navigation is usually performed on CPU, depending on the specific approach,

while distance computations - representing the compute-intensive part of the whole processing - always happen on GPU.

All these solutions typically exhibit a bottleneck, usually the fact that at least one of the core operations is performed on CPU whereas parallelizing these operations on GPU would speed up the entire processing remarkably. Operations which typically represent a bottleneck are the index construction or the index navigation (when computing the queries). Indeed, one of the most difficult problems in GPGPU computing, apart from devising proper algorithms and data structures, is to decide whether it is more profitable to execute a given task on CPU or GPU. Such decisions usually have far-reaching consequences on the design of processing pipelines and in many cases it translates into finding delicate and appropriate trade-offs. Another observation about the aforementioned works is that a consistent part of them are tailored for very specific scenarios and therefore find very little use in other contexts.

A recent work [40] targets spaces having low to moderate dimensionality (i.e., $\mathbb{R}^{4 \leq d \leq 20}$). This work is quite interesting since it exploits the usage of queue-based buffers to accumulate and distribute on the fly fairly uniform workloads across GPU streaming multiprocessors. The schema adopted is the following: first, a kd-tree is built over the set of points (this is done on CPU). Then, a set of queue-based buffers is associated with the set of kd-tree leaves, one per leaf: the idea is to associate queries with leaves through such buffers on the fly. Indeed, for each query we navigate the kd-tree by means of a recursive tree traversal (done on CPU), propagating queries across leaves (through the buffers) containing potential nearest neighbours. Once enough queries are accumulated inside each buffer (the same query may be possibly replicated in multiple buffers), the buffers content is flushed out and the GPU kicks in: for each leaf, the GPU computes the distances between the queries and the objects associated with it. This iterative process goes on until there is at least one query which requires to visit a tree leaf.

In general, none of the works mentioned above consider scenarios where massive amounts of moving objects repeatedly issue massive amounts of k-NN queries, and such scenarios must be handled in real-time while satisfying a given set of quality of service constraints.

2

GPU-Based processing of repeated range queries

In this chapter we tackle the problem of processing repeated *range* queries over massive moving objects observations by means of an hybrid CPU/GPU approach. To this end we present the *Quadtree* method (hereinafter referred as QUAD).

The key idea behind QUAD is to partition the problem space using a *point-region quadtree* inducing a regular grid; in turn, the cells of the grid represent a set of independently solvable problems, each one associated with a *data-parallel task* runnable on a GPU streaming multiprocessor, also capturing and adapting to possibly skewed spatial distributions.

This strategy allows us to obtain a quasi uniform distribution of the workloads among coarse-grained tasks – each task corresponding to a single cell of the index – with the aim of improving the overall efficiency of the system and maximizing the performance. In order to demonstrate the importance of this aspect, we also introduce a baseline spatial index, whose space decomposition relies on the usage of a *uniform grid*. We call this method *Uniform Grid* (hereinafter referred as UG).

Both QUAD and UG preprocess in parallel the data and store consecutively object and queries falling in the same grid cell, thus optimizing memory accesses. Further, to avoid the use of blocking writes and to ensure high throughput, both methods compute the query results by means of a two phase-approach using a particular *bitmap* intermediate representation [5].

To the best of our knowledge this is the first work that exploits the GPUs to efficiently solve repeated range queries on continuously moving objects, having care to tackle effectively skewed spatial distributions as well. There are few existing works that use the GPUs for spatial query processing, but they consider substantially different problems, as detailed in Section 1.3.1.

The main contributions can be summarized as follows:

- we define a hybrid CPU-GPU pipeline to process batches of range queries, which effectively exploit the GPU computational power while taking care of its architectural features and limitations. Thanks to its flexibility, the pipeline can be adapted to different spatial indices as well.

- we introduce a set data structures which allow the pipeline to perform operations that concurrently write interlaced lists of results, using coalesced memory accesses that avoid race conditions.

- while we adopt the usual query splitting approach to make the data-parallel tasks induced by the indices completely independent, we take advantage of sub-query areas that *completely cover* index cells to *save work* during the query processing (in terms of amounts of containment tests performed), and *compress* the information the GPU has to send back to the CPU when notifying the query results.

- we carry out an extensive set of experiments in order to compare QUAD against UG. The structural regularity and simplicity characterizing the uniform grids, onto which UG relies, is a well-known feature that fits very well the GPUs characteristics, but have structural limitations - mainly related to the inability to fully capture the skewness possibly characterizing the spatial data - which may seriously hinder the performances.

  On the other hand, QUAD is able to automatically adapt to very skewed spatial object distributions, assuring very good performances for a broad range of spatial distributions. We demonstrate this claim by comparing QUAD against UG. We also compare QUAD against the state-of-the-art (for what relates to the problem considered) sequential CPU algorithm, namely the *Synchronous Traversal* algorithm [16, 41].

The chapter is structured as follows: in Section 2.1 we present the spatial indices used by QUAD and UG to partition the workload, while in Section 2.2 we extensively detail the pipeline as well as the customizations needed by QUAD and UG. Finally, Sections 2.3 and 2.4 present an extensive set of experimental studies which show (i) the benefits coming from the usage of the proposed range query processing pipeline and data-structures, (ii) how QUAD spatial indexing is better with respect to the one used by UG and (iii) how QUAD outperforms the state-of-the-art sequential CPU competitor as well as outperform, or being on par with, UG.

We delegate the conclusions and possible directions of research to the conclusive chapter of the thesis.

2.1 Spatial indexing and data structures

In this section we discuss the inspiring principles behind QUAD and UG, along with the architectural features of GPUs that impact on the spatial indices and data structures design.

When processing repeated range queries, the same procedure is repeated for each tick. Thus, for the sake of readability, hereinafter we omit the subscript that

indicates the tick, and denote by P, Q, and R, respectively, the up-to-date object positions, the non-obsolete queries, and the result set associated with a generic tick.

2.1.1 Design considerations

A brute-force approach for computing repeated range queries entails $O(|P| \cdot |Q|)$ containment checks per tick. By using spatial indices it is possible to prune out consistent amounts of pairs of queries and object locations that do not intersect. However, when choosing or designing an appropriate index, we have to consider its pruning power along with its maintenance costs. For example, regular grid indices are generally reported to have low maintenance costs, and thus are suitable for update-intensive settings [17]. Another aspect is the number of cores and the memory hierarchy provided by the underlying computing platform. Given the same workload, different indices may be the best option for different platforms. With massively parallel platforms such as GPUs, the regularity characterizing spatial indices based on regular grids is attractive, as it enables fast and efficient parallel index updating and querying. Even if tree-based spatial indices are able to distribute objects evenly among the index cells (the tree leaves), we have to avoid navigating the tree, since this may severely hinder efficiency due to poor data locality when accessing the memory.

In a previous work [5] we devised a simple uniform grid-based spatial indexing used to partition the workload and prune out useless containment tests. In that work we chose to determine the size of the index grid cells on the basis of the query size: the rationale was to reduce the amount of index cells to be considered when processing each query. However, solutions based on uniform grids generally cannot cope efficiently with skewed spatial distributions. To solve this issue we propose the QUAD method, which relies on a tree-based recursive spatial indexing, induced by point-region quadtrees. To ensure an unbiased comparison between QUAD and uniform grid-based spatial indices, we also introduce the UG method as a baseline. UG relies on a simple uniform grid-based spatial index, without any a-priori constraint on the size of the grid cells. Also, UG integrates all the optimizations conceived for QUAD (such optimizations are detailed in Section 2.2.6).

2.1.2 Overview of the methods

In the following we give an overview of both UG and QUAD.

UG materializes at each tick a uniform grid over the minimum bounding rectangle enclosing the object positions. The only way UG can cope with data skewness is by changing the coarseness of the grid, targeting a coarseness tradeoff on the basis of the object densities in crowded areas and loosely populated ones.

QUAD still yields at each tick an index over the same bounding rectangle, but the index cells are of varied size as QUAD is able to dynamically tune their size according to local object densities. To this end QUAD utilizes a point-region quadtree, which

entails a space partitioning that ensures a pretty balanced distributions of objects among the index cells even in presence of skewed data. Even if tree data structures are, in principle, difficult to manage on GPUs, the direct relationship between the quadtrees structural properties and the Morton codes [22, Ch. 2][23] open up to the possibility of implementing efficient massively parallel quadtree construction and lookup algorithms on GPUs[18].

Regardless of the index adopted, queries can be processed concurrently according to a *per-query parallelization*. More specifically, since both indices induce a partition of the space, where each disjoint space tile corresponds to a cell of either UG or QUAD, we virtually split queries according to the space partition, thus producing a *subquery* for each index cell a query intersects. Indeed, each subquery yields an *independent subtask*, which we process in parallel by only accessing the objects falling in the associated index cell. Note that this approach decreases the overall amount of containment checks, although the splitting yields more subtasks to process as the same query is processed several times, once for each relevant index cell.

It is worth pointing out that we can have subqueries whose areas entirely cover small index cells. This allows us to strongly optimize the computation: first, we can compress the output, since all the objects of these cells falls into the subquery areas; second, for the same reason we can avoid processing covering subqueries, thus saving computation time (see the *covering* queries optimization, Section 2.2.6).

Both UG and QUAD use an ad-hoc lock-free data structure based on bitmaps [5], to manage the result sets while they are produced on GPU. This design choice entails a further post-processing step needed to enumerate the final results contained in this data structure. To prove the merit of this choice we consider a more basic baseline, i.e., a variant of UG, namely the *Baseline Uniform Grid* ($\mathsf{UG}_{Baseline}$) method, which uses atomic operations to ensure the consistency of the result sets content.

2.1.3 Space partitioning and indexing

In the context of parallel query processing, there are two main reasons for partitioning and indexing the data according to a given space partitioning approach: the first one, also common to sequential query processing, is to avoid redundant computations and access to irrelevant data, while ensuring fast access to relevant information. The second reason, which is triggered by the ability to process data in parallel, is to ensure independent computations, avoid redundant work and balance the workload among the processing unit cores.

In the following we introduce the two space partitioning methods onto which UG and QUAD rely. Both methods aim to adaptively partition the Minimum Bounding Rectangle (MBR) containing all the object positions during any tick. We denote this MBR by $\mathcal{G} = (x_a^{\mathcal{G}}, y_a^{\mathcal{G}}, x_b^{\mathcal{G}}, y_b^{\mathcal{G}})$, where $(x_a^{\mathcal{G}}, y_a^{\mathcal{G}})$ and $(x_b^{\mathcal{G}}, y_b^{\mathcal{G}})$ represent the lower-left and the upper-right corners of the MBR. Aside from the specific ways through which the methods define the geometries and enumerate grid cells, both of them assign queries and object locations according to the same mapping functions (see Section 2.1.3.3).

2.1.3.1 Uniform grid-based partitioning

UG partitions the space by superimposing a uniform grid $\mathcal{C}$, whose cells are of equal size, over $\mathcal{G}$.

Definition 7. *[MBR partitioning into a uniform grid]*
$\mathcal{G}$ is partitioned according to a uniform grid $\mathcal{C}$ of $N \cdot M$ cells of width W and height H such that the cell c_{ij} covers the following region:

$$(x_a^{\mathcal{G}} + i \cdot W, \quad x_a^{\mathcal{G}} + (i+1) \cdot W, \quad y_a^{\mathcal{G}} + j \cdot H, \quad y_a^{\mathcal{G}} + (j+1) \cdot H).$$

To ensure that the grid covers $\mathcal{G}$, constants N, M, W, and H are chosen so that $x_a^{\mathcal{G}} + N \cdot W \geq x_b^{\mathcal{G}}$ and $y_a^{\mathcal{G}} + M \cdot H \geq y_b^{\mathcal{G}}$ hold. We associate with each cell $c \in \mathcal{C}$ an integer ID, which enforces a total order among the index cells, by preserving spatial locality.

2.1.3.2 Quadtree-based partitioning

In the *quadtree based partitioning* case, $\mathcal{G}$ is covered by a quadtree-induced grid $\mathcal{C}$, determined on the basis of the local densities of moving objects. In this case $\mathcal{G}$ is therefore partitioned into a set of cells corresponding to the quadtree leaves.

Definition 8. *[MBR partitioning into a quadtree-induced regular grid]*
$\mathcal{G}$ is partitioned into a set of variably sized cells belonging to grid $\mathcal{C}$, induced by a point-region quadtree. Given a constant th_{quad}, denoting the maximum amount of objects allowed inside a single quadrant/cell of the final grid, we have that each cell of $\mathcal{C}$ corresponds to a quadtree leaf, and contains an amount of object not greater than th_{quad}. We associate with each cell $c \in \mathcal{C}$ an integer ID, which enforces a total order among the index cells, by preserving spatial locality.

2.1.3.3 Mapping of moving objects and queries to space partitions

Given an index $\mathcal{C}$ derived by QUAD or UG, we assign objects and queries to the index cells. Since the area of any query can intersect several cells of $\mathcal{C}$, this entails a partition of the area. We call this operation **query splitting**, which potentially yields a set of *subqueries* for each query. Finally, each subquery can be univocally assigned to a single index cell.

Definition 9. *[Mapping functions for object locations and subqueries]*
Given the set of cells of a grid $\mathcal{C}$, we have two ***mapping functions*** *$f : P \rightarrow \mathcal{C}$ and $g : Q \rightarrow 2^{\mathcal{C}}$* ***map****. Function f maps each object location $p \in P$ to the cell $f(p)$ that contains p. Function g maps each query $q \in Q$ to a set of cells $g(q)$, whose intersection with q is not empty. We use the term* ***subqueries*** *to denote the restrictions of a query q to each of these cells. Moreover, we call the operation performed by g* ***query splitting****. Finally, each subquery is classified as* ***intersecting*** *or* ***covering****, according to the fact that it partially/entirely covers the associated cell.*

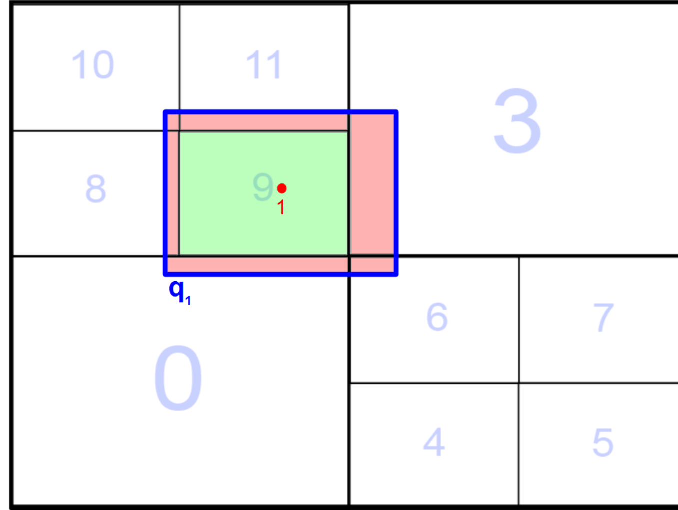


Figure 2.1: Simple mapping example over a quadtree-induced grid.

A simple example about how f and g operate is given in Figure 2.1: f maps object 1 to the cell having ID 9, while g splits query q_1 (issued by object 1) over 7 different cells. Among these, six are *intersecting* ones (highlighted in pink, namely the subqueries intersecting cells with IDs 0, 3, 6, 8, 10, and 11) while one is a *covering* subquery (highlighted in green, covering the cell with ID 9).

2.1.4 Data structures

As it will be pointed out in in Section 2.2, both QUAD and UG rely on a hybrid CPU/GPU processing pipeline, a pattern quite common in the context of General Purpose Computing on GPUs [42, 43]. Each stage of the pipeline performs a set of transformations on the data in order to produce a final output. To this end, the design of data structures should (i) allow data to be concurrently accessed with minimal use of atomic operations or barriers, thus avoiding locking related penalties; (ii) permit the use of coalesced memory accesses, in order to maximize the memory throughput; (iii) exploit spatial locality, whenever possible, in order to maximize the benefits deriving from coalescing and caching. In the following subsections we introduce the relevant data structures used by our approach.

2.1.4.1 Moving objects and queries data structures and their layout

Given a set of n-tuples representing a class of entities (in our context an object location or a query), the tuples elements are logically arranged by means of a *structure of vectors* (also known as *structure of streams* or *structure of arrays*) layout [44, Ch.33]. This layout groups a set of n vectors, each one representing a single element

of the tuples, and aligns the vectors elements with respect to the entities they are associated with. An example of such arrangement is given in Figure 2.2.

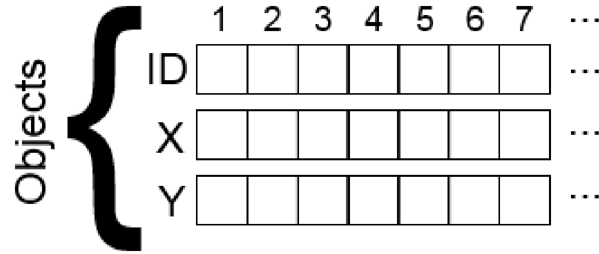


Figure 2.2: Example of a structure of vectors used on a set of objects described by 3-tuples.

The main benefits of this representation derive from the observation that it does not require complex pointer arithmetic, and it naturally makes possible to exploit coalesced memory accesses. Moreover, it is a representation commonly used in well established GPU algorithms, thus allowing an efficient interplay (and code reuse) between the operations making up the processing pipeline.

Consider that we aim at generating *independent tasks*, each one associated with a *subquery* and a specific *active cell*, i.e., a cell with at least one object and one subquery, where each task processes a single subquery over the objects of the associated cells (see Sections 2.2.4 and 2.2.5 for more details). In light of this, it is convenient to properly arrange the structures of vectors associated with objects and subqueries (each set has its own structure) in order to exploit data spatial locality and boost memory throughput. To this end, we have to arrange entities falling inside the same grid cell in contiguous memory locations (memory blocks). Therefore, first object locations and subqueries are sorted by the IDs of the associated index cells. As a side-note, we observe that the same originating query can be stored in several memory blocks, since function g potentially yields multiple intersecting cells for each query. Second, block boundaries are stored in a table, by distinguishing between the sub-blocks storing object locations and sub-blocks storing subqueries. This allow us to directly access the data belonging to any cell. The reader may refer to Section 2.2.3 for more details about the sorting operations performed in order to achieve such arrangements.

2.1.4.2 Intermediate bitmap representations of query result set

One of the main issues is related to the efficient collection of possibly huge sets of query results. According to the problem statement given in Section 1.1.5, the result of a single tick is described in terms of a set of pairs, each one consisting of an identifier associated with the object issuing a query and a set of identifiers related to the objects falling inside the query result set. Since query results are

produced concurrently, contentions when writing them out could seriously cripple massive parallelism.

To avoid this issue we exploit a two-phase approach relying on two intermediate data structures, based on a *bitmap* layout, in order to eliminate the need of threads synchronizing mechanisms while maximizing the overall memory throughput and minimize the amount of space used to store intermediate results on GPU. Since these data structures are strongly tied to the design of the algorithms in charge of the aforementioned operations, we postpone their description to Sections 2.2.4 and 2.2.5.

2.2 Query processing pipeline

The core computation to process each set of range queries can be surely ascribed to the containment tests between objects locations and query areas. Considering the potential huge amount of containment tests and results each tick can yield, this apparently simple and straightforward operation is very expensive. In order to improve its efficiency we embed this computation in a pipeline of concatenated operations. The various stages of the pipeline prepare the spatial index for improving the efficiency of the containment tests, compute the containment test outcomes in an intermediate format for efficiency reasons, and post-processes these results to produce the final query results.

In the following subsections we introduce the high-level pipeline, common to all methods. We also discuss the main design differences between UG, $\text{UG}_{Baseline}$ and QUAD in the implementation of each pipeline phase.

2.2.1 Pipeline description

The data entering the processing pipeline at the end of each tick are first processed to select the index parameters and build an empty index (phase 1, *index creation*). Then (phase 2, *moving object and query indexing*), objects and subqueries are mapped to index cells, and finally are sorted so that those contained in the same cell are stored in contiguous memory locations. The subsequent phase computes the containment tests between range queries and object locations (phase 3, *filtering with bitmap encoding*) producing an intermediate bit-encoded output which is structured to avoid contentions in memory access (issues *(ii)* and *(iv)* in Section 1.2.1). These intermediate results need a final post-processing phase to extract the final results (phase 4, *bitmap decoding*). Each phase takes advantage of a tight cooperation between the GPU and the CPU.

One of the key features of QUAD and UG is the ability to split the computation of each query among the space partitioning elements (cells) it intersects to reduce the total amount of containment tests. This entails the creation of a new set of *subqueries* originating from the query set Q. The distinction between UG ($\text{UG}_{Baseline}$)

and QUAD is related to the way they partition the space, that is, how a grid is materialized over the space and how objects locations and subqueries are mapped to grid cells. These aspects involve just the phases 1 and 2 of the pipeline, since the remaining ones directly use the cell identifiers associated with object locations and subqueries to determine which object locations and subqueries are relevant for a specific operation.

For this reason, in the following we describe phases 1 and 2 separately for UG (UG$_{Baseline}$) and QUAD, while the remaining ones can be described regardless of the involved spatial index.

2.2.2 Index creation and indexing in UG and UG$_{Baseline}$.

The performances of this method are significantly affected by the cells size used for $\mathcal{C}$. Choosing a suitable value is challenging since it depends on several factors, from the spatial distribution of data, to the opportunity of avoiding part of the computations thanks to optimizations that are triggered locally by grid and query based conditions (e.g., by exploiting the *covering* subqueries optimization described in Section 2.2.6).

In [5] we were able to optimally determine the cell size of a uniform grid index by assuming unrealistic uniform spatial distributions of objects.

Since the optimal granularity cannot be decided for any kind of dataset, but we need to still use uniform grid indexes as baselines for QUAD, we exploit an oracle to choose the grid coarseness for both UG and UG$_{Baseline}$. In practice, we determine the optimal grid coarseness parameter, for each tick and any kind of dataset, by performing parameter sweeping, and finally selecting the parameters that are the most favorable to UG and UG$_{Baseline}$ in each comparison. Accordingly, the goal of the index creation phase for the baselines UG and UG$_{Baseline}$ is simply the ad-hoc choice of the best grid granularity, in order to maximize the performance of the subsequent phases.

Index creation (UG/UG$_{Baseline}$). Since we already know the optimal grid cell size, the goal of the *index creation* phase in UG/UG$_{Baseline}$ is to determine the minimum rectangle $\mathcal{G}$ that bounds all the objects (*MBR*). The computation of the MBR is based on a GPU parallel reduction operation over the set of object positions and queries yielding the minimum and maximum coordinates.

Once $\mathcal{G}$ is set up, we use the cell size determined by the oracle to materialize an optimal uniform grid $\mathcal{C}$ over $\mathcal{G}$, so that objects and queries can be indexed accordingly.

Each cell of $\mathcal{C}$ is naturally associated with a pair (i, j), identifying the row and the columns of each cell. However, we adopt a transformation of (i, j) into a unidimensional identifier $CellID$, derived from (i, j) by interleaving the binary repre-

sentations of the two coordinates, thus obtaining the Morton code $z(i, j)$[1].

Moving objects and queries indexing (UG/UG$_{Baseline}$). Given an index $\mathcal{C}$, function f (Definition 9) maps a generic object location $p \in P$ to a cell $c \in \mathcal{C}$. In UG/UG$_{Baseline}$ the function consists of a simple algebraic expression that determines grid coordinates (which indeed correspond to a unidimensional Morton code identifying the cell) from object locations. This is implemented on the GPU by applying function f in parallel to all elements of P, thus obtaining a vector whose elements represent cell identifiers corresponding to each object location.

Still on the basis of index $\mathcal{C}$, function g (Definition 9) maps a generic range query $q \in Q$ to a set of cells in $\mathcal{C}$. The corners of each query q are mapped to grid coordinates, then a nested loop is used to enumerate the identifiers of cells intersected by the query. Since containment tests are superfluous for cells completely covered by q, the corresponding subqueries are marked as *covering* to enable the optimizations described in Sec.2.2.6.

In our GPU implementation of g, each query q is processed by a GPU thread that produces a set of triples *(queryID, cellID, coveringFlag)*[2] , each one representing an intersecting (covering) subquery. To avoid output write contentions without resorting to blocks and synchronization, a two-pass approach is adopted: the first dry-run pass determines the amount of triples per query, while the second pass writes out the triples to the correct positions in the output vector by exploiting the information created during the first pass. During the second pass, each subquery is also classified according to the intersecting/covering dichotomy.

The overall complexity of this phase is equal to $O(|P| + 2|Q| + |Q| + |\hat{Q}|) = O(|P| + 3|Q| + |\hat{Q}|)$: $|P|$ is due to the object locations indexing, $2|Q|$ is due to the two-pass approach, $|Q|$ is the cost to pay for the exclusive prefix sum performed between the first and the second pass needed to determine the subqueries locations in memory; finally, $|\hat{Q}|$ is related to the subqueries written out during the second pass.

Sorting (UG/UG$_{Baseline}$). Once object locations and subqueries are mapped to cells of $\mathcal{C}$, we sort them by the Morton codes of the cells, as illustrated in Figure 2.3. The goal is to store tuples mapped to the same index cell in contiguous memory locations, thus enhancing the *spatial locality* of each parallel block of threads working on subqueries and objects of a given active cell (i.e., a cell having at least one object) during the subsequent *filtering* and *decoding* phases (Sections 2.2.4 and 2.2.5 respectively).

[1] To this end, we adopt an optimized bitwise algorithm.

[2] In practical terms, the *coveringFlag* can be properly embedded inside the integer representing *cellID*.

Unsorted

ID	0	6	2	3	4	5	1
z	1	3	0	2	2	2	1

Sorted

ID	2	1	0	3	4	5	6
z	0	1	1	2	2	2	3

Figure 2.3: A simple example of a GPU-based sorting, based on the structure of vectors representation, of 8 entities according to their Morton codes. The discontinuities among the codes (thicker lines) determine the set of entities belonging to each cell.

Indeed, when sorting the subqueries we distinguish between covering and intersecting ones, in order to support the optimizations discussed in Section 2.2.6. In practice, we handle the covering queries in a different way, since the GPU does not need to process them: after the sorting operation, all intersecting subqueries, which need to be processed, are placed at the beginning of the subqueries structure of vectors.

Since the GPU sorting algorithm used throughout the pipeline will be the Radix Sort [43], the complexity of the sorting step is $O(b \cdot (|P| + |\hat{Q}|)) \approx O(|P| + |\hat{Q}|)$, where $\hat{Q}$ denotes the subqueries set.

2.2.3 Index Creation and Indexing in QUAD.

The key idea behind QUAD is to use a point-region (PR) quadtree as the backbone of its spatial index, exploiting the PR-quadtrees intrinsic ability to partition the space in differently sized parcels containing similar amounts of points.

Index creation (QUAD). The goal of this phase is to create a space partitioning $\mathcal{C}$ over $\mathcal{G}$, according to Definition 8, where each cell of $\mathcal{C}$ is a leaf PR-quadtree quadrant that does not contain more than th_{quad} objects. This property gives an upper bound to the containment tests computed by each GPU thread in charge of processing a query over all the objects falling in an index cell.

We observe that even if a space partitioning is determined according to local object densities for a particular tick, it can be often reused for consecutive ticks when the spatial distribution does not change significantly.

Therefore we compute the spatial quadtree partitioning during the first tick, and repeat this partitioning if the objects spatial distribution change significantly, since this event might potentially hinder the performances by increasing the overall amount of containment tests to be computed per query.

The construction of the quadtree proceeds top-down in an iterative manner, starting from the 4 equally sized quadrants that partition $\mathcal{G}$, and then splitting iteratively each quadrant containing more than th_{quad} objects. The whole procedure

Algorithm 1: GPU-based PR-quadtree construction

```
1  begin
2      V_P ← GPUcalculateMortonHash(V_P, I_A, l_max)
3      V_P ← GPUradixSort(V_P)

4      I_A ← {[0, |P| − 1]}
5      C ← ∅
6      l ← 1
7      repeat
8          I ← GPUdetectQuadrants(V_P, I_A, l, l_max)
9          (I_A, C) ← CPUcheckQuadrants(th_quad, I, l, l_max, C)
10         l_deep ← l
11         l ← l + 1
12     until (I_A ≠ ∅) ∧ (l ≤ l_max)

13     z_map ← GPUbuildZMap(C, l_deep)
```

is repeated level-wise, increasing the quadtree depth and splitting overpopulated quadrants if needed.

Algorithm 1 describes this iterative process. During the initial setup (lines 2 – 6), the function *GPUcalculateMortonHash* (line 2) computes the Morton codes z of all the objects stored in the structure of vectors V_p at the maximum quadtree level l_{max}. In practice, in this phase we consider a regular grid having $2^{l_{max}} \times 2^{l_{max}}$ cells. Morton codes z are computed in the same way as done in the $\mathsf{UG}/\mathsf{UG}_{Baseline}$ case, starting from the index (i, j) of the regular grid where each object falls into. Subsequently, V_P is reordered by *GPUradixSort* (line 3) according to the Morton codes z. Note that, given the z-code at the maximum quadtree level l_{max}, we can determine the quadrant index z' of any object at any level $l \leq l_{max}$ by simply truncating the binary representation of the Morton code z previously computed, which is equivalent to calculating $z' = \frac{z}{4^{l_{max}-l}}$. It is worth considering that the object order obtained by this sorting by z is invariant for any level $l \leq l_{max}$ of the quadtree. In other words, thanks to this sorting and the structural properties of quadtrees, objects contained in any quadtree leaf are memorized contiguously in V_P.

Subsequently, the algorithm initializes several variables: the set $\mathcal{C}$ of final leaves is initialized to $\emptyset$, the set I_A, containing the intervals of the quadrants to split, is initialized by inserting the interval related to the tree root, and, finally, the level l from which the iterative construction starts is set to $l = 1$ (lines 4 – 6).

Then, the algorithm iteratively builds (line 7) the quadtree level by level. *GPUdetectQuadrants* (line 8) identifies the starting and ending positions (i.e., the intervals) of the l-level quadtree quadrants related to the $(l-1)$-level quadrants added to I_A for splitting, and store such intervals in I. Then, *CPUcheckQuadrants* (line 9) determines which quadrants need further splitting at next level (their intervals are added to I_A) and which quadrants represent final leaves (their identifiers are added

to $\mathcal{C}$). The process ends whenever no more quadrants need to be split (i.e., I_A is empty) or the maximum possible quadtree level l_{max} is reached (line 12). In the latter case, all the quadrants found at level l_{max} are added to $\mathcal{C}$. We postpone the description of *GPUbuildZMap* (line 13) to a subsequent paragraph (see ***Indexing moving objects and queries (*QUAD*)***).

Functions *GPUcalculateMortonHash*, *GPUradixSort* and *GPUdetectQuadrants* are entirely implemented on GPU. On the other hand, *CPUcheckQuadrants* is executed on the CPU side, since the amount of quadtree quadrants created at each level are typically orders of magnitude lower than $|P|$.

Simple running example. Let us consider the example reported in Figure 2.4, where $l_{max} = 2$ and $th_{quad} = 1$. During the *Initialization* step, each object identified by an ID is associated with the Morton code z of the cell $c \in \mathcal{C}_{l_{max}}$, where $\mathcal{C}_{l_{max}}$ denotes a uniform grid, associated with the deepest possible quadtree level l_{max} (see the "Initialization" grid on the left of the figure). The pairs (ID, z) are stored in a table (see the "Unsorted" table). Subsequently, pairs are sorted according to the second elements, i.e., the Morton codes (see the "Sorted" table). Next, the algorithm proceeds building the quadtree, starting from *Level 1*, creating iteratively new levels, until at least one quadrant requires to be split ($I_A \neq \emptyset$) or l_{max} is reached.

At each level, the algorithm associates each object with a quadtree quadrant belonging to the currently considered level by computing the corresponding quadrant indices z'. In this regard see the "Iterations" table in Figure 2.4, where each row (after the second one) corresponds to an algorithm iteration working on a distinct level of the quadtree. On the right side of the same figure, we can also observe how the quadtree grows up at each iteration/level. More specifically, at each iteration the algorithm determines which quadrants need to be split. Objects falling in quadrants to split are re-assigned by computing the new quadrant indices z' (highlighted in red in the "Iterations" table). Quadrants that have not to be split are added to the set of $\mathcal{C}$ cells. Objects belonging to the latter kind of quadrants (highlighted in green in the "Iterations" table) can be ignored during the successive iterations (the ignored cells are highlighted in grey in the "Iterations" table), since these already belong to quadtree leaves.

In the running example the algorithm stops at **Level 2**, since all the quadrants created at this level contain an amount of objects not greater than th_{quad}. Note that also the maximum quadtree level $l_{max} = 2$ is reached for 8 leaves of 10.

Complexity. The computation of a single Morton code has a fixed cost determined by the number of bits used for coordinate representation; therefore, *GPUcalculateMortonHash* complexity is equivalent to $O(|P|)$. *GPUradixSort* complexity is $O(b \cdot |P|) \simeq O(|P|)$, where b represents the base value during sorting ($b \ll |P|$). *GPUdetectQuadrants* worst-case complexity is $O(l_{max} \cdot |P| + 2\sum_{l=0}^{l_{max}} 4^l)$, where the first term is due to the amount of objects scanned in V_P at each iteration, while $2 \cdot \sum_{l=0}^{l_{max}} 4^l = 2 \cdot \frac{1-4^{l_{max}-1}}{1-4}$ represents the maximum amount of starting and ending

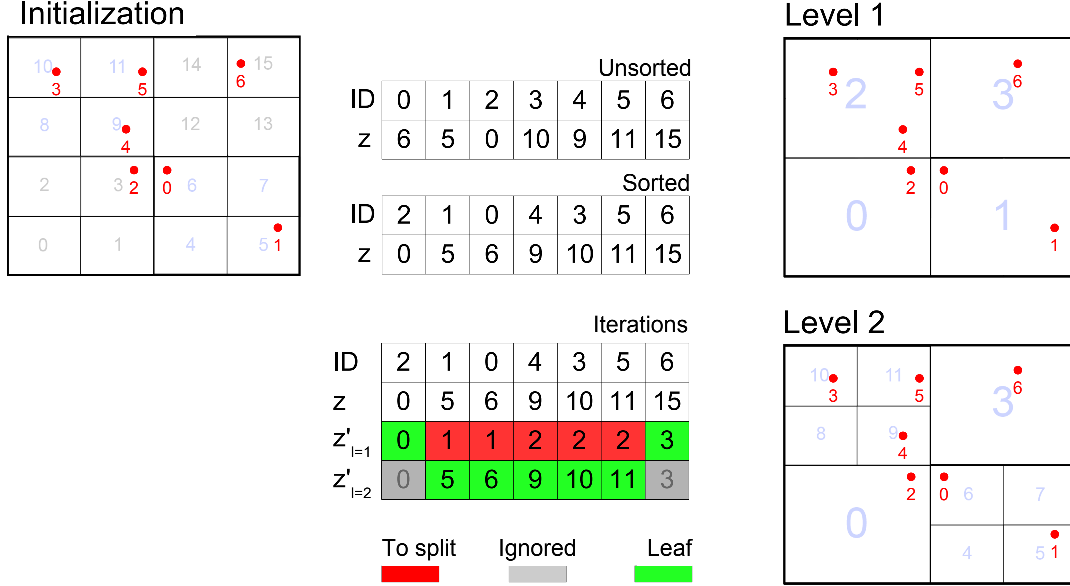


Figure 2.4: Example of quadtree construction with 7 objects, $th_{quad} = 1$ and $l_{max} = 2$.

indices - which has to be written out in memory - related to the 4^l quadtree quadrants at any level l. We observe that the amount of quadrants created at each level is orders of magnitude lower than $|P|$, hence the related computational overhead is negligible. As a consequence, the average complexity can be safely approximated to $O(l_{max} \cdot |P| + 2 \sum_{l=0}^{l_{max}} 4^l) \simeq O(l_{max} \cdot |P|)$. *CPUcheckQuadrants* has a worst-case complexity equal to $\sum_{l=0}^{l_{max}} 4^l = \frac{1-4^{l_{max}-1}}{1-4}$. Again, its complexity is practically negligible according to the above considerations.

Summing up, the complexity of the iterative process is dictated by the number of objects processed and the depth $l_{deep} \leq l_{max}$ reached in the quadtree construction, yielding $O(l_{deep} \cdot |P|)$. Since l_{deep} is usually a low constant, the overall complexity can be approximated to $O(|P|)$.

Building a lookup table to map coordinates to cells (QUAD). The usual approach for finding the quadtree leaf that corresponds to the coordinates of an object would consist in traversing the tree from the root, recursively choosing the relevant node until a leaf is reached. Unfortunately this approach entails repeated irregular memory accesses and a non predictable number of operations for each leaf search. The second issue, in particular, would cause branch divergence and potential sub-optimal occupancy of GPU cores.

For this reason we use a different approach, characterized by a slightly larger memory footprint. Let us suppose that the deepest level created in a quadtree $\mathcal{C}$ is l_{deep}, $l_{deep} \leq l_{max}$. Thus we virtually divide the space covered by $\mathcal{C}$ according to a uniform squared grid composed of $2^{\cdot l_{deep}} \times 2^{\cdot l_{deep}}$ cells, and denote it by $\mathcal{C}^{l_{deep}}$. In

other words, we cover $\mathcal{C}$ such that each quadtree leaf created at level l_{deep} corresponds exactly to a single cell in $\mathcal{C}^{l_{deep}}$. Thanks to the PR-quadtree properties, any quadtree leaf at a level l, $l \leq l_{deep}$, corresponds to the union of $4^{(l_{deep}-l)}$ contiguous cells of $\mathcal{C}^{l_{deep}}$. Therefore, a mapping between $\mathcal{C}^{l_{deep}}$ cells and $\mathcal{C}$ cells can be easily established by means of a lookup table z_{map}, which maps each $\mathcal{C}^{l_{deep}}$ cell, identified by a pair (i, j), to the $\mathcal{C}$ cell containing it.

The idea behind this approach is exemplified in Figure 2.5. The example is derived from the one in Figure 2.4, and therefore $l_{deep} = 2$. Each $\mathcal{C}$ cell (quadtree leaf) is identified by a pair (l, z) (an integer is indeed sufficient to store each pair), where l is the leaf level and z its Morton code at level l, whereas each cell in $\mathcal{C}^{l_{deep}}$ is associated with the pair (l, z) identifying the cell of $\mathcal{C}$ containing it. We can observe that 4 distinct $\mathcal{C}^{l_{deep}}$ cells are mapped to the same $\mathcal{C}$ cell $(1, 0)$, and other 4 distinct $\mathcal{C}^{l_{deep}}$ cells are mapped to the same $\mathcal{C}$ cell $(1, 3)$.

Therefore, given any pair of coordinates, it is possible to find the associated $\mathcal{C}$ cell by first computing the associated $\mathcal{C}^{l_{deep}}$ cell index, namely a pair (i, j), and then performing a lookup in z_{map}. Both operations have constant complexity, even though we have to mention that the performance related to the lookups in z_{map} heavily depends on the ability to exploit the GPUs caching capabilities. Indeed, z_{map} may have a relevant size - depending on l_{deep}. In light of this, it is important the *memory layout* of z_{map} to enhance data locality.

As regards the *memory layout* of the bidimensional array z_{map}, instead of using the typical row-major order memory layout, we access it according to the Morton code obtained from index pairs (i, j) used to access the array. Since all objects and queries are first associated with the Morton code of the cell (i, j) in $\mathcal{C}^{l_{deep}}$ which contains them, and then are *sorted* by this code, during the indexing operation described below we access z_{map} by exploiting temporal and spatial locality. This is because when we scan objects and queries that are memorized nearby, we also access nearby elements in z_{map}.

The initialization of z_{map} is performed entirely on GPU (function $GPUbuildZMap$, line 13 in Algorithm 1), by assigning each $\mathcal{C}$ cell (quadtree leaf) to a GPU streaming multiprocessor, which in turn initializes the interval of cells (elements of the lookup table) in $\mathcal{C}^{l_{deep}}$ contained by the $\mathcal{C}$ cell assigned.

The complexity of $GPUbuildZMap$ is $O(|\mathcal{C}| + |\mathcal{C}^{l_{deep}}|)$ and, in practical terms, negligible.

Indexing moving objects and queries (QUAD). The goal of this phase is to map objects locations and queries to C cells (quadtree leaves). Each object location is mapped to a single cell while each query can be potentially mapped to multiple cells (Definition 9).

To convert the position of all objects in P to cells identifier c of $\mathcal{C}$, their 2-dimensional coordinates are first mapped to grid coordinates (i, j) in the $\mathcal{C}^{l_{deep}}$ grid, where l_{deep} is the deepest level of $\mathcal{C}$. Subsequently, the Morton codes identifying

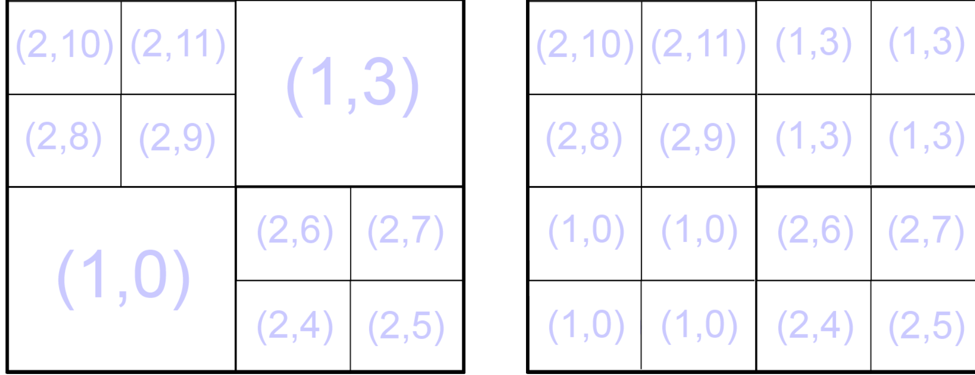


Figure 2.5: Example of the mapping established by z_{map} between the quadtree-induced grid $\mathcal{C}$ (left side) and the uniform grid $\mathcal{C}^{l_{deep}}$ (right side) related to the quadtree deepest level.

the cells of $\mathcal{C}^{l_{deep}}$ are derived from (i, j). Then, objects are *sorted* according to such Morton codes in order to exploit caching when subsequently accessing z_{map}, where z_{map} is used to retrieve the final quadtree cell identifier $c = z_{map}[i, j]$, $c \in \mathcal{C}$, in which the objects fall. We remark that objects remain sorted after such mapping thanks to quadtrees structural properties: this will be exploited during the filtering and decoding phases (see Sections 2.2.4 and 2.2.5), since query processing happens at cell level.

To convert a range query, characterized by a rectangular region, we need to identify all the relevant cells in $\mathcal{C}$, i.e., all the cells that spatially intersect the query. This process entails to identify, for each query q, a set of *subqueries*, each corresponding to the spatial restriction of the rectangular region of q to a relevant cell in $\mathcal{C}$. More formally, we have $g(q) = \{c_1, c_2, \ldots, c_n\} \subseteq \mathcal{C}$ (see Definition 9), where q intersects or covers each $c_i \in \mathcal{C}$. We thus refer to each pair (q, c_i) as a *subquery* of q.

First, as in the objects case, queries are first associated with a $\mathcal{C}^{l_{deep}}$ cell, namely their Morton codes, through their reference corner (in case part of their spatial extent falls outside the MBR $\mathcal{G}$, only the area in common with $\mathcal{G}$ is considered), and then sorted accordingly to such codes in order to exploit caching when subsequently accessing z_{map}.

Then, to obtain the subqueries, we start by identifying all the $\mathcal{C}^{l_{deep}}$ cells intersected by the query. We map each of these cells identified by a pair (i, j) to the corresponding $c \in \mathcal{C}$ cell by exploiting z_{map}. Depending on the spatial distribution, it is very likely to have multiple cells of $\mathcal{C}^{l_{deep}}$ that intersect the range query, and are thus mapped to the same $\mathcal{C}$ cell. This behavior could create *duplicate* subqueries, i.e., the same query mapped multiple times to the same cell of C. Figure 2.6 illustrates the problem and sketches our solution to avoid the presence of multiple subqueries mapped to the same $\mathcal{C}$ cell. In the left picture of the figure, we can see how the original query q_1 falls over multiple $\mathcal{C}$ cells (specifically, 6 distinct cells). Among these, we consider the intersection between q_1 and the $\mathcal{C}$ cell $(1, 3)$ (yellow

area). In the right picture, which illustrates the uniform grid $\mathcal{C}^{l_{deep}}$ associated with $\mathcal{C}$ through z_{map}, we can note that there are multiple cells on the right-upper part of q_1 that map to cell $(1,3)$ in $\mathcal{C}$ (specifically, 4 distinct cells of $\mathcal{C}^{l_{deep}}$). Therefore, q_1 would yield 4 subqueries that map to the same $\mathcal{C}$ cell $(1,3)$. To avoid duplicates, we always select the subquery having the minimal grid coordinates (highlighted in green).

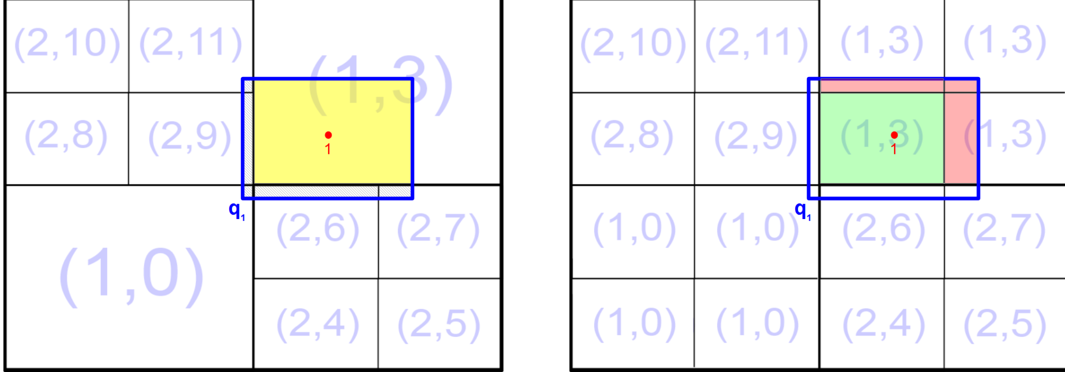


Figure 2.6: Query indexing example with QUAD.

The queries are indexed, similarly to UG ($\mathsf{UG}_{Baseline}$), in two separate phases. During the first phase the amount of subqueries per each original query is determined. In order to determine the memory location where each subquery will be written, an exclusive prefix sum is performed over the vector containing the amounts of subqueries per query. Then, in the second phase, subqueries are actually written using the information computed during the first phase, and classified according to the *intersecting/covering* dichotomy.

The overall complexity of the indexing phase can be expressed in the following terms:

- for what is related to the sorting operations needed to optimize the accesses in z_{map}, we have $O(|P| + |Q| + b \cdot (|P| + |Q|)) \approx O(2(|P| + |Q|))$: $|P|$ and $|Q|$ are due to l_{deep} Morton codes computations while $b \cdot (|P| + |Q|)$ relates to the actual sorting performed, by means of Radix Sort, over P and Q.

- for what is related to the subsequent operations, we have $O(|P| + O(2|Q| \cdot |\mathcal{C}^{l_{deep}}| + |Q| + 2|\hat{Q}|)$: $|P|$ relates to the lookups in z_{map}, $2|Q| \cdot |\mathcal{C}^{l_{deep}}|$ relates to the query indexing which happens in two separate phases ($|\mathcal{C}^{l_{deep}}|$ is due to the amount of $\mathcal{C}^{l_{deep}}$ cells spanned by an original query in the *worst* case) and $2|\hat{Q}|$ relates to the subqueries written during the second phase (lookups in z_{map} are included in the complexity), noting that $|\hat{Q}| = |Q| \cdot |\mathcal{C}^{l_{deep}}|$ in the worst case.

In light of these considerations, the amount of subqueries to be checked during indexing may be relevant, therefore we remark the importance of exploiting caching when accessing z_{map}.

Sorting (QUAD). Once subqueries are mapped to $\mathcal{C}$ cells, we sort the associated augmented tuples to store them in contiguous memory locations. The reason of this phase is analogous to the sorting carried out in the UG and $\mathsf{UG}_{Baseline}$ cases.

Note that, unlike the UG and $\mathsf{UG}_{Baseline}$ cases, for QUAD we do not need to re-sort the moving objects. The sorting done during the indexing phase, according to the cell identifiers of $\mathcal{C}^{l_{deep}}$ grid, is enough to guarantee locality during the following query processing phase, thanks to the quadtrees structural properties.

As regards subqueries, we have to sort them since there is no guarantee about the order in which they are written in global memory during the indexing phase. Consequently, the structure of vectors associated with the subqueries tuples, $\hat{Q}$, is sorted on GPU by means of Radix sort according to the identifier associated with the cell [3]. Moreover, in order to support the optimizations discussed in Section 2.2.6, each identifier is augmented so to signal whether a subquery is either covering or intersecting. In this way, after the sorting operation, all intersecting subqueries are placed at the beginning of their structure of vectors.

Since the sorting algorithm is Radix Sort, the complexity of the sorting step is $O(b \cdot (|\hat{Q}|)) \approx O(|\hat{Q}|)$.

2.2.4 Filtering

The goal of the filtering phase is to compute range queries over object locations, and store the containment test outcomes (i.e., which object locations are contained in each query range) conveniently. Since, by definition, covering subqueries entirely cover the cell onto which they fall, the filtering phase can be actually limited to intersecting subqueries, delegating the processing of the former type to the optimization described in Section 2.2.6.

In this context we conveniently denote by $\mathcal{C}_\alpha \subseteq \mathcal{C}$ the set of active cells, i.e., those cells containing at least one object.

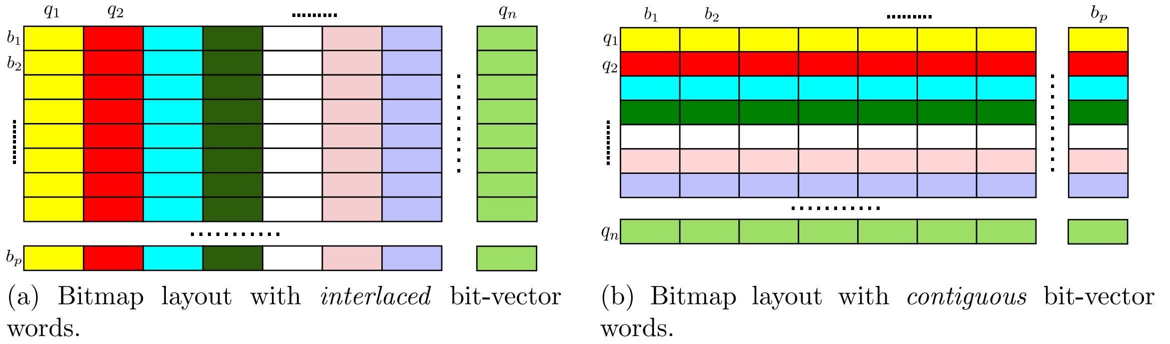
Both QUAD and UG store the containment test outcomes in form of bitmaps (one per *active* cell), which will be decoded at a later stage in order to obtain a final compact representation of the positive containment test outcomes.

Filtering is performed in parallel: each active cell in $\mathcal{C}_\alpha$ is assigned to a block of GPU threads to obtain a bitmap which refers to object locations and subqueries falling in the corresponding cell.

In the last part of this subsection we also detail the simplifications adopted by the $\mathsf{UG}_{Baseline}$ filtering algorithm. In the experimental section we will use $\mathsf{UG}_{Baseline}$ as a baseline to assess the benefits of using the bitmaps and an additional (decoding) phase needed to extract the final query results from these.

[3] While in the UG and $\mathsf{UG}_{Baseline}$ cases this identifier is represented by an integer storing grid coordinates, in the QUAD case it is a pair (l, z), which is indeed conveniently stored as an integer.

Bitmap layouts. Bitmaps are arranged in memory by using two different layouts across the following phases, since each layout better fit specific kinds of operations on GPU. For each active cell $c \in \mathcal{C}_\alpha$, the filtering phase initially compute a 2D bitmap B^c characterized by an interlaced column-wise layout (Figure 2.7.a), where each column q_i refers to a single query and each row b_j refers to a fixed block of w object locations inside the cell. The width of each column is w bits (here we assume $w = 32$) and the content of a w-bit word corresponding to q_i and b_j indicates if the object locations associated with block b_j are contained in the extent of query q_i. Thus, a single bitmap element $B_{n,m}$, where n and m are the row and column at bit level, represents the containment test outcome between the (m/w)-th query and the $((n \cdot w) + (m \ mod \ w))$-th object location.



(a) Bitmap layout with *interlaced* bit-vector words.

(b) Bitmap layout with *contiguous* bit-vector words.

Figure 2.7: Bitmap layouts used during the processing.

This interlaced layout choice facilitates coalesced memory accesses when threads in the same warp are in charge of computing containment tests of a consecutive group of subqueries (see *Interlaced bitmap generation*). Indeed, these threads can write results to consecutive memory positions, taking advantage of coalesced memory accesses. Since threads in the same warp perform exactly the same operations, and each thread is in charge of writing a distinct 32-bit word, this solution eliminates the need of any synchronization mechanisms at thread block or global levels.

While the interlaced layout entails benefits when the bitmap is produced, it hinders the extraction of all the containment test outcomes referred to the same subquery. For this reason, after production, each interlaced bitmap is transposed word-wise to improve the memory throughput when the bitmaps are decoded to extract the final results. We will refer to this transformation as the *linearization* of interlaced bitmaps. Indeed, in the row-wise layout resulting from the transformation (Figure 2.7.b), single containment test outcomes are linearly indexed and bit-vectors associated with each subquery have their words arranged consecutively in memory, which favours subquery-wise read coalescing during the decoding phase. The linearization transformation can be expressed as a massively-parallel operation which is efficiently performed on GPU.

Filtering – Interlaced bitmap generation. During this stage we divide query result computation to exploit three different kind of parallelism allowed by GPUs.

Block parallelism allows to process independent tasks. Since we are considering subqueries, which are restricted to a specific index cell by definition, the computation of the results in different cells can proceed independently, producing distinct result bitmaps. Thus, active (non-empty) index cells $c \in \mathcal{C}_\alpha$ are assigned to distinct *blocks* of GPU threads.

Each block of GPU threads is executed asynchronously by the same *streaming multiprocessor* (SM). *Thread parallelism* allows for cooperation among threads in the same block. Each thread in a block is in charge of computing a distinct subquery that is present in the index cell assigned to the block. Since each bitmap is common for all the subqueries(threads) in a cell(block), the cooperation among threads is used to ensure coordination when writing out the containment test outcomes (0/1) in the bitmap.

Whenever possible it is strongly suggested to orchestrate the thread scheduling to hide memory access latency by having an amount of threads per thread block exceeding the amount of cores per single streaming multiprocessor. However, only subsets of threads can run in parallel at a given time. These subsets of sz_{warp}[4] synchronous and data parallel threads are called *warps*. Thanks to synchronous execution, *warp parallelism* allows to avoid synchronization operations. Furthermore, threads in the same warp benefit from coalesced memory accesses when they access consecutive (or identical) memory positions, so that several memory accesses are combined in a single transaction.

In our solution all the threads of a warp access the device memory in an optimal way: they read the same input data (object locations) synchronously (this exploits GPU caching), access them consecutively (subqueries, spatial locality, this exploits coalescing) and, thanks to the interlaced bitmap layout, write simultaneously results (w-bits bitmap words) to consecutive memory locations (this entails coalesced memory access).

To better explain the latter point, we illustrate in Figure 2.8 the role of different threads in a thread block during the creation of an interlaced bitmap: each group of sz_{warp} columns is collectively updated by a warp of threads in the thread block. Each column contains a set of 32 bit-wide words, $b_1, \ldots, b_p$, associated with a subquery q. The first words (the b_1's) associated with the various subqueries, and computed by the warp threads, are stored simultaneously in memory. The same holds for the second block of words (the b_2's), which is stored immediately after, and so on. The bitmap words updated simultaneously by the threads are stored consecutively thanks to the interlaced layout of the bitmap. This permits the writes to be *coalesced*

[4] 32 threads per warp in current GPUs.

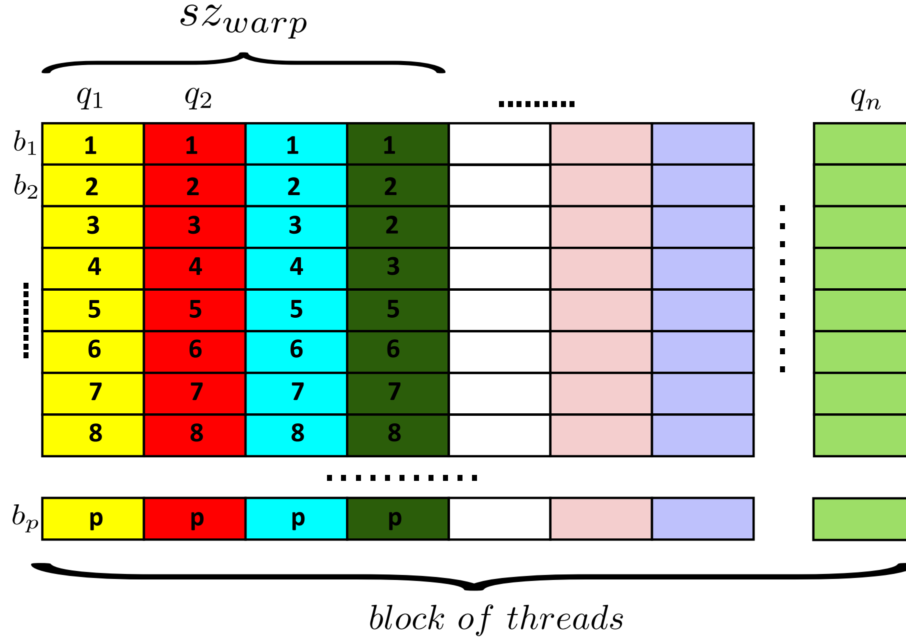


Figure 2.8: Collective creation of a interlaced bitmap by a block of GPU threads.

.

Algorithm 2: QUAD and UG filtering phase.

---

1 **begin**
2   $numPoints \leftarrow 0$
3   $wordIndex \leftarrow 0$
4   $wordBitmap \leftarrow 0$
5   **foreach** $c \in \mathcal{C}_\alpha$ **parallel**$_{\textbf{block}}$ **do**
6   **foreach** $q \in c$ **parallel**$_{\textbf{thread}}$ **do**
7   **foreach** $p \in c$ **do**
8         $numPoints \leftarrow numPoints + 1$
9   **if** $p \in q$ **then**
10           $setBit(wordBitmap, p)$
11   **if** $numPoints \ mod \ 32 = 0$ **then**
12           $writeBitmap(wordBitmap, wordIndex, q)$
13           $wordBitmap \leftarrow 0$
14           $wordIndex \leftarrow wordIndex + 1$

The pseudocode in Algorithm 2 illustrates the main points of the interlaced bitmap generation. Distinct *blocks* of GPU threads process in parallel active index cells $c \in \mathcal{C}_\alpha$ (line 5). Each thread in a block is in charge of computing the results of a distinct subquery present in c (line 6).

All threads read the same sequence of object positions and update a private *32 bit-wide* register that contains the bitwise information about the presence/absence

of 32 distinct object locations in its own range query (line 10). When the threads in a warp have completed the update of the current word (i.e., they have finished to compute a block of 32 containment tests or they have computed all the blocks), all threads proceed by flushing the content of their private registers simultaneously to the global device memory at the right memory displacement (line 12). The computation goes on until all the subqueries have been computed.

The execution of the inner loop (line 7) is scheduled by the GPU at warp level: it depends on resource availability and memory access latency, but threads in the same warp are granted to be synchronous. For example, *wordBitmap* will be completed simultaneously for all the threads in the same warp.

We finally note that all the threads of any warp access the device memory in an optimized way: they read the same input data synchronously (object positions, line 7) or consecutively (subqueries, line 6), thus always exploiting data spatial locality. Moreover, all threads write simultaneously words that are stored consecutively in memory, thus coalescing the writes and boosting the overall GPU global memory throughput (line 12).

Filtering – Bitmap linearization. The goal of this operation is to transform each bitmap from the interlaced column-wise layout to the linearized row-wise one, so that bitmaps can be more efficiently processed during the subsequent decoding phase (Section 2.2.5). This transformation is performed on GPU and is depicted in Figure 2.9. The Figure shows the work of a single warp composed of sz_{warp} threads. All the synchronous threads cooperate to transpose blocks of words, one block at a

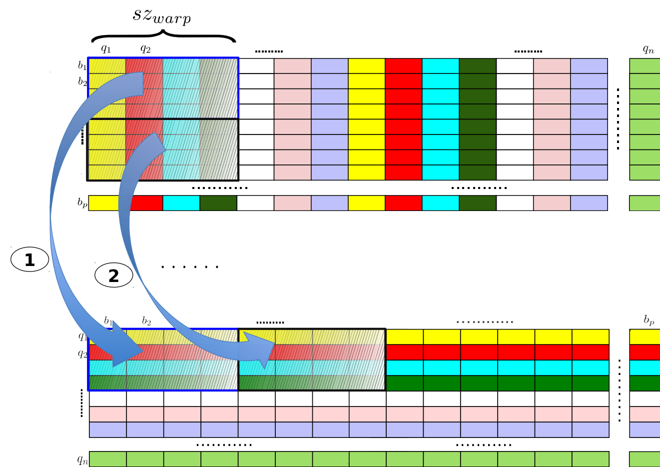


Figure 2.9: Collective linearization of a group of columns by a warp of GPU threads.

time. Each block is made up of $sz_{warp} \times sz_{warp}$ words. The linearization of a block of words is carried out in two steps:

1. First, all the sz_{warp} threads read in parallel the bitmap words associated with the subqueries assigned to them (a row of the input block of words at a time). The threads proceed until the end of the block (for sz_{warp} rows of the block). While reading the words associated with subqueries, each thread incrementally prepares a linearised block of words in a temporary buffer stored in the fast shared memory available to each SM.

2. Once the input block has been read completely and linearized in the shared memory, the warp threads start their second step, where they move the linearized block to the device global memory. The only difference is that they collaborate by writing portions of the bitmaps associated with each subquery in parallel. First they write the first row of the linearised block in parallel, then the second, and so on. This, in turn, entails coalesced writes.

Even if the shared memory has limited size, the block-wise linearization does not saturate the shared memory thanks to the small size of the data that are linearized simultaneously.

Filtering phase in UG$_{Baseline}$. The UG$_{Baseline}$ algorithm does not use bitmaps and computes the query results on the fly. Therefore, its filtering phase is simpler and does not require subsequent linearization and decoding phases. The pseudocode in Algorithm 3 illustrates this simpler strategy.

Algorithm 3: UG$_{Baseline}$ filtering phase

```
1 begin
2   foreach c ∈ C_α parallel_block do
3     shared resultBuffer ← ∅
4     foreach q ∈ c parallel_thread do
5       foreach p ∈ c do
6         if p ∈ q then
7           appendResult(resultBuffer, q, p)
8         if full(resultBuffer) = true then
9           flush(resultBuffer)
```

Each active cell is assigned to a single thread block (line 2). Query results (pairs of object locations and subqueries such that the object location is contained in the subquery range) are computed in parallel (line 6) and immediately appended to a buffer in shared memory (line 7) common to all the threads in the block. The threads access and update the buffer by means of *atomic operations* in order to guarantee its consistency. Once the buffer contains an amount of results greater than a fixed threshold (line 8), the threads synchronize and cooperatively flush its content out to global memory. In order to guarantee the consistency of the data written to the

global memory a result counter, shared among all the thread blocks, is accessed and updated atomically.

Filtering – Complexity. The *interlaced bitmap generation* (Algorithm 2) represents the time dominant part of the filtering phase. The filtering complexity, determined by the amount of containment tests to be computed, is:

$$O(\sum_{c \in \mathcal{C}_\alpha} |\hat{Q}_I^c| \cdot |P^c|), \tag{2.1}$$

where $C_\alpha \subseteq \mathcal{C}$ is the set of active grid cells, P^c the set of object locations in c and $\hat{Q}_I^c$ the intersecting subqueries associated with c.

In general we observe that decreasing the grid cells size yields a smaller number of containment tests, even if the number of intersecting subqueries to manage is larger. An arbitrary decrease, however, has negative side effects, such as the fragmentation of the intermediate results in a large number of small bitmaps, which in turn influences negatively the overall running time due to inefficient computational resource usage and scattered memory accesses. Moreover, an arbitrary cell size decrease may induce an intersecting subqueries increase rate eclipsing the decrease rate related to the average amount of objects per active cell, therefore raising the complexity at some point. In light of these considerations we argue there is a trade-off between decreasing the overall number of operations executed and optimizing parallelism and memory access costs.

The *bitmap linearization* has linear complexity with respect to the number of bitmaps words. Since each bit corresponds to an intersection test, the two subphases has the same complexity.

2.2.5 Bitmap decoding

The end product of the filtering phase of both UG and QUAD consists of a set of linearized bitmaps, one per active cell, containing both the positive and the negative containment test outcomes, related to object locations and subqueries associated with the active cells. The goal of the decoding phase is to process such bitmaps in order to extract the final query result set, i.e., the positive occurrences in the bitmaps.

Accessing the bitmaps content in order to extract the positive occurrences represents a memory and computationally intensive task which, thanks to the linearized layout, can be efficiently and conveniently parallelized on GPU.

The decoding phase, common to UG and QUAD, proceeds as follows: for each intersecting subquery a list of objects identifiers is generated (according to the problem definition in Section 1.1.5), where each identifier represents a positive occurrence. Since each active cell (bitmap) represents a single task, the decoding operation can progress by decoding the tasks in chunks: this allows us to transmit to the CPU

the information related to a previously decoded chunk of bitmaps while the GPU progresses by decoding the next chunk of unprocessed bitmaps. This also allows us to overlap computations carried on the GPU with I/O transfers from GPU to CPU. We highlight that the result lists related to intersecting subqueries originating from the same query have no results in common and preserve the identifier of the original query, so it is trivial to merge them to obtain the final result set.

The pseudocode in Algorithm 4 illustrates the GPU part of this strategy.

Algorithm 4: Decoding phase

```
1  begin
2    foreach bitmap parallel_block do
3      foreach q ∈ Q_bitmap parallel_warp do
4        q_id ← loadQueryID(q)
5        shared q_bv ← loadQueryBitVector(q, bitmap)
6        resultSet_q ← linearScan(q_bv)

7        foreach r ∈ resultSet_q parallel_thread do
8          p ← loadPoint(r, bitmap)
9          writePID(p, bitmap)

10       writeQueryDetails(q_id, |resultSet_q|, bitmap)
```

Each bitmap refers to a specific index active cell and is decoded by a specific thread block (line 2). Each intersecting subquery referred by the bitmap is assigned to a *warp* (line 3), so that each warp is in charge of several subqueries. Since threads in the same warp are synchronous, the use of warp level granularity allows to safely avoid the use of synchronization mechanisms inside the loop.

Threads in the same warp transfer (line 5) the words composing the bit vector of the currently considered subquery from device memory to shared memory. Since consecutive threads read consecutive memory positions, this operation yields coalesced memory reads.

Once the transfer is over, each thread in the warp determines the subset of results it will write to global memory so that writes can be coalesced (i.e., the i-th warp thread will write the (i mod *warpSize*)-th positive results). This is achieved by using the *linearScan* function (line 6). Here, each thread performs concurrently a linear scan over the query bit vector words in order to take note of the positive results it will write to global memory. This operation can be carried on efficiently thanks to the shared memories *broadcast* capability, which avoids bank conflicts between threads in a warp if they are all reading the same address. Moreover, each thread can store the information related to the positive results it has to write in its own private registers, since the decoding between lines 3 and 10 is scalable with respect to the subqueries bit vectors size (which is fixed for a given bitmap).

Once these information are determined, the warp threads finally perform a collective write of the subquery results, yielding coalesced writes (function *writePID*,

line 9). Each warp also knows exactly where the results of each subquery has to be stored in global memory, since the overall amount of results per subquery can be determined during the filtering phase and stored in a vector (therefore, an exclusive prefix sum over the vector returns the correct memory location for each result).

Decoding – Complexity. Considered we have one bitmap for each active cell c, the overall decoding complexity (Algorithm 4) is:

$$O(\sum_{c \in \mathcal{C}_{\mathcal{A}}} |\hat{Q}_I^c| \cdot |P^c| + \sum_{q' \in \hat{Q}_I^c} |R_{q'}|), \tag{2.2}$$

where R_q denotes the result set of a query q. The first term is due to the access for each cell $c \in \mathcal{C}_A$ to the respective bitmap, each containing $|\hat{Q}_I^c| \cdot |P^c|$ bits (number of intersecting subqueries times the number of objects in the cell), while the second term is due to the writes of the intersecting subqueries results. As we highlighted for filtering, the grid cells size indirectly affects the decoding complexity.

2.2.6 Optimizations

Covering subqueries optimization – covering subqueries information notification. UG, $\text{UG}_{Baseline}$ and QUAD take advantage of the covering subqueries (we denote their set by $\hat{Q}_C$) in order to reduce the result set the GPU computes, thus saving a relevant amount of GPU computations during the filtering and decoding phases and I/O traffic between the GPU and the CPU during the decoding phase. This is achieved by notifying the CPU the covering subqueries data, together with the object locations enclosed in the $\mathcal{C}$ cells, just after the end of the *sorting* phase.

Covering subqueries optimization – covering subqueries result set expansion. As soon as the data relevant for reconstructing the $\hat{Q}_C$ result set is notified, the CPU can start its expansion. Indeed, for each $q \in \hat{Q}_C$ we have to consider pairs $(q, c \in \mathcal{C}_\alpha)$, where each c represents the index of a cell entirely covered by q. After the final sorting step of the indexing phase, the lists of object locations associated with each cell of $\mathcal{C}$ are sent to the host memory, so that the CPU can directly access them. Therefore, by looking at these lists, the CPU can immediately extract the result set related to q. This operation is performed by the CPU in background, between the end of the *sorting* phase and the end of the *decoding* phase of the GPU.

Covering subqueries optimization – complexity. The cost related to the covering subqueries result set expansion is $O(\sum_{q \in \hat{Q}_C} |R_q|)$, due to the scan of the list of object locations associated with the cell of each covering subquery. Since this task is carried out by the CPU and overlapped with the tasks performed on GPU, in practice it has very little or negligible impacts on the overall execution time.

Task scheduling optimization. A proper GPU task scheduling policy can substantially improve the overall execution time by reducing the inactivity time of the GPU streaming multiprocessors in presence of unbalanced workload distributions. In this context we define a single workload GPU *task* as the *set of computations* related to the intersecting subqueries falling inside a specific grid active cell, whereas the *computational weight* of a given task is the amount of containment tests associated with it, i.e., the product between the amount of intersecting subqueries and the amount of object locations falling inside the related active cell.

In general we can take into consideration three high-level GPU task scheduling strategies when assigning the workload tasks to the GPU streaming multiprocessors [45], namely, the static task list strategy (which is the default one used by CUDA), the task queue based strategy and the task stealing strategy.

Since the computational weight of each task is known a-priori once the sorting phase is performed, and that new sub-tasks cannot be created at run-time, we deem that the first strategy, together with a reordering of the task list according to the tasks computational weight, is the best one for the scenarios considered; indeed, this strategy has the effect of batching together the execution of tasks having similar computational costs at the negligible cost related to the need of accessing the task list atomically whenever an idling GPU streaming multiprocessor available to take in charge the first non-assigned task (atomic access is required to ensure a single execution for each task).

This optimization is used during the filtering and decoding phases when assigning tasks (active $\mathcal{C}$ cells) to streaming multiprocessors.

2.3 Experimental Setup

All the experiments are conducted on a PC equipped with an Intel Core i3 560 CPU, running at 3,2 GHz, with 4 GB RAM, and an Nvidia GTX 560 GPU with 1 GB of RAM coupled with CUDA 5.5. The OS is Ubuntu 12.04. We exploit a publicly available framework [46, 16] for both workload generation and testing. The framework comes with a number of sequential, CPU-based iterated spatial join algorithms. Among these, the *Synchronous Trasversal* algorithm (CPU-ST) is shown to be consistently the best[16] and thus we compare our GPU-based approaches against this algorithm.

As regards our GPU-based proposed solutions (QUAD and UG), we slightly modified the framework in order to offload the most time-consuming parallelizable tasks to the GPU, while delegating the others (mostly related to the GPU management) to the CPU.

We use three types of synthetic datasets: *(i) uniform datasets*, in which moving objects are distributed uniformly in the space; *(ii) gaussian datasets*, in which moving objects tend to gather around multiple *hotspots* by following a normal distribution. The skewness in the gaussian datasets depends on the number of hotspots:

the more the hotspots are, the more the objects tend to be uniformly distributed; *(iii) network datasets*, in which moving objects are distributed uniformly over the edges of a bidirectional graph representing a road network. In our experiments we use the San Francisco road network, derived from TIGER/Line files. This kind of datasets are characterized by a mild skewness, due to the constraint on the positioning of the objects. All the datasets are created using the generator provided by the framework, which is partly derived from the Brinkoff generator [47]. We consider just synthetic datasets since they allow to explore the vast space of parameters possibly influencing the algorithms' performance, whereas this would not be possible (or it would be rather contrived to do) with real-world datasets; moreover, we note there is a lack of real-world datasets which would serve the purposes of this work. Overall, our choice is coherent with other relevant works (the reader may see, for example, [16]).

In all tests we compute repeated range queries over 30 ticks. To model object movements the framework generates 30 instances of each dataset, one for each tick.

Table 2.1 summarizes the main parameters used to generate the datasets. The listed parameters apply to all the datasets, except for the amount of hotspots which is relevant for gaussian datasets only. The framework uses a generic spatial distance unit u (e.g., meters).

The decoded results are produced by the GPU in blocks, i.e., for each query/subquery a list of (positive) results is produced, whereas for CPU-ST the results are produced one by one. To avoid bias in the performance comparison, we thus force QUAD and UG to report the GPU-generated results to the framework in pairs, hence expanding the lists of objects belonging to the result set of each query.

2.4 Experimental Evaluation

The experimental studies conducted for this work are introduced below, and are denoted by $S1, \ldots, S8$:

- *S1* We study how a *lock-free data structure*, like the bitmap proposed to encode the intermediate output of the range queries, entails considerable improvements over a baseline GPU algorithm that recurs to locks to assure the result buffer consistency.
- *S2* We analyze the advantages coming from the *covering subquery optimization* in reducing computations (and related I/O traffic) performed on the GPU side.
- *S3* We show how a proper GPU *task/block scheduling* can improve the UG and QUAD performances by reducing the workload unbalances deriving from skewed spatial distributions.
- *S4* We study how the data distribution skewness influences the choice of the optimal grid coarseness, focusing on UG.

Spatial region	All tests occur in a squared spatial region with side length of 22500 u.
Amount of objects	We vary the number of moving objects from $100K$ to $1500K$. In some tests the number of moving objects is fixed and the exact amount is explicitly stated in their descriptions.
Objects maximum speed	In all tests the maximum speed of each object is fixed to 200 u per tick (Δt), where the objects are allowed to change their speed as described in [16]. In general, changes in speed may slightly alter the objects distribution but do not change the distribution general properties.
Query rate	The percentage of objects that issue a range query during every tick is always set to 100%.
Query size	All queries in a test are squared and, depending on the experiment, they may be all equally sized or not. We vary the side length in the range $[200\ u, 800\ u]$. The default value is 200 u.
Amount of ticks	Whenever not specified, the default amount of ticks, corresponding to different snapshots of a dataset, is 30. Consecutive snapshots are expected to exhibit slight changes, according to the properties of the dataset spatial distribution.
Query location	All the queries are centered around the objects issuing them.
Amount of hotspots	Depending on the experiments goals and specificities, the amount of hotspots is varied in the $[10, 150]$ range. Whenever not specified the default value used is 25.

Table 2.1: Data and workload generation parameters.

S5 We study how QUAD is able to automatically adapt to the spatial data distribution of a dataset, even when the distribution is highly skewed.

S6 We analyze the impact of various spatial distributions on the performance. To this regard we study mean and dispersion index related to the amount of objects per active cell achieved by UG and QUAD, and the relationships that these measures have with some important features that impact the performance of the system, such as the overall amount of subqueries and the proportion of covering/intersecting ones.

S7 This study analyzes the sensitivity of the UG and QUAD performances with respect to datasets characterized by different spatial distribution properties, such as the amount of objects and the query area.

S8 This final study analyzes how the main factors characterizing the datasets, such as the amount of objects, the query rate, the query area and the skewness, affect the system bandwidth β (as defined in Section 1.1.4).

It is worth remarking that in the final *S6...S8* studies we turn on all the optimizations devised for both UG and QUAD. In particular, a relevant amount of computations is avoided by distinguishing between covering and intersecting subqueries; we exploit the lock-free bitmaps to store the intersecting subqueries intermediate results; we heuristically balance the workload between the GPU SMs by reordering the tasks/blocks to be scheduled, from the heaviest to the most lightweight; finally, we always adopt the best possible grid coarseness for UG, given any dataset, by means of an oracle, although this strategy cannot be adopted in practical settings since it requires additional work to profile the UG performance for the various datasets.

2.4.1 Analysis on the benefits coming from the usage of bitmaps (S1)

In this study we evaluate the benefits coming from the usage of a lock-free data structure such as the bitmaps proposed. We focus on the filtering and decoding steps, since these are the only phases during which the bitmaps are utilized in order to significantly improve the performances. We limit this comparison to UG and UG$_{Baseline}$, since QUAD filtering and decoding phases are similar to UG ones, and the benefits over a naive lock-based technique are thus analogous.

We briefly remember that in UG$_{Baseline}$ the query result set is computed and transmitted to the CPU counterpart on the fly, during the filtering step, thus avoiding the need of a subsequent decoding phase. Moreover, since the results of different queries can be interleaved in the output stream, each result is represented as a pair (*query:point*).

In Figure 2.10 we can see how UG outperforms UG$_{Baseline}$, even when considering modest workloads. For our purposes, it is enough to compare UG and UG$_{Baseline}$

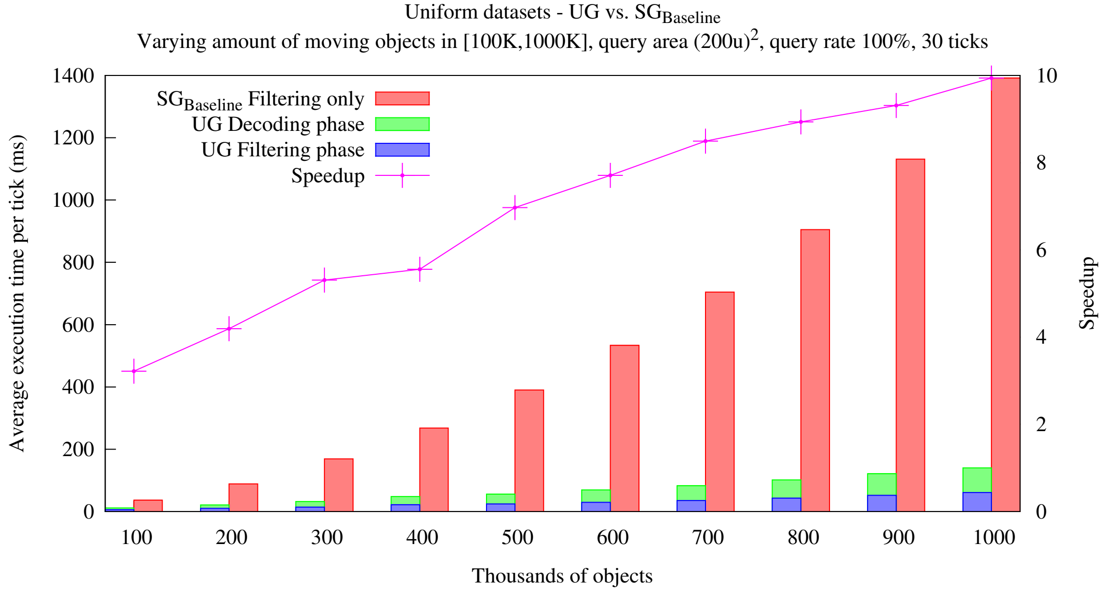


Figure 2.10: UG vs. $\mathsf{UG}_{Baseline}$ time analysis for uniform datasets by varying the number of objects in [100K,1000K]. The histograms of the UG filtering and decoding phases are time-stacked for clarity purposes.

on a uniform spatial distribution of object/queries. Note the line representing the speedup obtained by UG over $\mathsf{UG}_{Baseline}$ per single experiment: the $\mathsf{UG}_{Baseline}$ performance gets worse when the number of objects (and thus the output size) increases. This indicates that the main bottleneck of $\mathsf{UG}_{Baseline}$ is the synchronization mechanisms adopted, which affect negatively the performance when the amount of results increases.

2.4.2 Covering subqueries optimization (S2)

The overall goal of this study is to analyze the benefits coming from the covering subqueries optimization described in Section 2.2.6. We briefly remember that this technique aims to speed up the query processing in three ways: first, by reducing the overall amount of containment tests performed by the GPU; second, by reducing the amount of results determined at the GPU side, and thus the amount of data the GPU has to send back to the CPU once the filtering and the decoding phases are over; third, by leaving the CPU in charge of expanding the covering subqueries result sets, during the same time that the GPU processes the intersecting subqueries.

We focus on QUAD, given its more advanced spatial indexing and considered that, in terms of query covering management, UG carries out the same operations per each subquery covering an index cell. We compare two different versions of QUAD: the former, denoted by Covering ON, is the version where the covering subquery

optimization is exploited. The latter, denoted by Covering OFF, does not exploit the knowledge about the covering queries, thus considering all the subqueries as intersecting.

In our experiments we want to independently focus on two key parameters: spatial distribution *skewness* and *query area*. The skewness consistently influences *(i)* the *ratio* between covering and intersecting subqueries, and *(ii)* the *weight* of the covering subqueries in terms of the percentage of generated results. As regards *(i)*, the more the objects tend to gather in specific places, the more QUAD refines the grid in those areas, in turn increasing the aforementioned ratio. As regards *(ii)*, the smaller the average size of the QUAD cells is, the higher the probability that a subquery area completely covers a grid cell. Note that QUAD materializes dynamically the index cells, and generate smaller cells in correspondence to the spatial regions with higher object density. On these small and highly populated cells the ratio of covering subqueries to intersecting ones gets larger. This in turn increases the overall amount of results obtained from the covering subqueries, with positive returns on the performance.

Query area is another important factor, because it directly influences the ratio between covering and intersecting subqueries. Hence, we want to analyze how much this parameter influences the performances, aside from the skewness.

In the first batch of experiments we vary the skewness of the object spatial distribution in a set of gaussian datasets, by changing the number of hotspots in the interval [10,1000], while the amount of objects and the query area are kept fixed at 500K and $(400u)^2$, respectively. The top plot of Figure 2.11 shows how the exploitation of the covering subqueries greatly reduces the overall execution time, in particular when the skewness increases by reducing the amount of hotspots. The more the skewness is, the more the percentage of results coming from the covering subqueries is, thus increasing the performance gap between the two versions when the skewness gets larger. Finally, the bottom plot of Figure 2.11 gives a further explanation of the observed behaviour, and shows that the skewness is directly proportional to (or equivalently, the number of hotspots is inversely proportional to) the covering/intersecting ratio.

In the second batch of experiments we focus on a gaussian dataset having a fixed amount of 50 hotspots, thus characterized by a moderately skewed distribution. We vary the query area in the $[(200u)^2, (400u)^2]$ range, while the amount of objects is kept fixed at 500K. Figure 2.12 shows that, when the query area is increased, the gap between the two versions of QUAD gets larger as well, while the covering/intersecting subqueries ratio strictly follows the trend.

In conclusion, we argue that the percentage of results coming from the covering subqueries determines the extent of the advantages possibly coming from this optimization.This figure is essentially determined by the skewness characterizing the spatial object distribution, while the query area amplifies or reduces this phenomenon by changing the query results redistribution ratio between the covering and the intersecting subqueries.

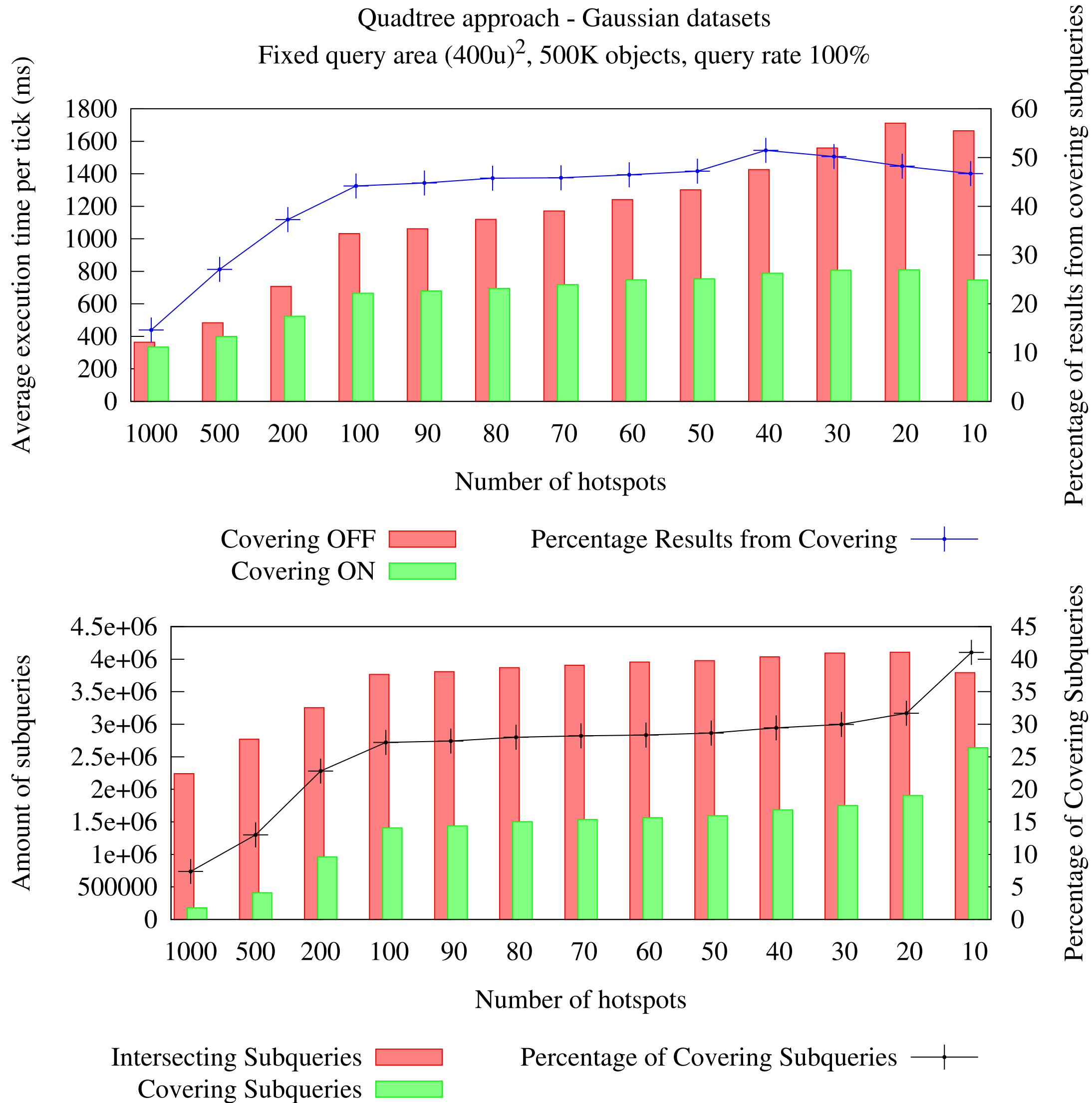


Figure 2.11: Gaussian datasets, 500K objects, query area $(400u)^2$, varying hotspots in [50,1000], query rate 100%, Covering ON vs OFF.

2.4.3 Task scheduling policy (S3)

In this section we investigate how the task scheduling optimization described in Section 2.2.6 can substantially improve the overall execution time of UG and QUAD by redistributing more evenly the workload among the GPU streaming multiprocessors. Besides analyzing the execution times in this study we also collect and study profiling data concerning the containment tests actually carried on by every streaming multiprocessor.

The top plot in Figure 2.13 shows that the reordering always reduces the exe-

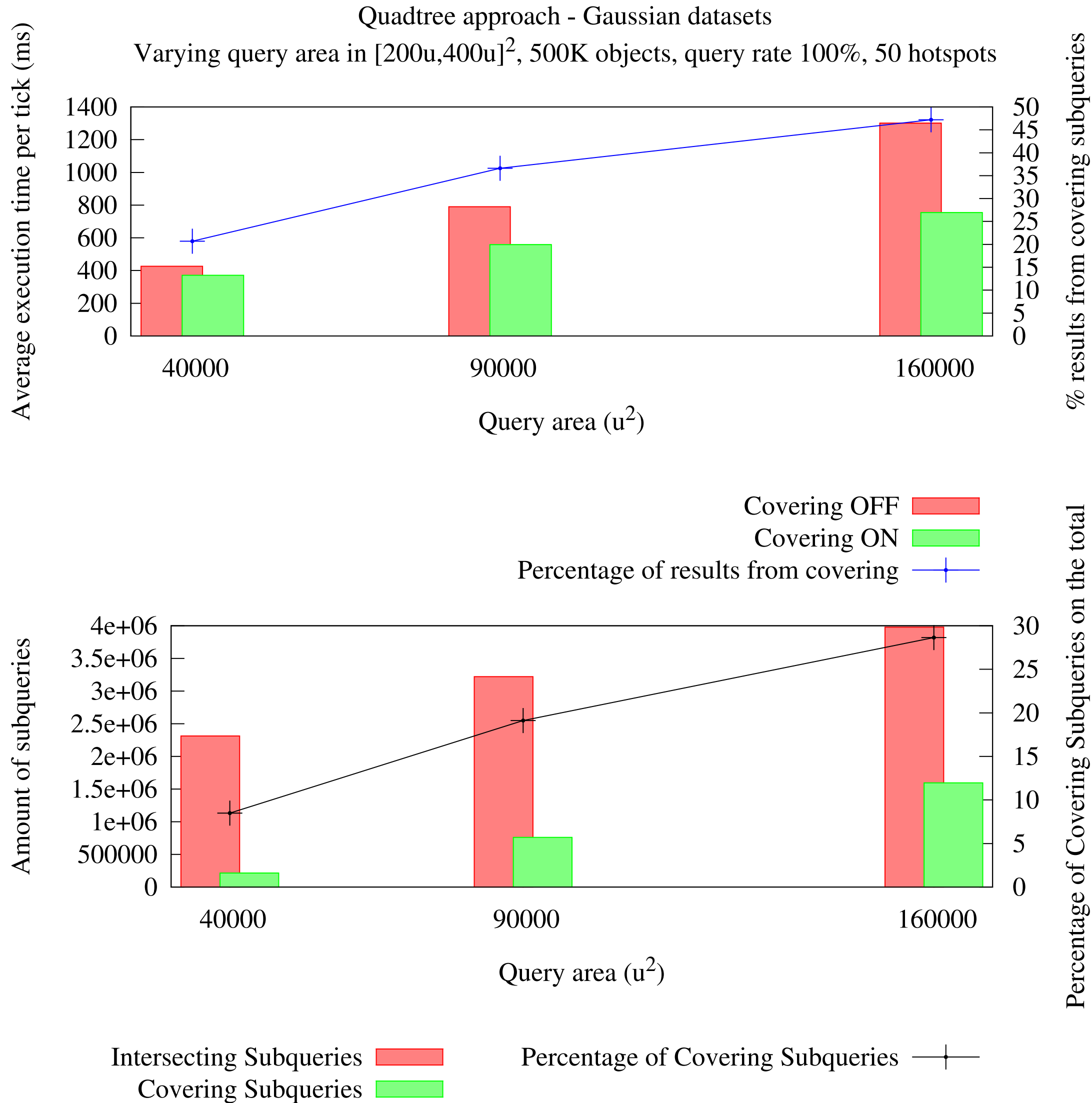


Figure 2.12: Gaussian datasets with 50 hotspots, 500K objects, query area varied in $[(200u)^2, (400u)^2]$, query rate 100%, Covering ON vs. OFF.

cution times of UG and QUAD. Note that in the case of UG the reordering entails higher performance improvements when the skewness gets large, while QUAD improvements are always moderate due to the ability of its underlying spatial indexing to dynamically produce tasks/blocks of similar weights.

We study in depth this behaviour by profiling the execution of the GPU SMs. In particular, we collect the per-tick amount of containment tests performed by each SM, and check whether the observed performance trends are reflected in workload unbalances among the SMs. In this context we define the *SM imbalance* measure during a single time tick as the *relative difference* between the highest amount of

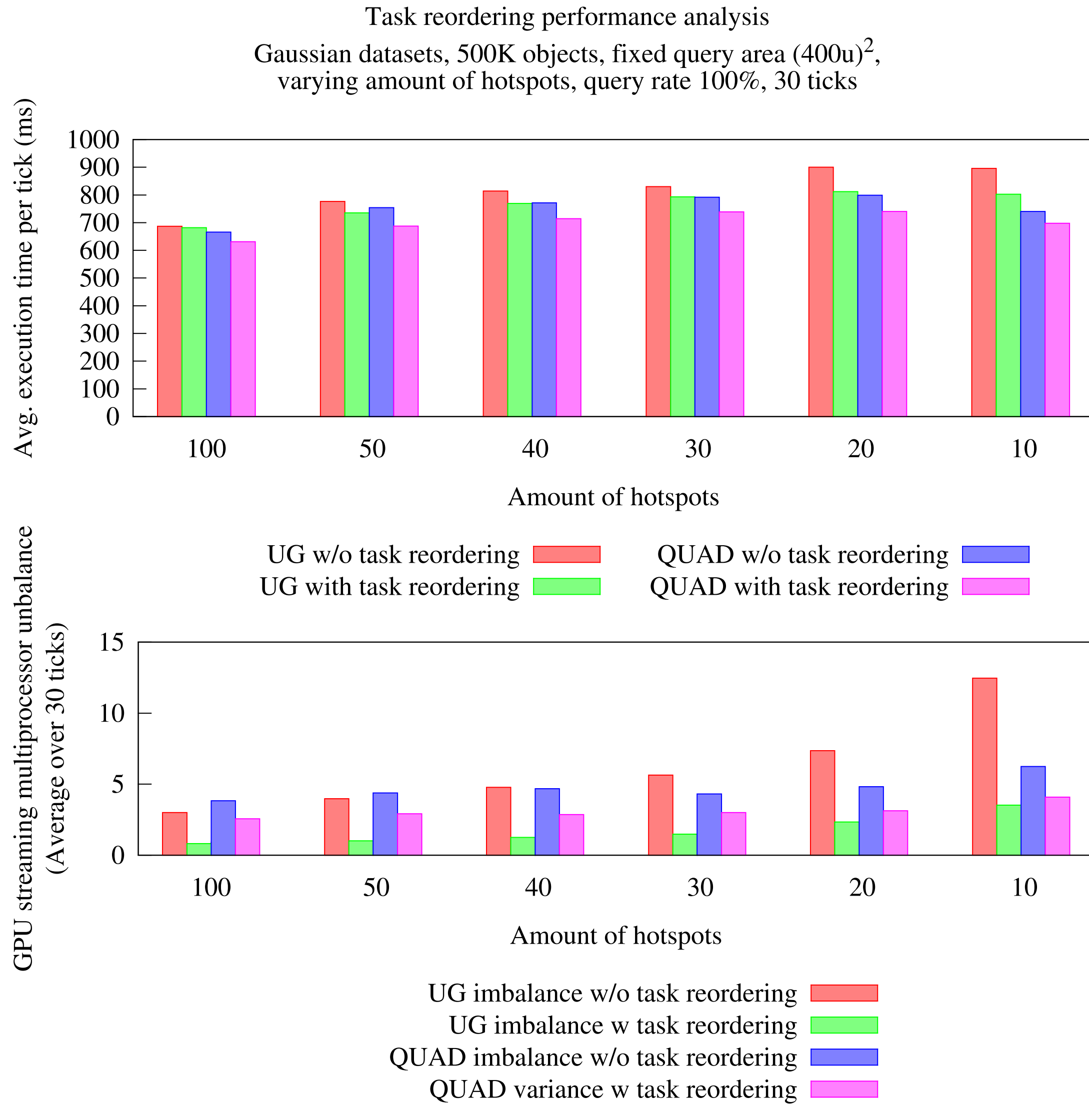


Figure 2.13: Analysis on the performances and workload redistribution among the GPU streaming multiprocessor with and without the static task list reordering - gaussian datasets, 500K objects, query area $(400u)^2$, query rate 100%, varying amount of hotspots. The top plot refers to the execution times observed while the bottom one refers to the profiling data collected during the filtering phase (decoding phase data is analogous).

containment tests performed by a single GPU streaming multiprocessor with the lowest amount performed by an another SM. Then, we compute the average of this measure across the ticks in order to characterize the average workload unbalance. From the bottom plot in Figure 2.13, we see how the trend of the *SM imbalance* follows the trend of the execution time, observed in the left plot of the same figure for both QUAD and UG. Hereinafter, all the experiments will be conducted by using the task list reordering optimization.

2.4.4 Data skewness and optimal grid coarseness for UG (S4)

The following set of experiments aims to show that the best coarseness used for the uniform grid onto which the UG spatial indexing relies depends on the specificities of the spatial distribution characterizing the objects at each tick. We therefore aim to show how it is not possible to find a unique optimal MBR *split factor* (i.e., the number of columns/rows in which the MBR is decomposed) that holds for all the datasets. Even more, we show how each pipeline phase has its own optimal MBR split factor given a single dataset.

We first focus on a gaussian dataset characterized by a mild skewness (150 hotspots), and study how the UG performance changes (during a single time tick) by varying the split factor. We decompose the overall execution time in three macro phases, namely the *indexing*, *filtering*, and *decoding* phases, where the former includes the *index creation*, *object/query indexing* and *sorting* phases (Sections 2.2.2 and 2.2.3). Figure 2.14 shows how the *indexing* time gets larger when we increase the MBR side split factor, as expected according to the costs described in Sections 2.2.2 and 2.2.3, due to the increase in the amount of subqueries created. As for the *filtering* phase, we see how the execution times trend exhibit a minimum. In general, too small split factors imply very large cells, few or none covering subqueries and potentially large workload unbalances, depending on the skewness. On the other hand, when the split factor is too large too many subqueries may be created, as well as there could be many active cells with small amounts of objects: this may represent a serious pitfall for an efficient usage of the memory/computational resources of a GPU. The same reasonings hold for the *decoding* phase as well. The overall execution time (the *Combined* bar in the plots) has a minimum obtained by using an optimal split factor equal to 110.

We replicate the same set of experiments with a consistently skewed gaussian dataset (Figure 2.15). The trends observed in Figure 2.14 are confirmed, although the UG optimal split factor value is different (95) due to the different dataset characteristics. This confirms that datasets having different spatial properties require the materialization of grids having different spatial characteristics in order to achieve the best possible performance.

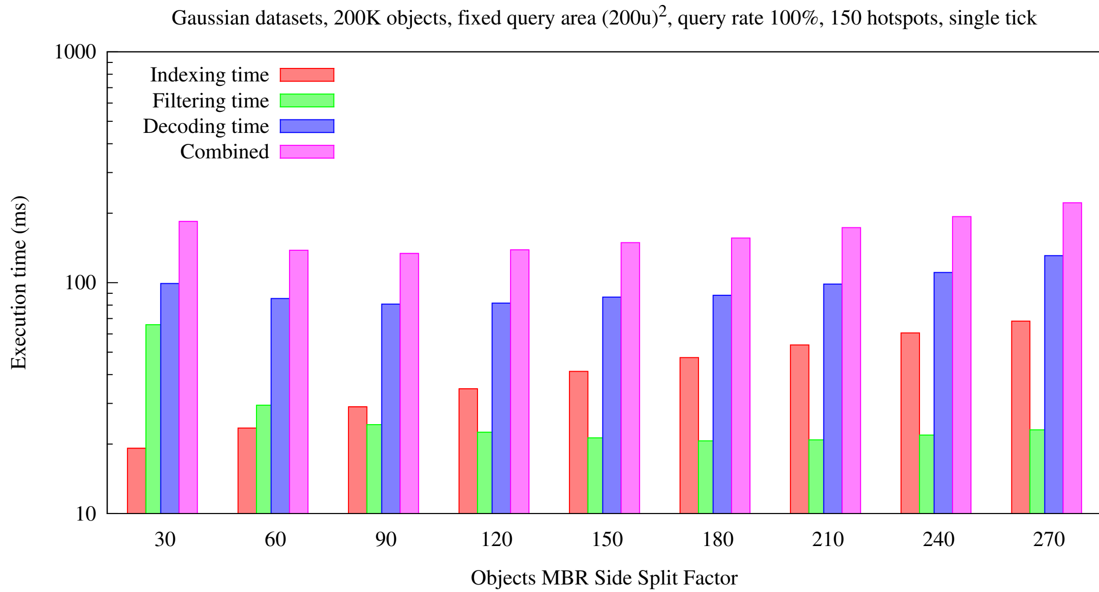


Figure 2.14: Gaussian dataset, 200K objects, query area $(400u)^2$, query rate 100%, 150 hotspots. The optimal value is equal to 110. Logscale on the y-axis is conveniently used to magnify small differences in the filtering execution times.

Figure 2.15: Gaussian dataset, 200K objects, query area $(400u)^2$, query rate 100%, 20 hotspots. The optimal value is equal to 95. Logscale on the y-axis is conveniently used to magnify small differences in the filtering execution times.

2.4.5 Data skewness and optimal cell size for QUAD (S5)

As already described in Section 2.2.3, in QUAD the size of the various cells is determined dynamically on the basis of data distribution and according to th_{quad}, a threshold determining whether a quadtree quadrant needs to be split at the next level according to the amounts of objects it contains at the time tick the quadtree is computed. Thus, we need to determine an optimal value for th_{quad} which hopefully does not change for datasets characterized by different object spatial distribution or query areas.

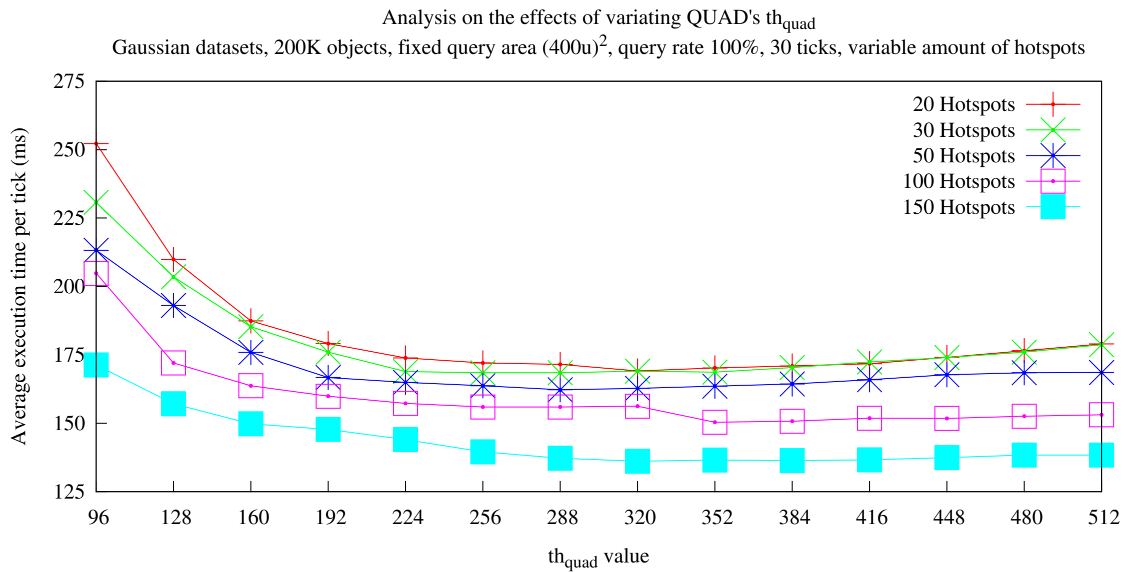


Figure 2.16: Performance analysis with different QUAD th_{quad} values when varying the skewness degree.

Figure 2.16 refers to a set of experiments in which, given a set of gaussian datasets with different amounts of hotspots, th_{quad} is varied in order to observe how QUAD behaves. The amount of objects characterizing each dataset is set to 200K, which allows the exploration of an extensive range of th_{quad} values: lower th_{quad} values increase the amount of resulting subqueries, which in turn increase the amount of GPU memory required for storing the subqueries. Figure 2.17 refers to a similar set of experiments in which the query area is varied among the datasets while the other characteristics are kept fixed.

In general we see how QUAD is resilient to dataset changes thanks to its low sensitivity with respect to th_{quad}, allowing an easy tuning of the system. Moreover, the search for an optimal th_{quad} is not so crucial, given the stability exhibited by QUAD for an ample interval of values. Increasing the query area has just the effect of increasing the execution times, while the trend remains the same for all the

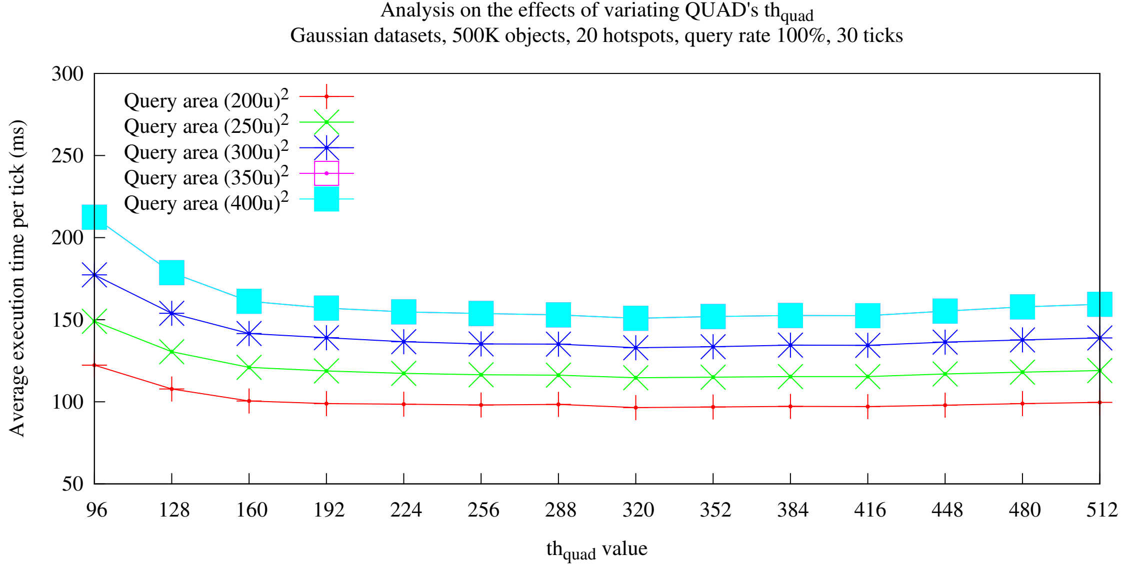


Figure 2.17: Performance analysis with different QUAD th_{quad} values when considering different query areas.

curves. Considering the results obtained above, in the experiments that follow we set $th_{quad} = 384$.

2.4.6 Impact of spatial distribution skewness on the performance (S6)

In this study we want to observe how UG and QUAD perform when varying the skewness degree by considering a set of gaussian datasets having different amounts of hotspots. In the experiments that follow we keep fixed the amount of objects (500K), the query area ($400u^2$) and the query rate (100%), whereas we vary the amount of hotspots in the $[10, 200]$ interval. For UG and QUAD we exploit all the optimizations, included the oracle used by UG (even though unusable in a practical setting).

Figure 2.18 shows that UG and QUAD have similar performances until the skewness becomes consistent, i.e., the amount of hotspots gets below 20. This is confirmed by the fact that QUAD is able to maintain stable and consistent speedups with respect to CPU-ST, even in presence of extremely skewed distributions, while UG slightly degrades.

We try to explain the observed performances in terms of the ability of UG and QUAD in redistributing the objects among the grid cells. To this end, we compute the *mean* and *variance* of the amount of objects in each grid active cell and the associated *dispersion index* ($D = \sigma^2/\mu$) characterizing the distribution of the objects

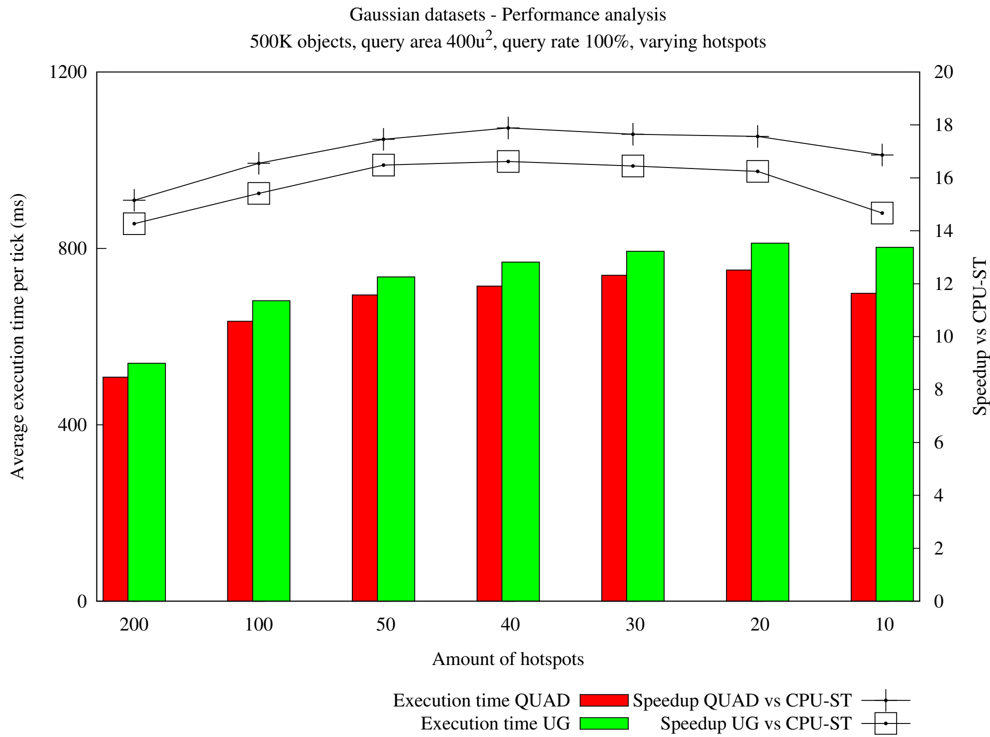


Figure 2.18: Gaussian datasets, 500K objects, query area $(400u)^2$, amount of hotspots varied in [10,200], average running times per tick and speedup against CPU-ST.

over the active cells. In general it is expected that, the finer a grid is, the lower the resulting mean and dispersion index are, although these figures are heavily influenced by the skewness characterizing the dataset. The mean and the dispersion indices obtained by UG and QUAD are shown in the top plot of Figure 2.19. UG always yields remarkably higher dispersion indices and lower means than the QUAD ones. The very low means observed for UG depend on the very fine uniform grid exploited, needed to avoid heavy populated cells which would entail very expensive tasks to be execute by a single GPU SM. Indeed, the ability of QUAD in properly redistributing workloads associated with objects living in densely populated regions is confirmed by the overall amounts of (intersecting/covering) subqueries produced (Figure 2.19, bottom graph) - amounts which are remarkably lower than the ones obtained by UG. For example, the amount of subqueries obtained when analyzing a very skewed dataset (such as the gaussian one with 10 hotspots) is about 12 millions for UG, and approximatively half for QUAD. Even if the size of the UG cells is very small, and thus the probability that a subqueries "covers" a grid cell gets large, the same plot shows that the proportion of results coming from covering subqueries in QUAD are almost on a par with the one obtained by UG. Finally, the remarkable smaller count of subqueries allows QUAD to have lower GPU memory requirements than UG when

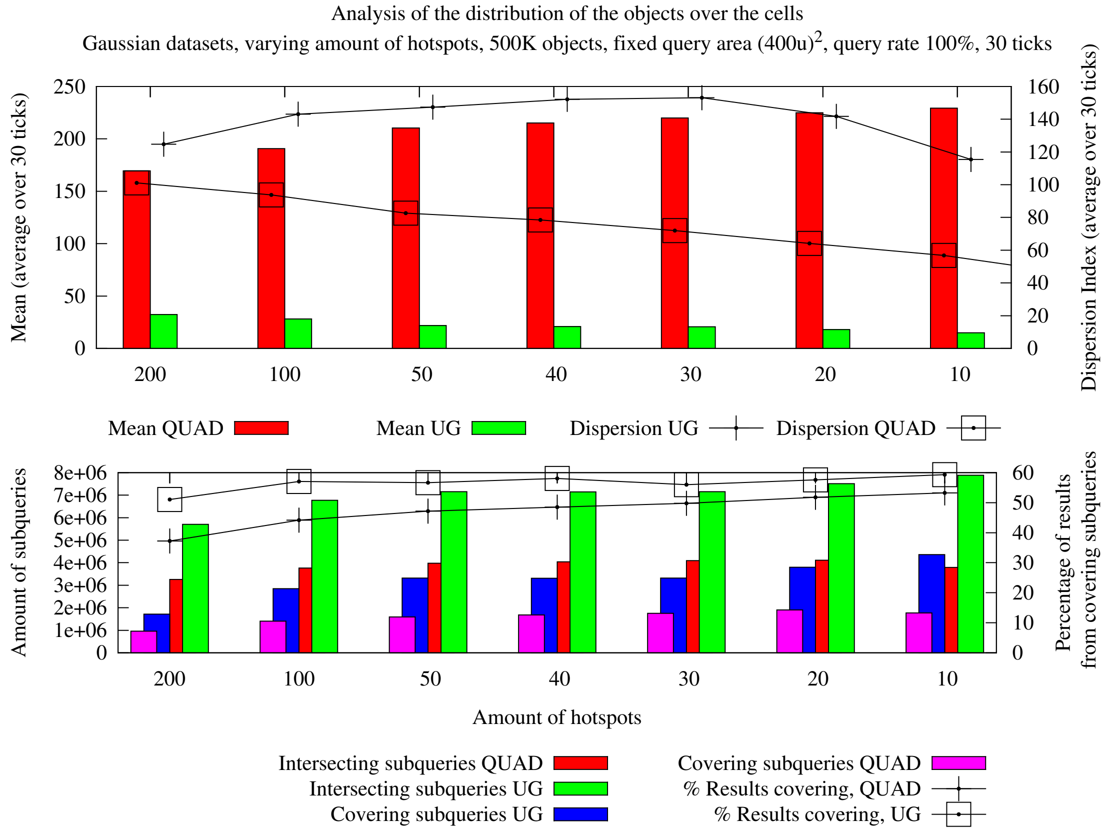


Figure 2.19: Gaussian datasets, 500K objects, query area $(400u)^2$, amount of hotspots varied in [10,200], mean and dispersion index over the grid active cells.

generating and computing the subqueries.

2.4.7 Performance analysis for different spatial distributions, amount of objects, and query areas (S7)

In this final study we analyze the performances of UG and QUAD with datasets characterized by different spatial distributions. In the experiments that follow we also vary the amount of objects and the query areas. For UG and QUAD we exploit all the optimizations. The goal is to show how QUAD is generally able to outperform UG, even if the latter relies on an expensive, and thus unfeasible, performance profiling (oracle) in order to select the best possible uniform grid coarseness for any dataset.

Variable amount of moving objects. In these experiments we exploit three types of datasets - uniform, gaussian and network-based - where we keep fixed the query rate and the query area at 100% and $(200u)^2$, respectively. For the gaussian

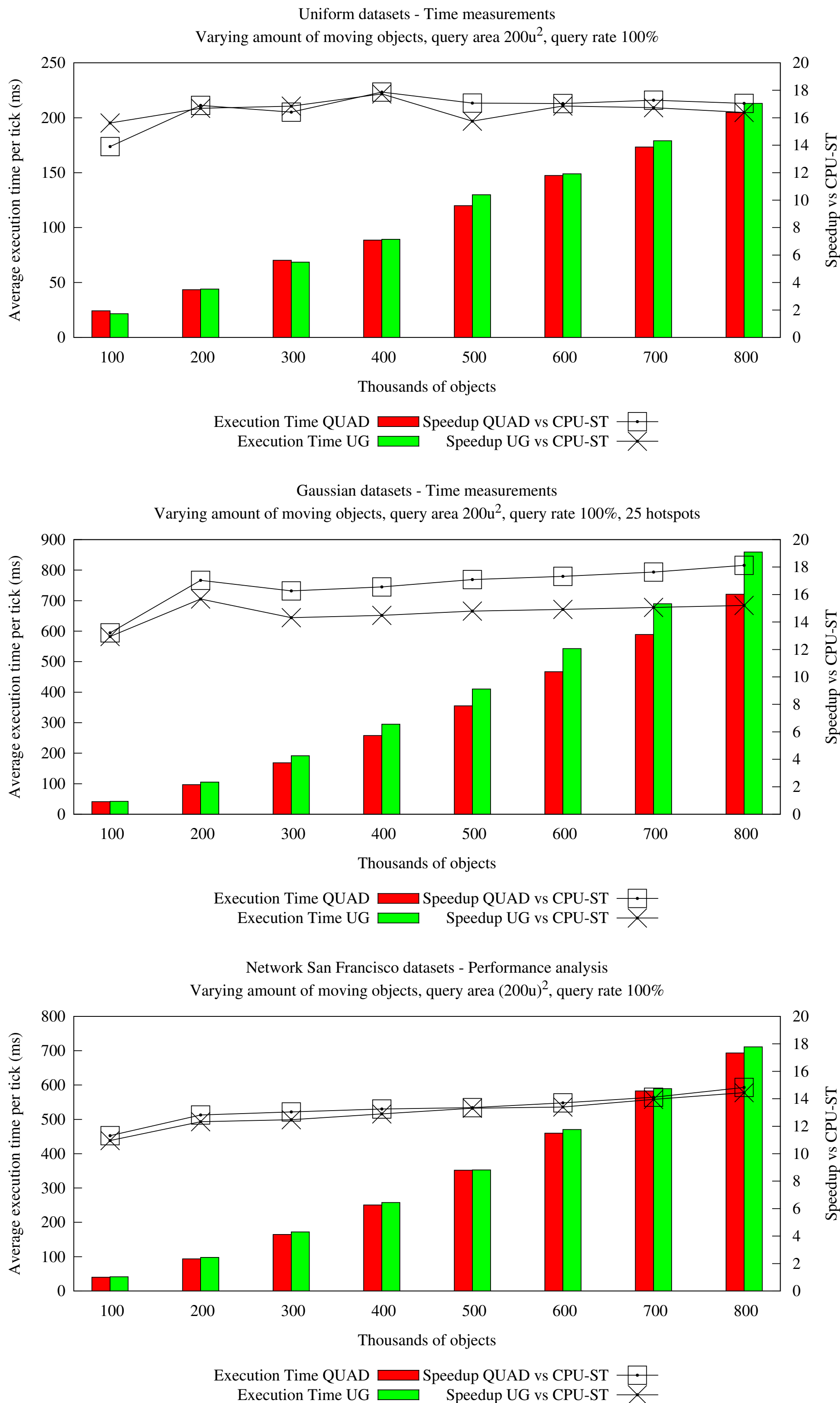


Figure 2.20: Varying the number of objects: average running time per tick and speedup versus CPU-ST. From top to bottom: uniform datasets, gaussian datasets with 25 hotspots, and San Francisco Network datasets.

datasets, the number of hotspots is fixed to 25. Figure 2.20 shows the execution times and the speedups versus CPU-ST for these three types of datasets when varying the amount of objects.

When uniform distributions are considered, UG and QUAD exhibit similar performances as expected. When gaussian datasets are considered, UG and QUAD exhibit stable and consistent performances, with QUAD performing noticeably better than UG. Finally, on network datasets UG and QUAD perform closely since these datasets are characterized by a very limited skeweness, with QUAD performing slightly better.

Variable query area. In this batch of experiments, whose results are shown in Figure 2.21, we vary the query area. All the queries are equally sized during a single experiment, while the amount of objects is fixed (700K for uniform, 500K for gaussian and network), as well as the query rate (100%) and the number of hotspots (25) for the gaussian datasets.

With uniform distributions UG and QUAD again perform similarly. With gaussian distributions, UG and QUAD maintain consistent performances, even though the advantage of QUAD over UG still holds. Finally, with network datasets UG and QUAD are almost on par, as already observed in the first batch of experiments, with a very slight advantage for QUAD.

Variable amount of objects and variable query area. In these experiments we consider different amounts of objects, each one issuing a query whose area is decided independently of the other objects and according to a uniform distribution in the $[(200u)^2, (400u)^2]$ range. For this experiments, we again consider uniform, gaussian (25 hotspots) and network datasets. In all cases the query rate is fixed at 100%.

When considering uniform datasets (Figure 2.22) UG and QUAD still exhibit similar performances. When considering gaussian datasets (Figure 2.23) UG and QUAD follow similar performance trends, with QUAD still performing noticeably better than UG. Finally, when considering network datasets (Figure 2.24) UG and QUAD exhibit again close performances, similarly to what is observed in Figures 2.20 and 2.21.

2.4.8 Bandwidth analysis (S8)

From lemma 2 in Section 1.1.4 we have that the system bandwidth β, expressed as the amount of queries processed per time unit (indeed, we use the second), is one of the crucial parameters in order to determine a suitable tick duration Δt, along with a given latency requirement λ and a maximum amount of queries which may occur during Δt, Q_{max}. Since λ and Q_{max} are fixed, the crucial parameter becomes β.

Consequently, the goal of this study is to observe how the bandwidth β of a given system reacts to a set of dominant factors, such as the *amount* of moving

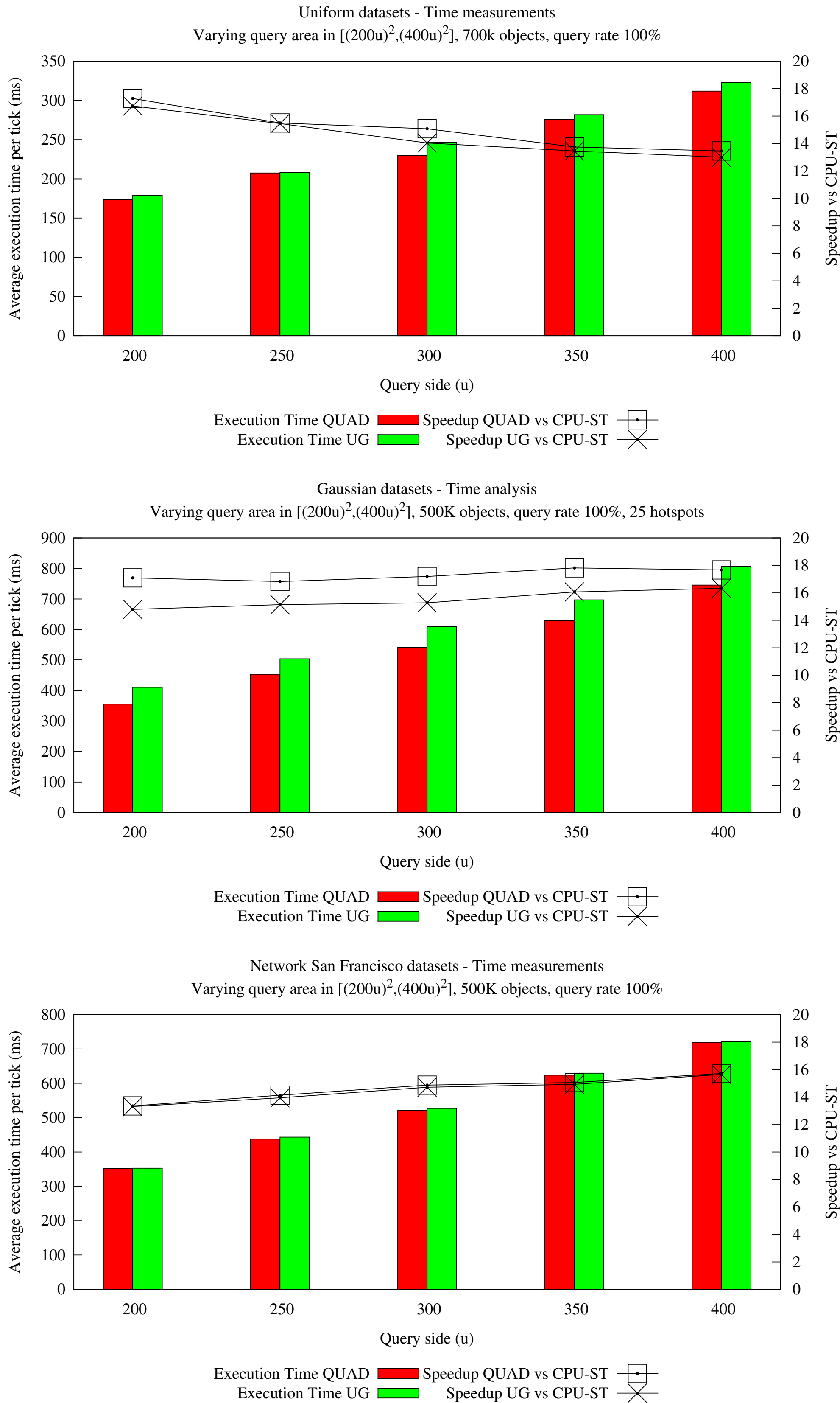


Figure 2.21: Varying the query area: average running time per tick and speedup versus CPU-ST. From top to bottom: uniform datasets, gaussian datasets with 25 hotspots, and San Francisco Network datasets.

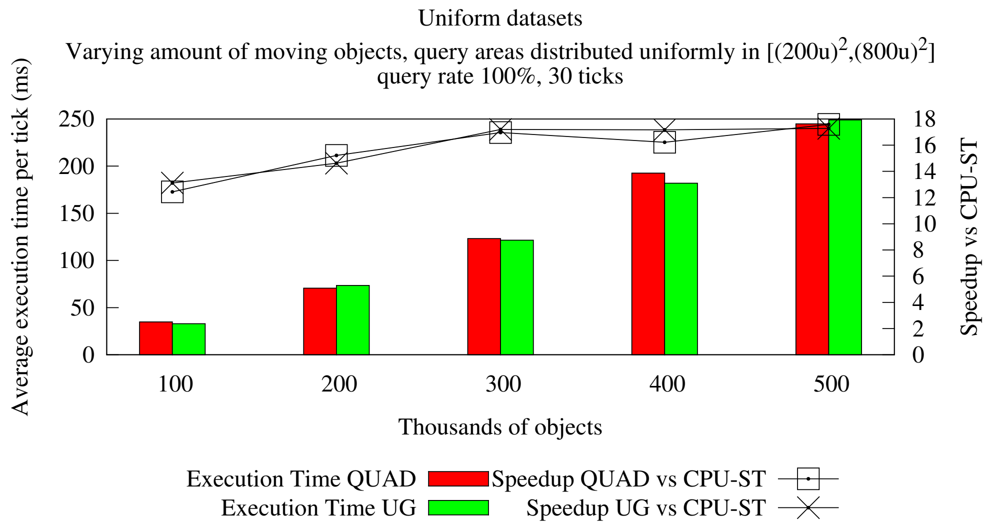


Figure 2.22: Variably sized queries: average running time per tick and speedup versus CPU-ST. Uniform datasets

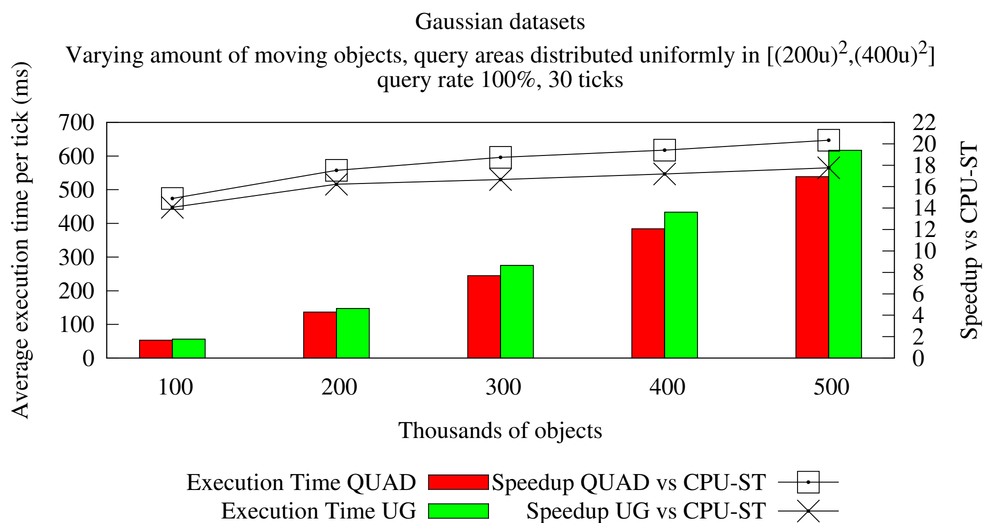


Figure 2.23: Variably sized queries: average running time per tick and speedup versus CPU-ST. Gaussian datasets.

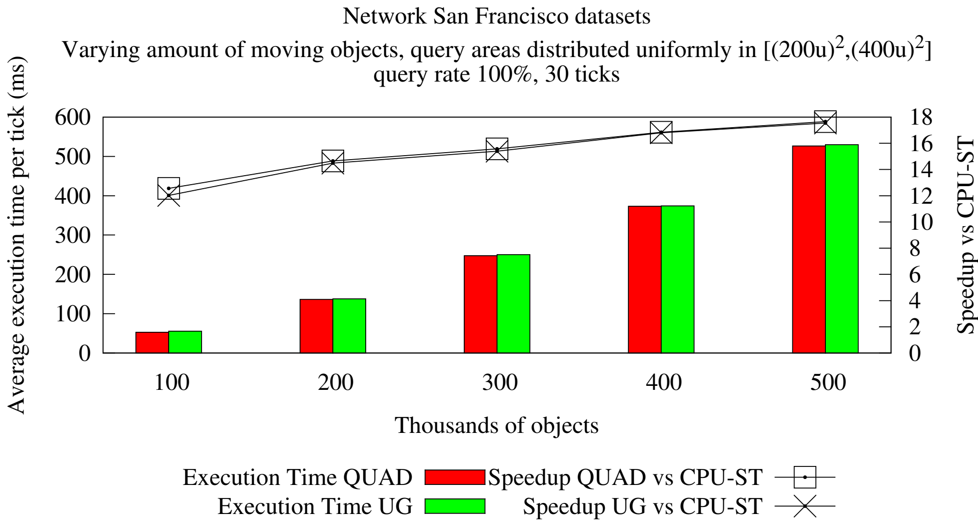


Figure 2.24: Variably sized queries: average running time per tick and speedup versus CPU-ST. Network datasets.

objects, the *query rate* (i.e., the factor of moving objects issuing a query during a time unit), the *query area*, and the *skewness*. QUAD will be used to conduct all the experiments.

Figure 2.25 presents the results of the first batch of experiments, where we test the behaviour of β with respect to different amounts of objects and degrees of skewness. In order to conduct these experiments a set of gaussian datasets were considered. From the Figure we see how the system bandwidth decreases whenever the amount of objects or skewness degree (ranging from uniform-like distributions - 10000 hotspots - to moderately skewed ones - 25 hotspots) increase, due to an increase in the overall amount of containment tests and results that the underlying system must handle in the same time unit. We also observe how highly skewed datasets produce the most notable negative consequences on the performance, thus requiring particular care.

Figure 2.26 reports the results related to the second batch of experiments, where we analyze the behaviour of β with respect to the query rate (we observe it corresponds to changing Q_{max}) and the query area. To this end we consider a set of uniformly distributed datasets characterized by different query rates and query areas. We observe that increasing the query area decreases the system bandwidth, due to a quadratic increase in the amounts of containment tests and results the system must handle per time unit. As regards query rate, we see how the bandwidth increases whenever this parameter is increased. Even if this phenomenon may seem counter-intuitive at first, we observe that the action of increasing the query rate has the effect of increasing linearly (and not quadratically) the amount of containment

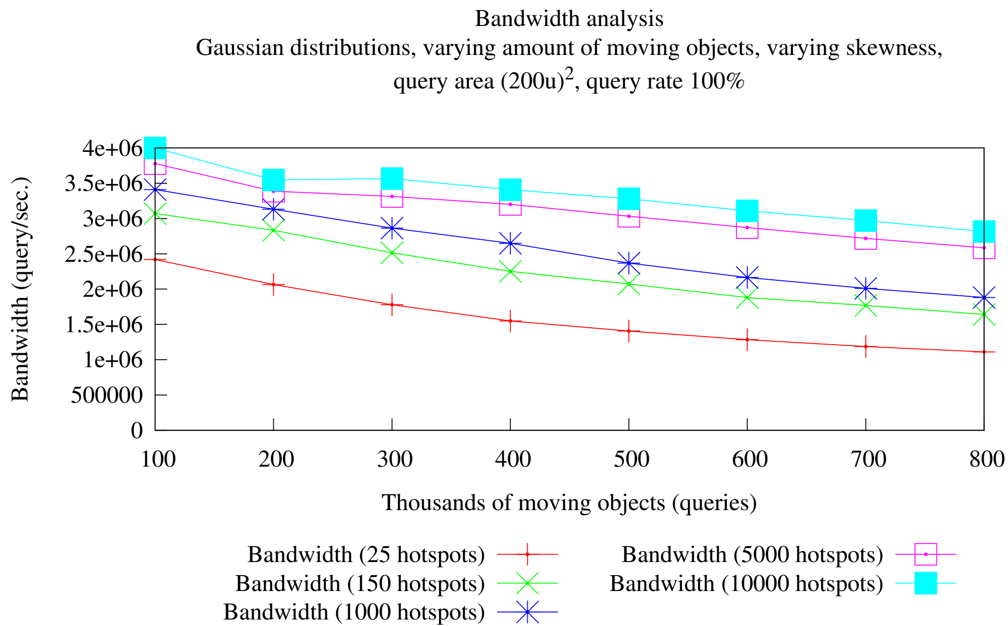


Figure 2.25: System bandwidth analysis when varying the amount of moving objects or the dataset skewness.

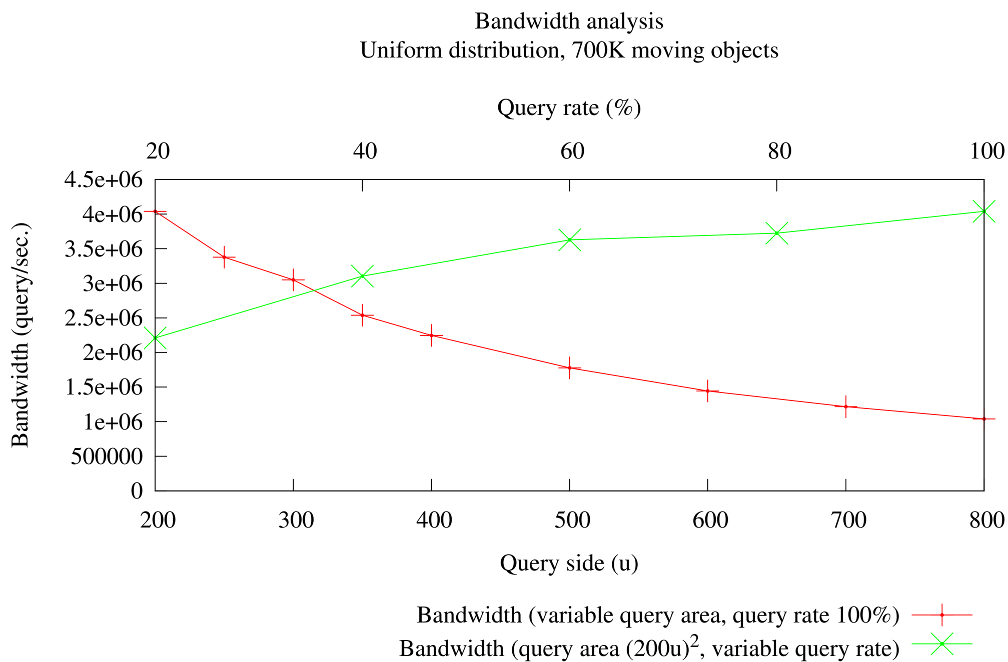


Figure 2.26: System bandwidth analysis when varying the query area or the query rate.

tests and results produced. These increases, however, are compensated by an increased efficiency of the system. That is, the GPU resources are more utilized and thus better exploited, and this in turn increases the overall bandwidth. We note that this behaviour can be replicated with any spatial distribution.

3

GPU-Based processing of repeated k-NN queries

In this chapter we address the problem of processing repeated k-NN queries over massive moving objects observations by means of an hybrid CPU/GPU approach. Computing k-NN queries poses different challenges with respect to the ones encountered with range queries, mainly stemming from the fact that the spatial extent of k-NN queries is not known beforehand.

As a starting point we use the framework introduced in Chapter 1 to model the processing. Starting from this, thanks to the flexibility of the processing pipeline introduced in 2.2.1 we reuse a relevant part of algorithms and data-structures introduced previously. However, customizations in the pipeline composition and processing flow are required in order to handle the k-NN query processing efficiently: on one hand we have to cope with the uncertainty bounded to the queries spatial extent, which has far reaching consequences on the processing pipeline; on the other hand we have to devise a clever strategy through which we materialize uniform workloads to be distributed across the GPU streaming multiprocessors. In reality, these issues represent two sides of the same coin, as we motivate later on. The end product of such effort will be represented by the $\mathsf{K\text{-}NN_{GPU}}$ approach.

The main contributions contained in this chapter can be summarized as follows:

- we use the processing framework described in Chapter 1 and build on the hybrid CPU-GPU pipeline presented in Chapter 2, partly reusing the operations introduced previously and partly customizing the pipeline's composition and execution flow, in order to devise an efficient and scalable approach for processing batches of k-NN queries, $\mathsf{K\text{-}NN_{GPU}}$.

- we introduce algorithms exploiting proper memory access patterns and key properties of elements sorted according to spatially preserving functions in order to benefit from coalescing and caching as much as possible. In this sense we show how careful algorithmic design choices allow to parallelize on GPU operations which are usually executed on CPU and represent major bottlenecks when designing hybrid CPU/GPU k-NN query processing pipelines.

- we carry out an extensive set of experiments in order to study how K-NN$_{\mathsf{GPU}}$ varies its performance with respect to run-time and dataset key parameters, and show how these parameters affect the system bandwidth as well. We also compare K-NN$_{\mathsf{GPU}}$ against a well-known GPU baseline, [35], as well as against a state of the art CPU sequential competitor, proving the effectiveness of our proposal.

The chapter is structured as follows: in Section 2.1 we give an overview about K-NN$_{\mathsf{GPU}}$ as well as on the main data structures used, while Section 2.2 details the k-NN query processing pipeline, where we outline two slightly different K-NN$_{\mathsf{GPU}}$'s variants as well, that is, K-NN$_{\mathsf{GPU}}^{CACHE}$ and K-NN$_{\mathsf{GPU}}^{COALESCE}$. In Sections 3.3 and 3.4 we provide the experimental part, where (i) we study how K-NN$_{\mathsf{GPU}}$ behaves when changing run-time or dataset key parameters, such as the nearest neighbours list size, k, the amount of objects, the amounts of queries issued per tick and the skewness characterizing the spatial distribution, and (ii) how such parameters affect the system bandwidth. Then, (iii) we assess the benefits of K-NN$_{\mathsf{GPU}}$ against a GPU baseline based on a quadratic brute-force approach [35] and (iv) how K-NN$_{\mathsf{GPU}}$ outperforms a state-of-the-art sequential CPU competitor.

As in the case of the previous chapter, we delegate the conclusions and possible directions of research to the conclusive chapter of the thesis.

3.1 K-NN$_{\mathsf{GPU}}$ overview

In the following we give an overview on K-NN$_{\mathsf{GPU}}$, motivating the main ideas used to address the challenges posed by the problem considered. Many inspiring principles and design choices introduced in Chapter 2 are reused here as well, since the processing framework is the same and the processing pipeline introduced for range queries can be adapted fairly easily to k-NN queries. However, the fact that the spatial extent of k-NN queries is not known beforehand has far-reaching consequences on the pipeline's composition and execution flow: this issue will mainly drive the design of K-NN$_{\mathsf{GPU}}$.

3.1.1 Motivating challenges

Computing repeated k-NN over massive moving object observations from the GPGPU perspective may seem, at first, easier than computing repeated range queries, since the result set size is unknown in the latter case. However, such indeterminateness just shifts to the spatial extension of the queries in the k-NN case, since it depends on local objects densities.

The usual approach to reduce the amount of computations per query is to adopt an index based on some tree. In order to compute a k-NN query one then has to perform a recursive tree visit, exploring only parts of the tree corresponding

to regions possibly enclosing nearest neighbours. Obviously, such visit must be performed in a clever way, minimizing both the amount of visited leaves and the amount of computations per query: this can be achieved by selecting a proper starting leaf (i.e., the leaf containing the query center), then expanding the visit by considering only those nodes/leaves whose spatial extension may contain nearest neighbours.

Since the spatial extension of each k-NN query is unknown, different queries possibly require to visit different paths inside the tree or different amounts of leaves. Moreover, depending on the kind of tree used, each leaf possibly contains different amounts of objects with respect to other leaves, thus strengthening the challenge of materializing uniform GPU workloads. In other words, the problem is to find a way to batch enough work per GPU streaming multiprocessor while entailing uniform workloads. Apart from the challenges mentiond above, such problem has profound consequences even on the layout used to arrange the result set.

Our approach tries to tackle these issues by adopting an iterative approach while reusing and properly customizing the processing pipeline introduced for QUAD (Section 2.2.1). The presentation of the pipeline is postponed to Section 3.2 since it requires in-depth explanations; in the following we give an overview about the main data-structures used throughout the pipeline.

3.1.2 Relevant data structures

In this section we review the main data structures used during the query processing.

3.1.2.1 General overview

Data structures containing objects or queries information use the *structure of vectors* (SoV) layout, as done previously in the range queries case (see Section 2.1.4). As already stated, the SoV layout gives remarkable benefits when designing GPU algorithms, above all code reuse and efficient interplay between different operations carried on GPU. Moreover, it facilitates the exploitation of data locality and the use of coalescing or caching, whenever possible, thus offering substantial chances to boost the overall memory throughput, an aspect of paramount importance when designing GPU algorithms.

3.1.2.2 k-NN queries result set layout

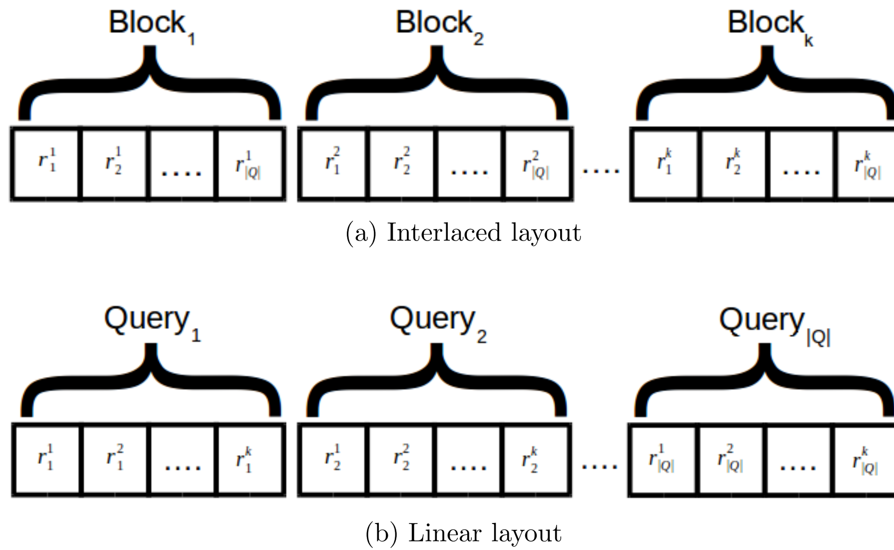
Some words have to be spent about how query results are produced and managed during the processing. Unlike the range queries case, we do not need to recur to bitmaps since the result set of each k-NN query (from now on also denoted as *nearest neighbours list*) has fixed size (depending on k). As such, one would be tempted to arrange the lists by means of an interlaced layout, writing out in global memory the results in blocks - similarly to the interlaced format used for bitmaps (Section

2.1.4.2). An example of this layout is depicted in 3.1a. Indeed, this layout yields the maximum memory throughput when using a brute-force approach, such as the one proposed in [35], since the amount of distances to be computed per query would be predetermined.

However, given that we use an index-based approach we have to update the k-NN query lists as the tree visits progress, since we do not know beforehand which and how many leaves we have to visit per query. This implies that different queries possibly require to consider different amounts of leaves - due to different paths followed during tree navigation - in order to correctly compute the final results. This, in turn, requires to adopt an iterative approach which batches conveniently the workload resulting from such visits, since we want to exploit the GPUs computational power.

However, adopting the aforementioned interlaced layout with a tree-based iterative approach would frustrate the benefits deriving from coalescing, since *inactive* queries, that is, queries which have terminated their computation with respect to others that require further iterations, would create "holes" (this is quite evident from Figure 3.1a) when writing out the data. Also, benefits deriving from caching would be heavily frustrated for the same reasons.

The only way to fully use coalescing by employing a result set interlaced layout would be to sort queries according to their *active/inactive* status; however, the amount of time needed to rearrange queries would outweigh the benefits deriving from coalescing, since the amount of data to be moved may be quite relevant. As a consequence, this option must be ruled out.



(a) Interlaced layout

(b) Linear layout

Figure 3.1: Interlaced and linear result set layouts. In the Figure we denote the j-th result of query q_i as r_i^j.

For this reason we rely on GPU caching capabilites by arranging the queries lists

linearly, as depicted in Figure 3.1b: this allows to exploit caching capabilities until k does not get too large; moreover, in tandem with the GPUs ability in hiding the latencies related to memory operations (up to a certain extent), this layout allows to achieve good memory throughputs. Finally, at the cost of a slightly increased computational overhead it is possible to update the query lists by means of access patterns yielding coalesced accesses, a strategy which may be quite convenient when k gets large. We motivate better the benefits deriving from the linear layout in Section 3.2.2.

3.2 Processing Pipeline

When computing k-NN queries issued during a time tick, the core operation is represented by the computation of distances between objects locations and queries centers. As in the range queries case, this apparently simple and straightforward operation is embedded in a more complex pipeline of concatenated operations in order to improve its efficiency.

Since the respective problem settings are very similar, range and k-NN queries share many operations inside the processing pipeline; as a consequence, many concepts and design choices can be reused.

Differently from range queries, however, k-NN queries pose serious challenges due to the fact that the related workload cannot be determined a-priori since their spatial extent is unknown. Since we rule out the usage of brute-force approaches, choosing instead to rely on some kind of spatial index, we need to compute k-NN queries by means of an iterative schema. This in turn requires to find an approach able to create partial GPU workloads on the fly, as the computation of the queries progresses.

In the following we introduce the K-NN_{GPU} approach for processing batches of k-NN queries. K-NN_{GPU} employs the framework described in Section 1.1.1 and conveniently adopts a tick-based processing pipeline derived from the one used by QUAD (Section 2.2.3). K-NN_{GPU}'s pipeline can be described in terms of a succession of three macro phases: (i) *index creation*, (ii) *moving objects indexing* and (iii) *iterative query processing*.

When processing repeated k-NN queries, the same procedure is repeated for each tick. Thus, for the sake of readability, hereinafter we omit the subscript that indicates the tick, and denote by P, Q, and R, respectively, the up-to-date object positions, the non-obsolete queries, and the result set associated with a generic tick.

3.2.1 Index Creation and Moving Objects Indexing.

In this Section we briefly review the *index creation* and *moving objects indexing* phases, since these are equivalent to the ones described in Section 2.2.3.

3.2.1.1 Index Creation.

The index used to partition the space is based on a point-region quadtree built over a given set of objects. Observing that, most of the times, space distributions do not change their characteristics dramatically over short time intervals, the index is rebuilt only whenever needed, for example when we detect that the overall amount of computations yielded during the last tick exceeds by a given factor the amount of computations yielded during past, recent ticks. The overall complexity of this phase can be approximated to $O(|P|)$.

The set of cells related to the materialized index (i.e., the quadtree leaves) will be denoted by $\mathcal{C}$ from now on.

3.2.1.2 Moving objects indexing.

During this phase we have to assign each moving object to a specific quadtree leaf. Also, we want to sort moving objects according to the leaf in which they fall, in order to determine the set of active cells, that is, the set of cells enclosing at least one object. This phase is equivalent to the *moving objects indexing* phase already described in Section 2.2.3, even though limited to moving objects since queries are possibly assigned to more than one leaf, due to the fact that their spatial extent is unknown. Here we briefly review the procedure.

Leaf assignment. At this stage, each moving object has to be mapped to a specific grid cell, according to the function f introduced by Definition 9 (Section 2.1.3.3). More precisely, for each object it is first determined the cell in the uniform grid - related to the quadtree deepest level - in which the object falls; subsequently, the identifier of the cell is associated with the object. Next, objects are sorted according to cell identifiers in order to exploit the GPU caching capabilities when accessing the inverted index z_{map}. Finally, each object is associated - through z_{map} - with the quadtree leaf covering the cell to which they were assigned, i.e., each object is associated with a pair (l, z) [1] representing the quadtree leaf in which the object location falls. The complexity of the indexing phase is $O(|P| + d \cdot |P| + |P|)$: the first term is related to the computation of Morton codes, the second term is due to sorting - performed by means of Radix Sort, while the third term is related to the accesses in z_{map}. Since the complexity of the sorting operation can be approximated to $O(d \cdot |P|) \approx O(|P|)$, the overall complexity of this subphase can be approximated to $O(3 \cdot |P|)$.

Sorting. Objects come already *sorted*, according to the identifier of the cell in which they fall, from the leaf assignment subphase described above. As a conse-

[1] We remember that l represents the level of the quadrant while z its Morton code. Moreover, we remember that such pair can be conveniently represented through an integer, thus allowing to easily sort the objects according to assigned cells.

quence, moving objects falling inside the same cell are arranged in contiguous memory blocks, thus favouring caching, by means of spatial locality, when computing the distances during the *Iterative k-NN queries computation* phase of the pipeline (Section 3.2.2).

Active cells materialization. Finally, the sorted struct of vectors is *indexed* so to determine the positions of the first and last object belonging to index cells enclosing at least one object. We define this set as the set of *active cells* and we denote it by $\hat{\mathcal{C}}$ (where $\hat{\mathcal{C}} \subseteq \mathcal{C}$).

The complexity of this operation is equal to $O(2 \cdot |P| + |\hat{\mathcal{C}}|)$: the first term is due to the double scan over the set of objects needed to detect the discontinuities between objects belonging to different cells, while $|\hat{\mathcal{C}}|$ represents the amount of active cells for which we actually have to write out the related indexing information. We note that $|\hat{\mathcal{C}}| \ll |P|$, so the complexity may be approximated to $O(2 \cdot |P|)$.

3.2.2 Iterative k-NN queries computation

Once the set of active cells is determined, the actual query computation may start. Since we rely on a PR-quadtree based spatial index, for each query we have to perform a *recursive tree visit* in order to compute the correct list of k nearest neighbours. Considering that our goal is to exploit the GPUs computational power we obviously need to devise an effective way to create GPU workloads on-the-fly, as tree visits progress, which fits well the GPUs architectural peculiarities. More precisely, (i) workloads should be distributed as much evenly as possible across the GPU streaming multiprocessors and (ii) objects, queries and results data should be arranged so that *coalescing* and *caching* are exploited as much as possible. The idea is to achieve these goals by means of an *iterative* approach, exploiting spatial proximity between nearby queries as visits progress.

We conveniently distinguish between the operations carried on during the first iteration and those carried on during subsequent iterations, since this facilitates the description and the implementation of our proposal.

3.2.2.1 First iteration

The first iteration has to orchestrate the work associated with the beginning of the queries recursive tree visits. This corresponds to associating each query with the cell (quadtree leaf) in which its center falls and subsequently computing the distances between the query center and the objects enclosed by the cell. The first iteration ends by updating the queries nearest neighbours lists according to the distances computed.

We note that the operation of associating each query with a cell yields a set of tasks, one per active cell having at least one query to process, which represents the foundation through which we orchestrate distance computations carried on GPU.

We conveniently structure the first iteration in two smaller phases: (i) *query indexing and task materialization* and (ii) *distance computations* between queries and objects.

3.2.2.1.1 Query indexing and task materialization. Since each k-NN query is represented by a location, queries must be mapped to specific grid cells according to the function f introduced by Definition 9 (Section 2.1.3.3). Consequently, this subphase is equivalent to the moving objects indexing phase described in Section 3.2.1.2).

If we denote by $\overline{\mathcal{C}}$ the set of cells having at least one query assigned, where $\overline{\mathcal{C}} \subseteq \mathcal{C}$ and $\overline{\mathcal{C}}$ is possibly different than $\hat{\mathcal{C}}$, the complexity of such operation can be expressed as $O(5 \cdot |Q| + |\overline{\mathcal{C}}|)$.

Clearly, we have to consider only those cells which are actually active cells and have at least one query assigned. To this end, if we define $\overline{\mathcal{C}} = \overline{\mathcal{C}} \cap \hat{\mathcal{C}}$, we have that each $\overline{\mathcal{C}}$ cell implicitly carries with itself the computations needed to determine, for each query, the set of (up to) k nearest objects inside that cell. Such cells determine the tasks to be assigned to the GPU streaming multiprocessors.

3.2.2.1.2 Distance computations. At this point the goal is to compute, for each query, the list of (up to) k nearest objects within the assigned cell. We saw how the end product of the query indexing and sorting operations consists of a set of tasks, one per active cell enclosing at least one object, representing the GPU workload in charge of computing such lists. This approach allows to conveniently pack together computations related to spatially nearby entities, aiming to reduce the overall amount of computations and to exploit data locality. The challenge is therefore to orchestrate the computations inside each task cleverly, especially when it comes to maximize the memory throughput.

In order to reach this goal our approach basically relies on two pillars. The first one is represented by a *k-selection* algorithm based on buckets, such as the one described in [48]. Starting from a set of objects, this algorithm allows to find the k nearest objects without having to explicitly store and sort the distances in memory, thus reducing the overall complexity in terms of time and space. This is better than strategies based on distance sorting [49] or based on the maintenance of priority queues (one per query; on this matter the reader may refer to [49] as well). The second pillar is represented by a proper access pattern which allows to maximize the memory throughput when updating the queries nearest neighbours lists. Considering the linear layout used for the queries result set (Figure 3.1b), different strategies may represent the best choice, depending on the requested amount of nearest neighbours per query. To this end, in the following we introduce two different write strategies: the first one relies only on GPU caching capabilities, and is expected to give the best performances when k is low; the other one exploits coalescing capabilities as well, so to tackle effectively those cases where k is high.

Distance computations – *cached writes* approach. In this paragraph we focus on a strategy which relies only on GPU caching capabilities when updating the queries nearest neighbours lists. Algorithm 5 reports such strategy.

Algorithm 5: $distComp(\overline{C}, Q, P, k)$

Input : The set of active cells with at least one query, $\overline{C}$.
The reordered query set Q and object set P, along with the indexing information associated after the respective sorting phases.
The size of the queries neighbours lists, k.

Output: The struct of vectors containing the query result set, $(ID, DIST)$.
The vector containing the maximum distance detected for each query, $MAXDIST$.
The vector containing the amount of nearest neighbours found for each query, $NUMRES$.

```
1  begin
2    foreach c ∈ C̄ parallel_block do
3      foreach q ∈ c parallel_thread do
4        local (dist_min, dist_max) ← findMinMaxDist(q, c)
5        local dist_k ← findKDist(q, c, dist_min, dist_max, k, 32)
6        local i ← 0, maxdist ← 0
7        foreach p ∈ c do
8          if ((dist(p, q) < dist_k) ∧ (q_id ≠ p_id)) then
9            maxdist ← max(maxdist, dist(p, q))
10           (ID_q[i], DIST_q[i]) ← (p_id, dist(p, q))
11           i ← i + 1
12       MAXDIST_q = maxdist
13       NUMRES_q = i
```

Each task is assigned to a specific streaming multiprocessor (line 2) and executed according to a per-query parallelization (line 3). Each thread first loads the query information (line 3), i.e., query coordinates and the associated cell identifier. In this context, let c be the cell in which the query falls. Since queries belonging to the same cell are stored in contiguous memory locations, accesses to query data are *coalesced* during the first iteration.

Subsequently, each thread finds out the minimum and maximum distance ($dist_{min}$ and $dist_{max}$ respectively) between the query center and the objects within the cell (function *findMinMaxDist*, line 4). This is achieved through a simple scan over the set of objects enclosed by c. Since every thread in a warp perform such scan by accessing objects data in the same order, and considering that different warps in a block access nearby objects, this access pattern is able to effectively exploit the GPU *caching* capabilities.

Once $dist_{min}$ and $dist_{max}$ are determined, the algorithm goes on by computing a distance below which *only* the k nearest objects within the cell are located with respect to the query center. We denote such distance by $dist_k$ and is determined

by calling the *findKDist* function (line 5). *findKDist* implements the k-selection algorithm based on buckets, iteratively going on until a suitable $dist_k$ is determined. We note that whenever the amount of objects in c is less than k, the function can immediately return $dist_k = +\infty$ without performing any computation. Even inside *findKDist* threads within the same warp access c's objects in the same order; since different warps access objects arranged in nearby memory locations, this entails again an efficient usage of GPU caching capabilities. *findKDist* is detailed in Algorithm 6 and explained thoroughly below.

Once $dist_k$ is determined, each thread can actually start writing the list of nearest neighbours associated with the query. Such operation consists essentially in a scan over the set objects in c (line 7) where an object is copied in the query list only if its distance with respect to the query center is less than $dist_k$, and the identifier of the object is different from the query one. We recall that query lists are implemented as structures of vectors, where results are stored in global memory on the basis of the linear layout shown in Section 3.1.2. As a consequence, each thread stores the nearest neighbours it finds according to such layout. The thread terminates once it writes out the distance of the farthest object in the list, $MAXDIST_q$, and the actual amount of results written in the list, $NUMRES_q$ (lines 12 and 13), both stored in global memory. We note that such writes are coalesced, thanks to per-query parallelization.

We observe that in Algorithm 5 only private registers (i.e., the variables declared as **local**) and global memory are used, for what relates the memory hierarchy. Also, given that threads write out query lists by using the linear format, we claim that GPU caching capabilities, in tandem with the GPUs ability to hide latencies related to memory accesses, are able to achieve consistent memory throughputs, at least until k does not get large. This claim is verified in the experimental part of this chapter (Sections 3.3 and 3.4).

We now analyze *findKDist*, whose pseudocode is reported in Algorithm 6. The function, which is invoked by *distComp*, searches for a suitable $dist_k$ by adopting a simple, yet GPU-friendly, *k-selection* bucket-based algorithm, such as the one shown in [48]. More precisely, *findKDist* carries on an iterative process (line 5) which focuses, at each iteration, on the bucket containing the k-th nearest neighbour. Initially, the algorithm considers the interval in which all the distances fall, i.e., $[dist_{min}, dist_{max}]$, and divides it in *numBins* equi-width buckets. Then, the function updates the counters of the buckets according to the objects falling inside them (lines 8-12) and subsequently determines the bucket in which the k-th nearest neighbour falls (lines 15-21): if the k-th nearest neighbour represents the bucket's last element as well, then the upper-bound of the interval associated with the bucket represents also the distance below which *only* the k nearest objects fall with respect to the query center (in such case the cycle terminates). Otherwise, the algorithm focuses on such bucket (lines 23-24) and splits it in *numBins* equi-width buckets, thus iterating the schema. We note that the cycle may terminate even when $dist_{min} = dist_{max}$ (line 25): such event occurs when we have a zero-width bucket induced by a set of equal

Algorithm 6: $findKDist(q, c, dist_{min}, dist_{max}, k, numBins)$

Input : The query of interest, q.
The considered cell, c.
The range $[dist_{min}, dist_{max}]$ considered by the k-selection algorithm.
The amount of nearest neighbours, k.
The amount of bins used to split an interval, *numBins*.
Output: The k-th distance upper-bound.

```
1  begin
2      local counterBins[numBins]
3      local found ← false
4      local runningSumIterations ← 0
5      while ¬found do
6          counterBins ← [0, 0, ... , 0]
7          binWidth ← (dist_max − dist_min)/numBins
8          foreach p ∈ c parallel_block do
9              dist_p = dist(q, p)
10             if (dist_p ≥ dist_min) ∧ (dist_p < dist_max) then
11                 indexBin ← (dist_p − dist_min)/widthBin
12                 counterBins[indexBin] ← counterBins[indexBin] + 1
13         local runningSum ← runningSumIterations
14         local i = 0
15         while i < numBins do
16             runningSum ← runningSum + counterBins[i]
17             if runningSum ≥ k then
18                 if runningSum = k then found = true
19                 runningSum ← runningSum − counterBins[i]
20                 break
21             i ← i + 1
22         runningSumIterations ← runningSum
23         dist_min ← dist_min + (i ∗ widthBin)
24         dist_max ← dist_min + widthBin
25         if dist_min = dist_max then found = true
26     return (dist_max)
```

distances where the k-th nearest neighbour falls in - an event which is quite rare anyway. This case is signalled in the function's output and handled appropriately by the invoking function (omitted for brevity). We note that even *findKDist* relies only on global memory and private registers.

Distance computations – *coalesced writes* approach. This approach, which is almost equivalent to the one presented above except for the access pattern used to write out queries results, relies on a *per-warp* parallelization strategy when updating the queries nearest neighbours lists in order to exploit coalescing. Indeed, we

claim that relying only on GPU caching capabilities becomes unsatisfying in terms of performances when k starts to get large. Algorithms 7 and 8 outline this strategy.

Finding a distance below which only the k *nearest objects fall* - To begin with, we have to find out for each query a suitable distance, $dist_k$, below which *only* the k nearest objects fall inside the enclosing cell (Algorithm 7) with respect to the query center. This is equivalent to the first part of Algorithm 5, so we refer the reader to the associated presentation.

Algorithm 7: $distCompPhase1(\overline{C}, Q, P, k)$

Input : The set of active cells with at least one query, $\overline{C}$.
The reordered query set Q and object set P, along with the indexing information associated after the respective sorting phases.
The size of the queries neighbours lists, k.

Output: The vector containing, for each query, the distance of the farthest nearest neighbour found so far, $MAXDIST$.
The vector containing the amount of nearest neighbours found for each query so far, $NUMRES$.

```
1 begin
2   foreach c ∈ C̄ parallel_block do
3     foreach q ∈ c parallel_thread do
4       local (dist_min, dist_max) ← findMinMaxDist(q, c)
5       local dist_k ← findKDist(q, c, dist_min, dist_max, k)
6       local i ← 0, maxdist ← 0
7       MAXDIST_q = dist_k
8       NUMRES_q = min(k, |{P ∩ c}|)
```

Nearest neighbours lists update - Once we have determined a suitable $dist_k$ for each query, we can actually start updating the queries lists. At this stage the key idea is to parallelize the computation at warp level, by assigning each query to a warp. In turn, inside each warp we parallelize distance checks at thread level, while writes related to lists updates are cooperatively orchestrated at warp level, thus allowing to exploit coalescing when flushing out data in global memory. It is evident that such cooperative strategy requires the usage of a temporary buffer stored in shared memory, coupled with proper management operations. This strategy is sketched out in Algorithm 8. The **shared** keyword is used to denote variables stored within the GPU shared memory.

Each task is assigned to a specific streaming multiprocessor (line 3), while each query is assigned to a specific warp (line 4). Once a warp is in charge of a query, it considers successive blocks of objects falling within c, each one having (maximum) size $warpSize$ (lines 10-13 and 28-30). Subsequently, we parallelize at thread level, where each thread checks whether the distance between the query and the considered object is below $dist_k$ or not, and updates a boolean field ($isResult$) accordingly (lines

Algorithm 8: $distCompPhase2(\overline{C}, Q, P, MAXDIST, NUMRES, k)$

Input : The set of active cells with at least one query, $\overline{C}$.
The reordered query set Q and object set P, along with the indexing information associated after the respective sorting phases.
The size of the queries neighbours lists, k.
The vector containing, for each query, the distance of the farthest nearest neighbour found so far, *MAXDIST*.
The size of a warp of threads, *sizeWarp*.

Output: The struct of vectors containing the queries result set, $(ID, DIST)$.

1. **begin**
2. **shared** buffID[$2 \cdot warpSize$], buffDist[$2 \cdot warpSize$]
3. **foreach** $c \in \overline{C}$ **parallel$_{\text{block}}$ do**
4. **foreach** $q \in c$ **parallel$_{\text{warp}}$ do**
5. **local** $resultsFound \leftarrow 0$
6. **local** $resultsWritten \leftarrow 0$
7. **local** $occupancyBuffer \leftarrow 0$
8. **local** $isResult$
9. **local** $dist_k \leftarrow MAXDIST_q$
10. $c_P = \{c \cap P\}$
11. $i = 1,\ j = min(|c_P|, warpSize)$
12. $s_P = < p_i, ..., p_j > \in c_P$
13. $c_P \leftarrow \{c_P - s_P\}$
14. **while** $(s_P \neq \emptyset)$ **do**
15. $isResult \leftarrow$ false
16. **foreach** $p \in s_P$ **parallel$_{\text{thread}}$ do**
17. **if** $((dist(p, q) < dist_k) \wedge (p_{id} \neq q_{id}))$ **then** $isResult \leftarrow$ true
18. **local** $bitMaskResults \leftarrow ballot(isResult)$
19. **local** $resultOffset \leftarrow computeResultOffset(bitMaskResults, occupancySharedBuffer)$
20. **local** $resultsFoundBlock \leftarrow popCount(bitMaskResults)$
21. $resultsFound \leftarrow resultsFound + resultsFoundBlock$
22. **if** $(isResult = true)$ **then**
23. $(buffID, buffDist, occupancyBuffer) \leftarrow updateBuffer(resultOffset, occupancyBuffer, buffID, buffDist)$
24. **if** $(resultsFound - resultsWritten \geq warpSize)$ **then**
25. $(buffID, buffDist) \leftarrow flush(buffID, buffDist, occupancyBuffer, resultsWritten)$
26. $resultsWritten \leftarrow resultsWritten + warpSize$
27. $occupancyBuffer \leftarrow occupancyBuffer - warpSize$
28. $i \leftarrow j + 1,\ j \leftarrow min(|c_P|, j + warpSize)$
29. $s_P \leftarrow < p_i, ..., p_j > \in c_P$
30. $c_P \leftarrow \{c_P - s_P\}$
31. **if** $resultsFoundneqresultsWritten$ **then**
32. $flush(buffID, buffDist, occupancyBuffer, resultsWritten)$

16-17).

Once this check is over, threads inside a warp have to find out which threads actually found out a nearest neighbour (if any) inside the currently considered objects block and, for what relates to such threads, find out the position in which they can store their result inside the temporary buffer (lines 18 - 21). First, we exploit the *ballot* function [2] (line 18), a native CUDA instruction which allows to test in parallel a given predicate within a warp. The output of such function is a *sizeWarp*-wide integer, whose bits represent the outcome of the test for each thread within the warp. We note that this instruction allows to avoid expensive synchronization mechanisms at warp level, since in the opposite case we would be forced to use some kind of atomic counters in order to find out the correct write positions.

In any case, after using the *ballot* function each thread in the warp proceeds by inspecting *bitMaskResults* in order to find out (i) the position in the temporary buffer where it can store the nearest neighbour it found (if this is the case) (*computeResultOffset* function) and (ii) determine the amount of nearest neighbours found inside the currently considered block (*resultsFoundBlock* field; we note that the latter operation equates to a bit counting operation, which is natively done on GPU through the *popCount* function). *resultsFoundBlock* is subsequently used to update *resultsFound*, the field containing the overall amount of nearest neighbours found so far by the warp (line 21).

Once all these information are determined, the warp proceeds by updating the temporary buffer in shared memory (line 23). We note that given the set of information available, the size of the buffer ($2 \cdot warpSize$) and the maximum amount of results a warp can store inside the buffer when scanning a block of objects ($warpSize$), bank conflicts among threads are impossible when writing in shared memory, thus assuring the best possible performances.

Warp threads obviously have to flush out periodically the temporary buffer content; this is required whenever (i) the amount of nearest neighbours stored inside exceeds half of its capacity (line 24) or (ii) the buffer content has to be flushed out before termination (line 31): this is achieved by means of a simple *flush* operation - called by all warp threads - which flushes out *warpSize* buffer elements at a time by means of a cooperative pattern, which in turn entails coalesced write accesses.

Query computation terminates whenever the warp has processed all the objects in c_P (i.e., $s_P = \emptyset$) and flushed out the remaining temporary buffer content (if any) in global memory.

3.2.2.1.2.1 Complexity. Aside from the strategies used to write out queries results, the *cached* and *coalesced* strategies are, on the whole, almost identical. For this reason, we take Algorithm 5 as a point of reference. If we consider the operations carried on within each task, the complexity of the distance computation phase is

[2] This instruction is natively supported by CUDA devices whose hardware revision is 2.0 or greater.

mainly dictated by the scans over the set of objects belonging to each cell $c \in \overline{C}$, as well as by the k-selection algorithm.

The overall number of scans inside Algorithm 5 are two, one at line 4 and the other one at lines 7-11. The related complexity is therefore equal to $O(2 \cdot |\{P \cap c\}|)$. For what is related to the k-selection algorithm (Algorithm 6), we have that the number of iterations required to find a suitable $dist_k$ strongly depends on local densities affecting the spatial distribution within a cell. If d represents the minimum distance between pairs of objects in c and *numBins* represents the amount of bins used, such figure is equal to [48]:

$$O\left(\left\lceil \left| log_{numBins}(\frac{dist_{max} - dist_{min}}{d}) \right| \right\rceil\right), \tag{3.1}$$

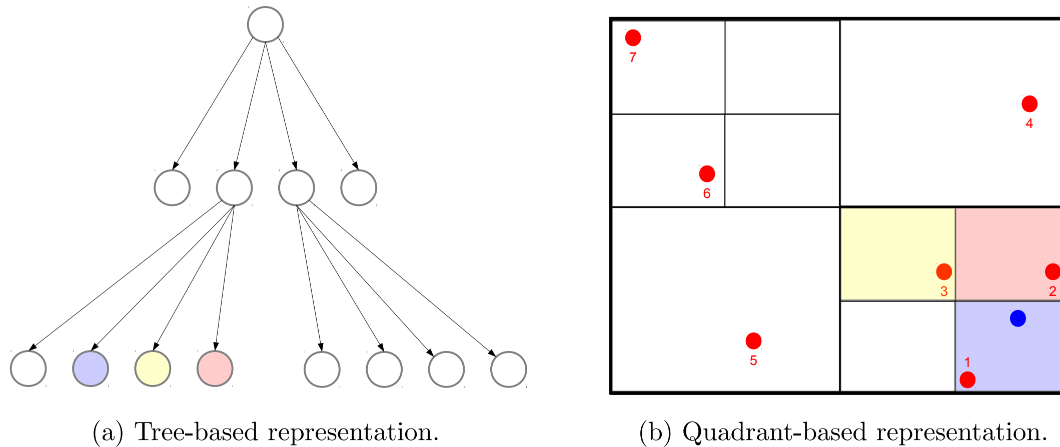
Summing up the considerations done above, if we denote the amount of iterations yielded by *findKDist* in the worst case as *maxIterations*, the overall complexity of the distance computation phase becomes:

$$O\bigg((2 + maxIterations) \cdot |\{P \cap c\}|\bigg). \tag{3.2}$$

3.2.2.2 Subsequent iterations

After the first iteration, each query has associated a list of nearest neighbours, each one containing up to k elements, contained within the cell in which its center falls. Depending on the spatial distribution and on the materialized index, however, it is very probable that a substantial fraction of queries have an incorrect or incomplete list. The former case happens when there is at least one object falling outside the cell in which the query center falls, and such objects are nearer than the farthest object in the list computed during the first iteration. The latter case happens when a query falls inside a cell containing less than k objects (and $k < |P|$), and therefore the resulting list is - at least - partial: clearly, we need to consider other neighbouring, non-devoid cells in order to complete the list. Obviously, it is possible to have a combination of these two cases as well. Hereinafter we denote such queries by *active queries*.

Since the latter case is obvious we focus on the former case, presenting a toy example in Figure 3.2. In this example we have a 1-NN query, represented by the blue dot, and seven moving objects, represented as red dots. By design, during the first iteration only objects falling in the cell where the query center is located (the quadrant highlighted in *blue*) are considered. Consequently, after the first iteration we have that the list associated with the query will be $\{o_1\}$. However, it is evident that objects o_2 and o_3 are closer than o_1, therefore we have to consider the cells (highlighted in red and yellow, respectively) in which such objects fall, in order to compute the final, correct query list.



(a) Tree-based representation. (b) Quadrant-based representation.

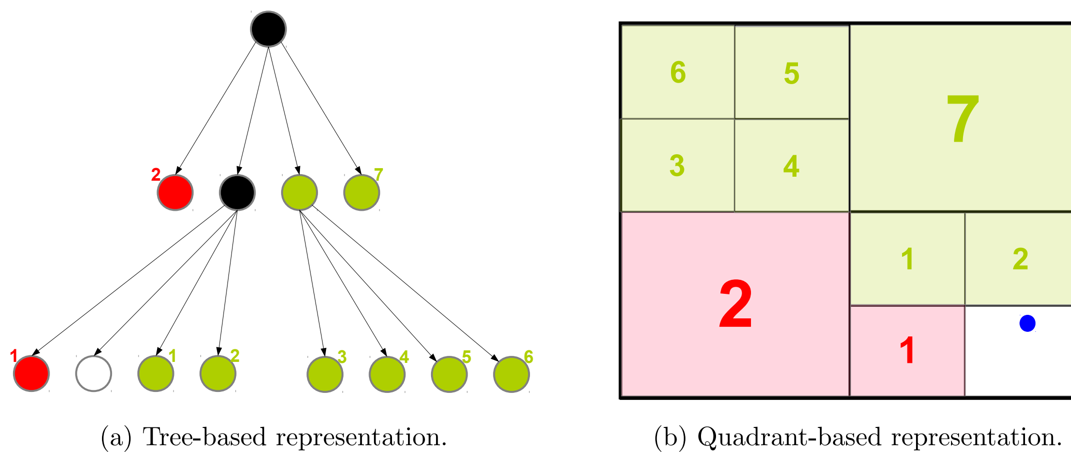
Figure 3.2: Toy example of a 1-NN query for which we have to analyze the content of neighbouring quadrants in order to compute the final, correct result set.

To this end we need to perform a *recursive tree visit*, starting from the quadtree leaf in which the query center falls, considering only those leaves whose (i) spatial extent may contain potential nearest neighbours and (ii) contain at least one object. As the visit progresses, the query list (and the related auxiliary information) must be updated accordingly.

Going back to the example, we have that the visit must consider the siblings of the quadtree leaf in which the query center falls. Accordingly, the visit starts from the *white* quadrant, which can be immediately discarded since it is devoid of objects. The visit proceeds by considering the *yellow* quadrant: here we have that o_3 is nearer than o_1, therefore the query list is updated to $\{o_3\}$. Finally, the visit reaches the *red* quadrant, where o_2 is located: since o_2 is nearer than o_3, the query list is updated to $\{o_2\}$. At this point the visit goes up all the way to the root, since the remaining quadtree nodes do not contain objects which are possibly nearer than o_2.

Considering the potential amount of active queries to be processed after the first iteration, the main challenge becomes to devise a strategy able to batch the work resulting from such tree visits in order to generate workloads suitable for GPU processing. Our proposal consists in an iterative approach, where tree visits and distance computations related to spatially nearby active queries are packed together, iteration after iteration, until no query remains active, i.e., every query has associated the final, correct list of nearest neighbours. In order to achieve this goal, our proposal has to find an effective way to combine GPUs caching capabilities with Morton codes spatially preserving properties. Algorithm 9 sketches out our proposal, which is discussed in the following.

Strategy overview. The basic idea is to perform the aforementioned recursive tree visit by splitting it into two sub-visits, one that proceeds towards left while the other one proceeding towards right, until potential nearest neighbours can be found. Both sub-visits start from the leaf in which the query center falls and shall be ideally conducted in an *alternate* fashion, so to minimize the amount of nodes to visit before reaching the final, correct query list. In this sense, an active query may be indeed active in one direction but not in the other, depending on the list of nearest neighbours found so far and on the unvisited, non-devoid quadrants spatial extent. Figure 3.3 gives an example about how these sub-visits may span the set of quadtree nodes with respect to the query given in Figure 3.2, along with the order in which they would consider quadtree leaves.



(a) Tree-based representation. (b) Quadrant-based representation.

Figure 3.3: Left and right sub-visits, quadtree nodes coverage example.

Starting from the leaf in which the query center falls (*white* leaf/quadrant in the Figures; the query is represented as a blue dot in Figure 3.3b), the sub-visit proceeding towards left has to cover the nodes highlighted in *red*, while the other one has to cover the nodes highlighted in *green*. Black nodes represent nodes which are considered by both sub-visits. We note how this visiting schema partitions the set of leaves into two disjoint sets - left apart the one where the query center falls. This is even more evident if we take into consideration the quadrant-based representation (Figure 3.3b).

Both Figures highlight the order in which the sub-visits consider the leaves: we note that this order *follows* the order imposed by Morton codes over the quadtree structure. One could reasonably observe that following such order represents a sub-optimal strategy, since it does not ensure to visit the minimum possible amount of leaves; indeed, selecting neighbouring cells according to their spatial extent would, on the contrary, achieve this goal. However, in favour of the former strategy we observe that following the order imposed by Morton codes approximates quite well spatial proximity, usually yielding an amount of visited leaves equivalent or near to

the minimum one. Additionally, visiting the tree according to the latter strategy would require to implement a GPU-based, massive tree visiting algorithm having an higher computational overhead. Putting together the above considerations, we deem that possible benefits deriving from the latter strategy may be not worth the effort.

Algorithm 9: K-NN$_{GPU}$ – Schema used for subsequent iterations

Input : The structure of streams containing all the information about the set of queries Q after the first iteration.
The quadtree-based index, $\mathcal{C}$.
The size of the queries neighbours lists, k.
The result set $(ID_k, DIST_k)$.
The temporary result set $(ID_k^{temp}, DIST_k^{temp})$.
The vector containing, for each query, the distance of the farthest objects in each list, *MAXDIST*.
The vector containing the amount of neighbours found per each query, *NUMRES*.
Output: The final result set, $(ID_k, DIST_k)$.

```
1  begin
2      global Q̄_left ← initNavigation(Q, left)
3      global Q̄_right ← initNavigation(Q, right)
4      local direction ← left
5      while (Q̄_left ≠ ∅ ∨ Q̄_right ≠ ∅) do
6          if (direction = left) then Q̄_processed = Q̄_left
7          else Q̄_processed ← Q̄_right

8          Q̄_processed ← navigateTree(Q̄_processed, C, z_map, k, MAXDIST, NUMRES)
9          Q̄_processed ← sortActiveQueries(Q̄_processed)
10         (C̄, Q̄_processed) ← indexBlocksActiveQueries(Q̄_processed)

11         ((ID_k^temp, DIST_k^temp), MAXDIST) ←
           distComp(C̄, Q̄_processed, MAXDIST, NUMRES, k)

12         (ID_k, DIST_k, MAXDIST, NUMRES) ←
           updateNNLists(C̄, Q̄_processed, (ID_k^temp, DIST_k^temp), (ID_k, DIST_k), MAXDIST, NUMRES, k)

13         if (direction = left ∨ Q̄_left = ∅) then
14             Q̄_left ← Q̄_processed
15             direction ← right

16         else
17             Q̄_right ← Q̄_processed
18             direction ← left

19     return (ID_k, DIST_k)
```

In line with this idea, we consider the strategy sketched in Algorithm 9. First, we initialize the structures of vectors containing the tree navigation status of the queries. We keep track of the information related to the sub-visit towards left in $\overline{Q}_{left}$, while

the other one in $\overline{Q}_{right}$ (lines 2-4); both start from the quadtree leaf containing the query center. We note that queries in Q come sorted according to the identifiers (augmented Morton codes) of the cells with which they were associated during the first iteration: this is the *key property* exploited, iteration after iteration, in order to pack together the workload related to spatially nearby queries. We also mention that in $\overline{Q}_{left}$ and $\overline{Q}_{right}$ we just keep query references without copying the actual query data, so to avoid moving too much data when subsequently sorting active queries. As a consequence, during the subsequent iterations we rely on caching also when recovering query data.

Then, the iterative process begins (line 5): at the beginning of each iteration, the approach alternatively considers one sub-visit, as long as both sub-visits are active (lines 6-7), and updates the navigation status of each active query in the direction considered (*navigateTree*, line 8). Since we are orchestrating the iterations by alternating the sub-visits, hereinafter we conveniently say that a query is active, in the direction considered, whenever the related sub-visit has not terminated. From this, it follows that a query is, in general, active whenever it still has a sub-visit going on. At the end of this operation we expect that each active query is assigned to the first unvisited, non-devoid leaf whose spatial extent may contain possible nearest neighbours (in practical terms, each active query is associated with the identifier of such leaf), if any. If no useful leaf is found, the query is flagged as *inactive*: this case corresponds to the event in which the visit reaches the root and all its childs - in the direction considered - were already visited. We embed the inactive flag in the most significant bit of the integer used to store the cell identifier in order to discern between active and inactive queries - by means of sorting - later on.

After the tree navigation update phase, queries are *sorted* according to the newly assigned cell identifiers (*sortActiveQueries*, line 9). We exploit the sorting operation in order to partition between active and inactive queries as well: once active queries get sorted, it suffices to find the first query having the inactive flag set in order to determine the extents of the active and inactive sets. Such simple, yet massively parallel, operation is conveniently performed on GPU as well.

After the sorting operation we focus just on those queries still considered active, while discarding the inactive ones. By virtue of sorting, active queries are arranged such that those assigned to the same cell are displaced in contiguous memory locations. We exploit again this property, as done already in Section **??**, in order to determine the first and last query for each cell having at least one query assigned (line 10). This yields a set of tasks, one per active cell associated with at least one active query. We denote again such set by $\overline{\mathcal{C}}$.

Subsequently, the approach has to update the list of nearest neighbours for each active query with respect to the newly assigned cell (lines 11-12). This step is almost equivalent to the one described in Algorithm 5: first (function *distComp*, line 11), we find a suitable distance below which we have (up to k) potential nearest neighbours in the newly assigned cell (this is done by means of the k-selection algorithm) and store such objects in a temporary result list, $(ID_k^{temp}, DIST_k^{temp})$. Finally (function

updateNNLists, line 12), we update the query list, $(ID_k, DIST_k)$, according to its current content and the content of $(ID_k^{temp}, DIST_k^{temp})$.

The iteration concludes by updating $\mathcal{Q}_{left}$ ($\mathcal{Q}_{right}$) and by selecting the sub-visit of the next iteration according to $\mathcal{Q}_{left}$ and $\mathcal{Q}_{right}$ content (lines 13-18).

Why we use this two-pass strategy based on temporary lists? The main reasons are two, both strictly correlated: first, for each query we can ignore consistent amounts of objects in the newly assigned cell simply by looking at the distance of the farthest object in the query list found so far (information which can be retrieved from $MAXDIST$); we note that such optimization cannot be exploited in case a query has less than k results in the list computed so far. Second, we resort to a temporary result buffer since executing the k-selection algorithm in one single shot, i.e., by considering the objects in the query list and the objects within the newly assigned cell as well, usually yields an increased amount of iterations due to the lack of the aforementioned pruning.

In the following paragraphs we describe in detail the main operations carried on during any subsequent iteration.

3.2.2.2.1 Subphase 1 – Massive tree navigation. As sketched out in Algorithm 9, we conveniently keep two distinct navigation tree statuses, one keeping track of the visits towards left (the related struct of vectors containing such information is $\overline{Q}_{left}$) and the other one towards right ($\overline{Q}_{right}$). We also saw how during any subsequent iteration we consider *one* direction at a time - as long as both sub-visits are going on, so to minimize the amount of overall iterations (quadtree leaves) per query needed to converge to the final lists.

During this phase (Algorithm 9, line 8) we update the navigation status of each active query with respect to the sub-visit *currently considered* (where $\overline{Q}_{processed}$ represents just a reference to the struct of vectors considered). In the end, we expect that each active query is assigned to an unvisited leaf containing potential nearest neighbours, if any (in this case the query will be still considered *active*), or flagged as *inactive* in no useful leaves are found.

The goal is to conveniently perform such operation on GPU by assigning one thread per active query. However, considering the hardware platform we want to exploit the main challenge is to devise a strategy which is able to minimize issues arising from the observation that, the farther the queries are, the more they may exhibit different tree traversals. Indeed, if we happen to pack in the same thread block queries exhibiting different traversals, we may encounter two serious performance issues: the first one is due to *execution branching* inside warps (different paths may require the execution of different operations); the second one originates from the observation that we may achieve far from optimal memory accesses due to different traversals accessing quadtree information stored in far memory locations, thus denying any possible caching benefit.

However, we can minimize these issues since active queries come already sorted

from previous iterations. More precisely, given that spatially nearby active queries are arranged in nearby memory locations, if we statically partition the queries by retaining such order we expect that queries processed inside the same thread block will exhibit *equivalent* or *similar* tree traversal. In Algorithm 10 we show the algorithm used to perform massive tree navigation on GPU, thus allowing to better motivate our strategy.

Algorithm 10: $navigateTree(Q_{process}, \mathcal{C}, z_{map}, k, MAXDIST, NUMRES)$

Input : The set of active queries, $Q_{process}$, coming sorted from the previous iteration.
The spatial index, $\mathcal{C}$ (along with auxiliary information).
The vector containing, for each query, the distance of the farthest nearest neighbour found so far, *MAXDIST*.
The vector containing, for each query, the amount of nearest neighbours found so far, *NUMRES*.

Output: The set of active queries, $Q_{process}$, updated.

```
1  begin
2    local l_deep ← getQuadtreeDeepestLevel(C)
3    foreach q ∈ Q_process parallel_thread do
4      local maxdist ← MAXDIST_q
5      local numres ← NUMRES_q
6      local (l, z) ← retrieveNavigationStatus(q)
7      if (hasSibling(l, z, direction) = true) then (l, z) ← getSibling((l, z), direction)
8      else (l, z) ← getParent(l, z)
9      while (l ≠ 0) do
10       if (containsPotentialNeighbours(q, (l, z), maxdist, numres)) then
11         if (l = l_deep) then
12           (l, z) ← getQuadtreeLeaf(z_map, (l, z))
13           if notEmpty((l, z), C, P)) then
14             Q_process ← associateLeaf(Q_process, q, (l, z))
15             return
16           if hasSibling(l, z, direction) then (l, z) ← getSibling((l, z), direction)
17           else (l, z) ← retrieveParent(l, z)
18         else (l, z) ← getFirstChild((l, z), direction)
19       else
20         if hasSibling(l, z, direction) then (l, z) ← getSibling((l, z), direction)
21         else (l, z) ← retrieveParent(l, z)
22     Q_process ← setInactiveFlag(Q_process, q)
```

As mentioned before, we assign each active query to a specific thread (line 3). Subsequently, the thread in charge retrieves the main information about the query navigation status, such as the quadtree leaf (l, z) assigned during the last iteration considering the *same* sub-visit, the distance from the farthest object in the list computed so far, *maxdist*, and the amount of results in the list, *numres* (lines 4-6).

Then, the algorithm checks whether (l, z) has a sibling in the direction considered, or the visit must restart from the leaf's parent (lines 7-8).

Subsequently, the tree visit cycle kicks in (line 9). Here, the algorithm first determines if the quadrant currently considered may contain potential nearest neighbours, i.e., its borders are nearer than the query farthest neighbour found so far (line 10). If this condition holds, the algorithm checks whether the quadrant level is equal to the deepest quadtree level (line 11); if this is true, the algorithm goes on by retrieving the quadtree leaf covering such quadrant and checks whether the leaf contains at least one object (lines 12-13): if this holds, the query will be still considered *active* and assigned to the leaf for further processing (line 14) (in this case the thread terminates as well), otherwise the visit continues (lines 16-18).

In case a quadtree node/leaf does not contain potential nearest neighbours, the algorithm goes on with the visit, checking whether there are other useful internal nodes/leaves in the direction considered (lines 20-21).

The visit terminates whenever the visit reaches the quadtree's root (level 0): in such case the query is flagged as inactive (line 22).

Some side remarks must be done about important design choices done inside Algorithm 10: first, we take into consideration the spatial extent of quadtree quadrants instead of the MBRs enclosing the objects inside: even if this choice may slightly increase the amount of leaves considered per query, it also avoids a relevant amount of lookups in memory, since (i) the spatial extent of any quadtree quadrant can be determined on the fly and (ii) a lookup is required only when we have to understand which quadtree leaf covers a leaf reached in l_{deep} (by means of z_{map}). Second, and strictly bounded to the above design choice: we always reach l_{deep} before assigning a query to a new quadtree leaf through the inverted index z_{map}. On one hand this may slightly elongate the traversals, thus increasing the amount of operations; on the other hand, it avoids to check each time (by means of a lookup in an appropriate data structure) whether any quadrant at any level is actually a leaf or not.

Complexity. Query-wise, the complexity of this subphase is dictated by the set of active queries in the direction considered, i.e., $O(|Q_{processed}|)$; if we focus on a single query, the worst case scenario corresponds to visiting all quadtree nodes when such quadtree corresponds to a uniform grid, given that we use z_{map} to recover leaves identifiers (line 12). If l_{max} is the deepest quadtree level, this corresponds to a complexity equal to $O(\sum_{l=0}^{l_{max}} 4^l) = O(\frac{1-4^{l_{max}-1}}{1-4})$.

However, these complexities tell us very little on the overall amount of iterations we have to expect from our approach, since this depends on portions of the tree involved by queries, which in turn depend on a complex mix of factors such as the objects spatial distribution, how such distribution may vary over the time, the quadtree height, how objects spread across the quadtree leaves and the amount of nearest neighbours per query to compute (k).

It would be interesting to perform an analysis in order to provide an estimate

for such figure; however, some of the factors mentioned are not known beforehand and they may vary over the time, therefore making the conduction of such analysis almost impossible. In general, whenever we fix a tree height we expect that the amount of iterations needed to converge to the final, correct query result set is slightly higher whenever distributions get skewed.

3.2.2.2.2 Subphase 2 – Active queries sorting and task formation. After subphase 1, each query considered as *active* during the previous iteration, in the direction considered, is associated with a cell identifier representing a quadtree leaf containing possible nearest neighbours, or flagged as *inactive* otherwise.

Afterwards, queries are sorted according to the associated cell identifier. Since inactive queries get flagged during subphase 1, by means of sorting we can conveniently displace at the beginning of the associated struct of vectors those queries still considered active. Then, in order to determine the extent of the active queries set we use a simple GPU kernel where we assign each query to a single thread: the thread detecting the last active query will return its position in memory, while other threads won't do anything. Once we have such information we can discard the whole set of inactive queries, since these can be ignored during subsequent iterations. All the aforementioned operations are performed in Algorithm 9, line 9 (*sortActiveQueries* function).

Finally, our approach determines the first and last active query for each cell having at least one query assigned and creates an ad-hoc index (Algorithm 9, line 10, *indexBlockActiveQueries* function).

The end product of such chain of operations is, again, represented by a set of tasks, one per active cell with at least one active query assigned (we denote such set by $\overline{\mathcal{C}}$, again), representing the GPU workload.

Complexity. The complexity of this subphase is dictated by the set of active queries in the direction considered. Let's denote this set by $Q_{processed}$. First, queries are sorted according to the identifier of the newly assigned cell: this has complexity $O(b \cdot |Q_{processed}|) \approx O(|Q_{processed}|)$, since the sorting algorithm is Radix Sort (and b the base used). Then, we determine the extent of the active query set: this has complexity equal to $O(|Q_{processed}|)$, since we assign each thread to a query in Q. Finally, we determine the set of cells, which has complexity equal to $O(|Q_{processed}| + |\overline{\mathcal{C}}|)$. In light of the above considerations, the overall complexity is therefore:

$$O\left(3 \cdot |Q_{processed}| + |\overline{\mathcal{C}}|\right). \tag{3.3}$$

3.2.2.2.3 Subphase 3 – Nearest neighbours lists update. Once the set of tasks is materialized, we can proceed by updating the nearest neighbours lists of the active queries (Algorithm 9, lines 11-12).

First, for each active query q our approach computes the list of the (up to) k

nearest neighbours inside the newly assigned cell (let us denote it by *c)* and store it in a temporary result buffer represented by the couple $(ID_k^{temp}, DIST_k^{temp})$ (line 11). This operation is essentially equivalent to the one described in Section 3.2.2.1.2 (putting aside the specific writing strategy used), yet with an important distinction to be made: if the amount of c's objects having a distance below $MAXDIST_q$ is greater than k, then we must use the k-selection algorithm in order to select the k nearest objects inside c; otherwise, we can simply copy in $(ID_k^{temp}, DIST_k^{temp})$ the objects falling below such threshold.

Then, the algorithm proceeds by fusing the main and temporary lists. This operation is, again, almost equivalent to the one described in Section 3.2.2.1.2, however by noting that if $|(ID_k, DIST_k) \cup (ID_k^{temp}, DIST_k^{temp})| > k$ the algorithm must apply the k-selection algorithm in order to correctly update $(ID_k, DIST_k)$, otherwise it just suffices to perform a simple union set $(ID_k, DIST_k) = (ID_k, DIST_k) \cup (ID_k^{temp}, DIST_k^{temp})$.

Complexity. The overall complexity of this subphase can be determined by means of Equation 3.1, since the characterizing subcomplexities are equivalent to the ones of the *distance computation* phase (Section 3.2.2.1.2).

More precisely, if c denotes the cell considered for a given query q, we have that the complexity related to line 11 is equal, in the worst case, to:

$$O\Big((2 + maxIterations) \cdot \Big|\{p|p \in c \wedge d(p,q) < MAXDIST_q\}\Big|\Big), \tag{3.4}$$

while the operations performed at line 12 induce a complexity which is equal, in the worst case, to:

$$O\Big((2 + maxIterations) \cdot 2k\Big). \tag{3.5}$$

3.3 Experimental Setup

All the experiments are conducted on a PC equipped with an Intel Core i7 2600 CPU, running at 3,4 GHz, with 16 GB RAM and an Nvidia GTX 580 GPU with 3 GB of RAM coupled with CUDA 5.5. The OS is Ubuntu 12.04. For what concerns workload generation and testing we reused the framework already employed in Section 2.3.

As done in the experimental section of Chapter 2 (see Section 2.3), we exploit three types of synthetic datasets, i.e., *uniform datasets*, *gaussian datasets*, and *network datasets*. In all tests we compute repeated k-NN queries over 30 ticks. To model object movements the framework generates 30 instances of each dataset, one for each tick.

We use K-NN_{GPU} to generally denote the GPU-based solution we propose. Depending on the specific approach used to update the queries nearest neighbours

list, we denote by $\text{K-NN}_{\text{GPU}}^{CACHE}$ the flavour which uses only the GPU caching capabilities when updating the queries lists, while we denote by $\text{K-NN}_{\text{GPU}}^{COALESCE}$ the approach using coalescing capabilities as well. For both flavours the reader must refer to Section 3.2.2.1.2. Wherever not specified, we suppose that K-NN_{GPU} refers to $\text{K-NN}_{\text{GPU}}^{CACHE}$.

In order to prove the performance benefits given by K-NN_{GPU}, we pick up as a direct competitor the exact k-NN search sequential algorithm offered by the FLANN library [3] , one of the most popular libraries designed for nearest neighbour matching. We denote such competitor by K-NN_{CPU}. In order to compute a set of k-NN queries K-NN_{CPU} relies on the construction of a kd-tree over the set of points to be searched, with a subsequent series of k-NN searches performed over the kd-tree. As a consequence, K-NN_{CPU} applies such *build and search* schema at every tick. K-NN_{CPU} is used inside our tests such that it uses just 1 CPU core and uses an optimized L2 distance functor when computing distances in $\mathbb{R}^2$. We also set to 32 the maximum amount of objects per kd-tree leaf, since this value proves to be the best choice in almost all cases inside our experimental setting.

Finally, we consider a well-established GPU-based baseline approach, i.e., the brute-force algorithm presented by Garcia et al. in [35] [4] . We denote such approach by $\text{K-NN}_{\text{BASELINE}}$ and we use it in order to prove the merit of our proposal with respect to GPU naive solutions.

Table 3.1 summarizes the main parameters used to generate the datasets. The listed parameters apply to all the datasets, except for the amount of hotspots which is relevant for gaussian datasets only. The framework uses a generic spatial distance unit u (e.g., meters).

K-NN_{CPU} reports the result set of each k-NN query in form of a list of k object identifiers, thus returning a set of lists at the end of each tick. To avoid bias when making performance comparisons, we force K-NN_{GPU} to transfer the GPU-generated results to the host memory in form of a set of k-NN lists, one per query, having an identical layout with respect to those returned by K-NN_{CPU}. The same applies to $\text{K-NN}_{\text{BASELINE}}$.

3.4 Experimental Evaluation

The experimental studies conducted for this work are introduced below, and are denoted by $S1, \ldots, S4$:

S1 We study how K-NN_{GPU}'s performance is affected when varying the maximum amount of objects per quadtree leaf (thus influencing the tree height), the amount of results per query (k), the dataset skewness and the fraction of objects issuing a query (query rate).

[3] http://www.cs.ubc.ca/research/flann/, version 1.8.4.
[4] http://vincentfpgarcia.github.io/kNN-CUDA/

Spatial region	All tests occur in a squared spatial region with side length of 22500 u.
Amount of objects	We vary the number of moving objects from $100K$ to $1500K$. In some tests the number of moving objects is fixed and the exact amount is explicitly stated in their descriptions.
Objects maximum speed	In all tests the maximum speed of each object is fixed to 200 u per tick (Δt), where the objects are allowed to change their speed as described in [16]. In general, changes in speed may slightly alter the objects distribution but do not change the distribution general properties.
Query rate	The percentage of objects that issue a range query during every tick is always set to 100%.
Amount of ticks	Whenever not specified the default amount of ticks, corresponding to different snapshots of a dataset, is 30. Consecutive snapshots are expected to exhibit slight changes, according to the properties of the dataset spatial distribution.
Query location	All the queries are centered around the objects issuing them.
Amount of hotspots	Depending on the experiments goals and specificities, the amount of hotspots is varied in the $[10, 150]$ range. Whenever not specified the default value used is 25.
Neighbours list size k	Depending on the experiments goals and specificities, the neighbours list size is varied in the $[1, 512]$ range. Whenever not specified the default value used is 32.

Table 3.1: Data and workload generation parameters.

S2 We show how K-NN$_{\text{GPU}}$ outperforms K-NN$_{\text{BASELINE}}$.

S3 We compare K-NN$_{\text{GPU}}$ against K-NN$_{\text{CPU}}$. To this end we vary again main dataset and run-time parameters, such as the amount of objects and the neighbours list size k, as well as consider three different spatial distributions (uniform, gaussian and network) characterized by different degrees of skewness. We also make a comparison between two different flavours of K-NN$_{\text{GPU}}$, i.e., K-NN$_{\text{GPU}}^{CACHE}$ and K-NN$_{\text{GPU}}^{COALESCE}$, when varying k, in order to assess the benefits deriving from the usage of different writing strategies with different k values.

S4 We analyze how some of the aforementioned parameters affect the system bandwidth β (according to the definition provided in Section 1.1.4).

3.4.1 (S1) Tree height, neighbours list size, query rate and spatial skewness impacts on K-NN$_{\text{GPU}}$'s performance

In the following we analyze the impacts of few crucial factors influencing K-NN$_{\text{GPU}}$'s performance, such as the tree height, the neighbours list size (k), the query rate and the skewness characterizing a spatial distribution.

Tree height and neighbours list size. When processing any dataset, two crucial parameters are represented by the tree height, which is controlled indirectly by altering the maximum amount of objects admitted in a single quadtree leaf (th_{quad}), and the neighbours list size k associated with the each query. Choosing an optimal th_{quad} is strictly connected to k: if the tree height is too high with respect to k, then many leaves with few objects shall be visited, thus increasing the amount of operations needed to perform such operations. On the other hand, if the tree height is too low we end up visiting few leaves with many objects, thus possibly performing a relevant amount of useless computations due to the reduced pruning power of the index. As a consequence, it is necessary to find an appropriate th_{quad} for a given k.

In the batch of experiments that follows, we use a uniform dataset having a fixed amount of moving objects equal to 500K, where we test different combinations of th_{quad} and k values; other dataset characteristics are set to their defaults, as specified in Table 3.1.

Figure 3.4 shows how each k is associated with a range of optimal th_{quad} values for which the algorithm's execution time is minimized. We observe how such ranges are relatively wide as well, a property which is desirable since this minimizes performance fluctuations even when not using an optimal value. We also observe how the higher k is, the more the relative optimal th_{quad} range shifts towards higher values, as expected from the considerations done above. Finally, even if it is not evident for every curve the execution time starts to increase again whenever th_{quad} gets too high with respect to k, since the pruning power of the index gets reduced. Incidentally,

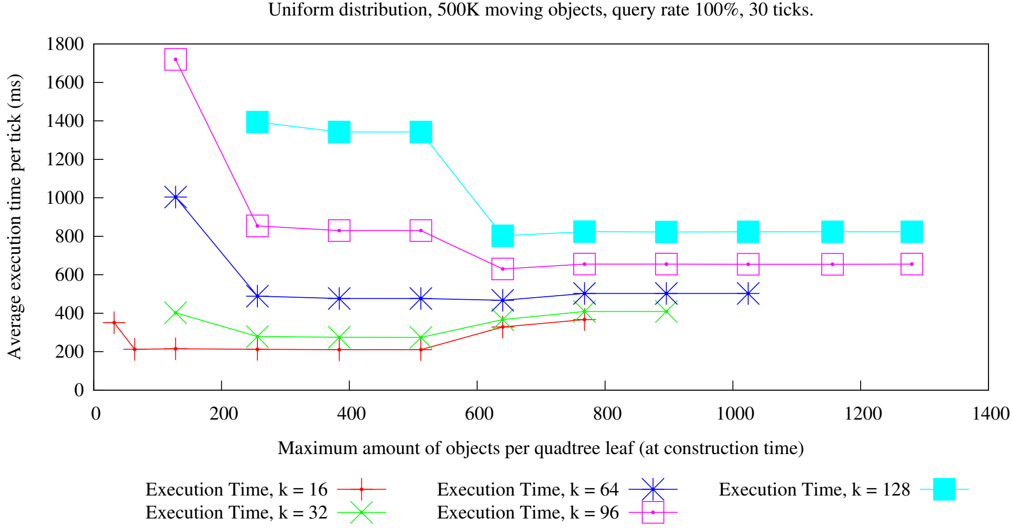


Figure 3.4: Relationship between the tree height (indirectly controlled through th_{quad}) and k, and its repercussions on K-NN$_{\text{GPU}}$'s performance.

we observe how the execution time increases whenever k increase, which is expected due to the higher amount of computations needed to compute the queries nearest neighbours. However, we will study extensively K-NN$_{\text{GPU}}$'s performance with respect to k in study S3, where we take into consideration both K-NN$_{\text{GPU}}$ flavours, i.e., K-NN$_{\text{GPU}}^{CACHE}$ and K-NN$_{\text{GPU}}^{COALESCE}$.

Spatial skewness impact on finding an optimal th_{quad}. In this batch of experiments we want to check whether the skewness has relevant impacts on finding an optimal th_{quad} range. Datasets are distributed according to a gaussian distribution, each characterized by a different amount of hotspots in order to yield differently skewed distributions. All datasets have a fixed amount of objects equal to 500K; other dataset characteristics are set to their defaults, according to Table 3.1. The size of nearest neighbours lists is set to $k = 32$.

From Figure 3.5 we see how the skewness does not influence, if not marginally, the optimal th_{quad} range for a given k. Even if in this series of experiments we use a fixed k, it is possible to show that the performance trend associated with different k values is approximately the same.

Query rate. In this batch of experiments we want to determine the impact of the query rate on K-NN$_{\text{GPU}}$'s performance. In the following we use a fixed uniform dataset having 500K objects; other dataset characteristics are set according to defaults (Table 3.1). We set the amount of nearest neighbours per query to $k = 32$; accordingly, we set $th_{quad} = 384$ in order to guarantee that K-NN$_{\text{GPU}}$ exhibits the best possible performances.

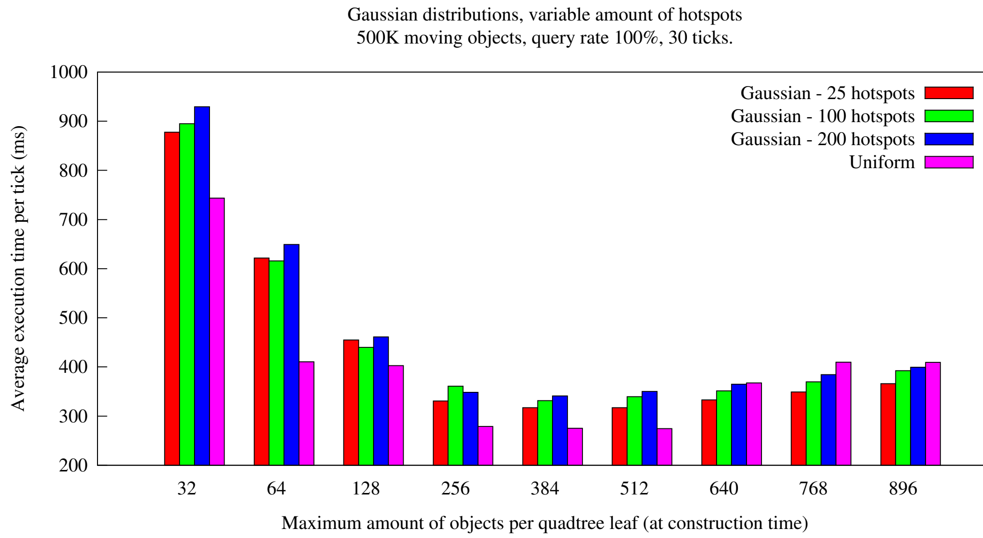


Figure 3.5: Skewness repercussions on th_{quad}'s optimality.

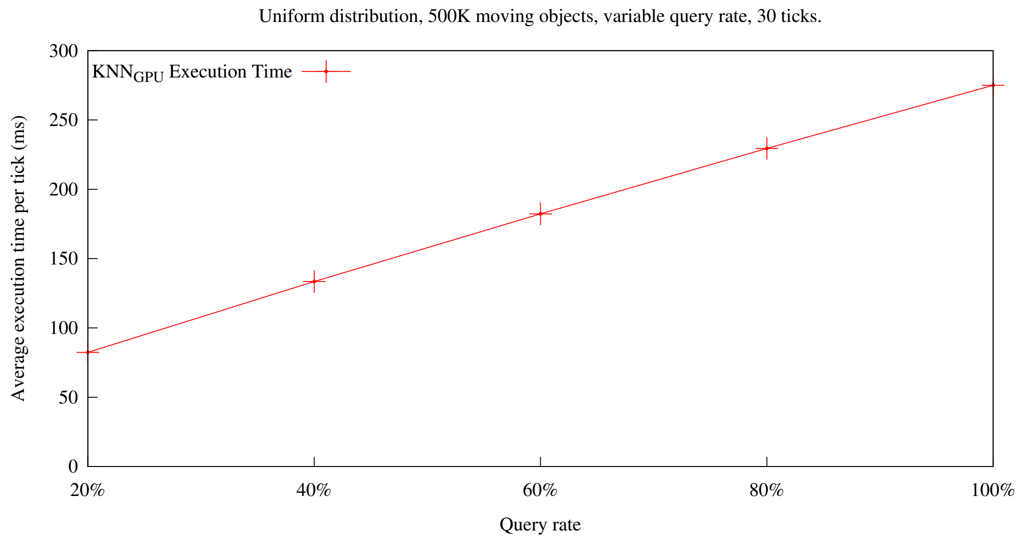


Figure 3.6: Query rate influence on K-NN$_{\text{GPU}}$'s performance. The amount of objects (500K), the neighbours list size ($k = 32$) and the maximum amount of objects per quadtree leaf ($th_{quad} = 384$) are all fixed across the experiments.

From Figure 3.6 we see how the query rate influences linearly K-NN$_{\text{GPU}}$'s execution time, as expected from the complexities described in Section 3.2.

3.4.2 (S2) K-NN$_{GPU}$ vs K-NN$_{BASELINE}$

In this study we take into consideration the GPU-based brute-force algorithm proposed in [35] by Garcia et al.. From now on we denote this algorithm as K-NN$_{BASELINE}$. We quickly remember that K-NN$_{BASELINE}$ computes for each query the distances with respect to all the objects in the dataset, subsequently sorting such distances in order to find the first k ones. In this study we aim to show how K-NN$_{GPU}$ outperforms K-NN$_{BASELINE}$. We take into consideration the two main parameters mainly affecting the performance, that is, the *amount of objects* and the amount of nearest neighbours per query list, k. In light of the results shown in study S2, for what regards K-NN$_{GPU}$ we set $th_{quad} = 12k$ when $32 \leq k \leq 256$ since this assures, on average, the best performance. For the remaining cases, we use $th_{quad} = 192, k < 32$ and $th_{quad} = 2048, k > 128$.

In the first batch of experiments we consider uniform datasets, where we vary the amount of moving objects while keeping fixed other dataset characteristics according to the defaults specified in Table 3.1.

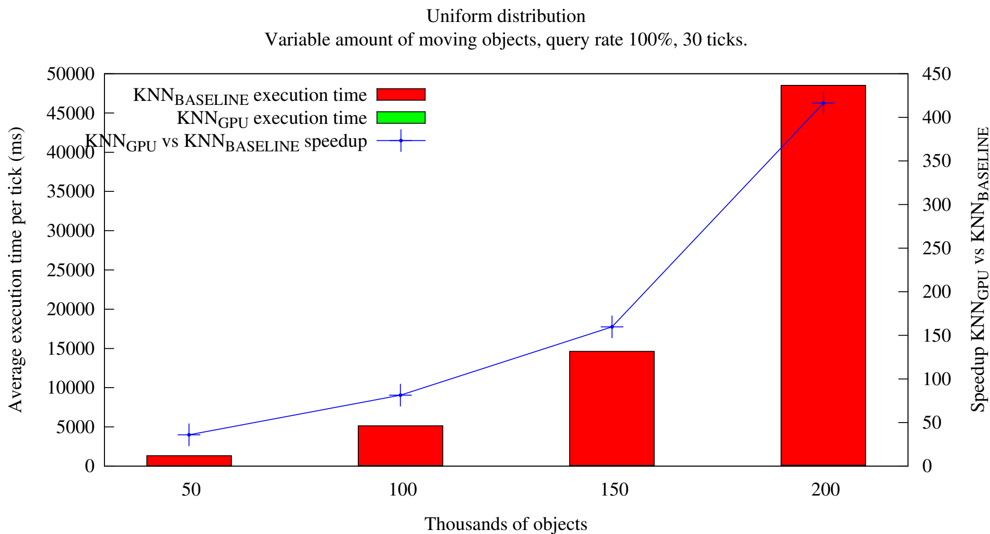


Figure 3.7: K-NN$_{GPU}$ vs K-NN$_{BASELINE}$, variable amount of moving objects, $k = 32$.

From Figure 3.7 we see how K-NN$_{GPU}$ heavily outperforms K-NN$_{BASELINE}$ as soon as the amount of objects gets relevant, since the usage of K-NN$_{BASELINE}$ becomes impractical with huge amounts of objects.

In the second batch of experiments we want to observe what happens whenever we vary the amount of nearest neighbours per query (k). Here, we consider a single uniform dataset having an amount of moving objects equal to 100K, while other characteristics are set to their defaults (according to Table 3.1).

From Figure 3.8 we see how K-NN$_{BASELINE}$'s execution time is fixed, since it depends exclusively on the amount of distances to compute and sort. On the other

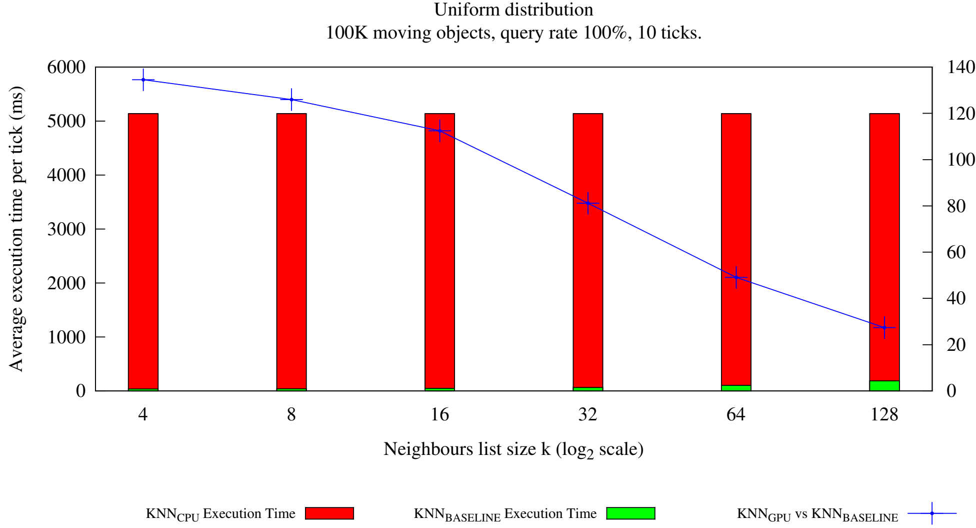


Figure 3.8: K-NN_{GPU} vs $\text{K-NN}_{\text{BASELINE}}$, variable nearest neighbours list size k.

hand, K-NN_{GPU}'s execution time increases when k increases, as expected, thus reducing the advantage gap with respect to $\text{K-NN}_{\text{BASELINE}}$. Putting together the above findings, we conclude that K-NN_{GPU} heavily outperforms $\text{K-NN}_{\text{BASELINE}}$, at least until k does not get very high.

In this study we didn't take into consideration the skewness, since this factor was outside the scope of the experiments. However, we expect that skewed distributions yield more irregular workload distributions, with slightly negative effects on K-NN_{GPU}'s performance. We study extensively the effects of this factor in study S3.

3.4.3 (S3) K-NN_{GPU} vs K-NN_{CPU}

In this section we compare K-NN_{GPU} against K-NN_{CPU}; in this analysis we consider datasets characterized by different spatial distributions and different amounts of moving objects. Also, we study how K-NN_{GPU}'s behaves according to different amounts of nearest neighbours per query (k).

As stated in the introductory part of the experimental section, we use a kd-tree leaf size equal to 32 for K-NN_{CPU}, since this assures to the CPU competitor to have the best possible performances within our experimental setting.

For what regards K-NN_{GPU}, we set $th_{quad} = 12k$ when $32 \leq k \leq 256$ since this assures, on average, the best performance. In the other cases we use $th_{quad} = 192, k < 32$ and $th_{quad} = 2048, k > 128$.

Varying the amount of moving objects. In the following batch of experiments we study how K-NN_{GPU} compares with respect to K-NN_{CPU} when varying the amount of moving objects, since this influences the overall amount of distances to be com-

puted per query. To this end we consider all the three different spatial distributions, i.e., uniform, gaussian and network-based. The only dataset characteristics varied across the experiments are (i) the spatial distribution and (ii) the amount of moving objects between 100K and 1500K; the other ones follow the defaults specified in Table 3.1.

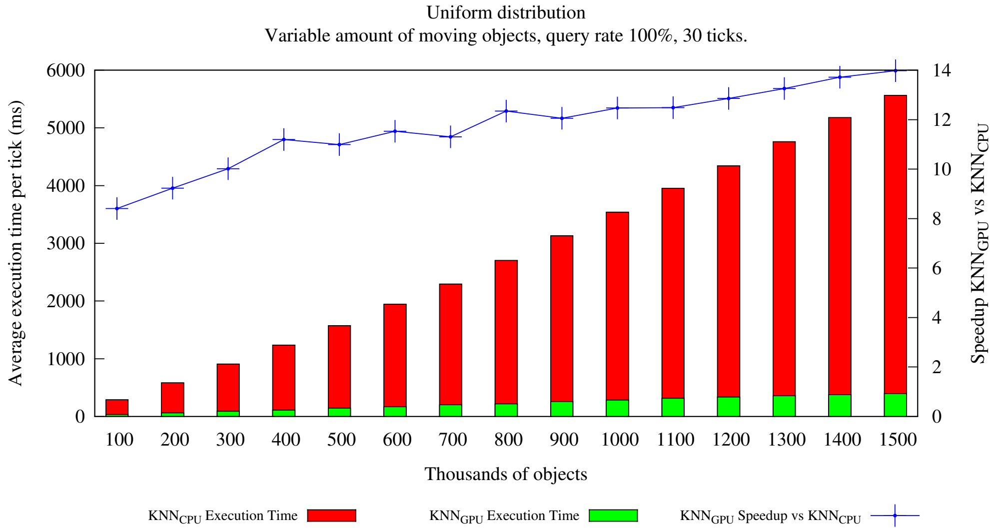


Figure 3.9: K-NN$_{GPU}$ vs K-NN$_{CPU}$, variable amount of moving objects, uniform datasets.

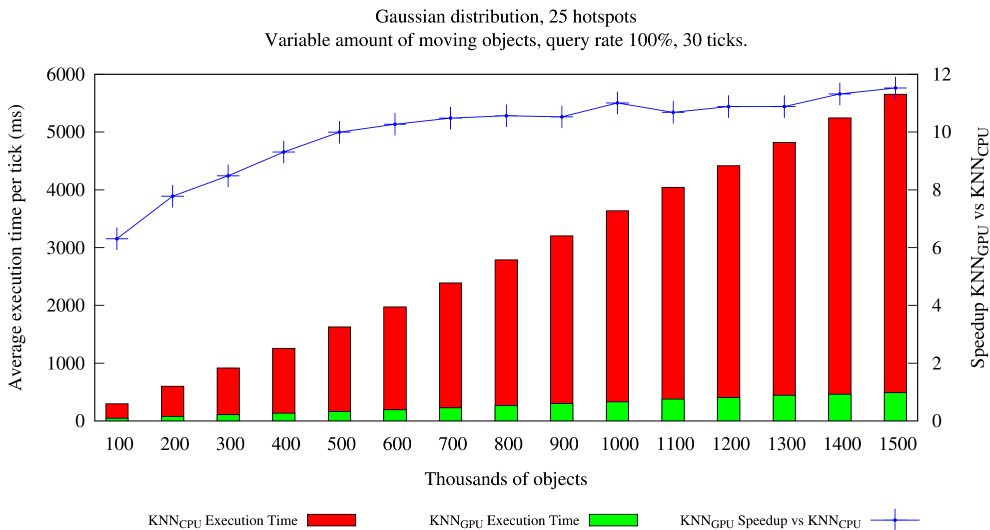


Figure 3.10: K-NN$_{GPU}$ vs K-NN$_{CPU}$, variable amount of moving objects, gaussian datasets, 25 hotspots.

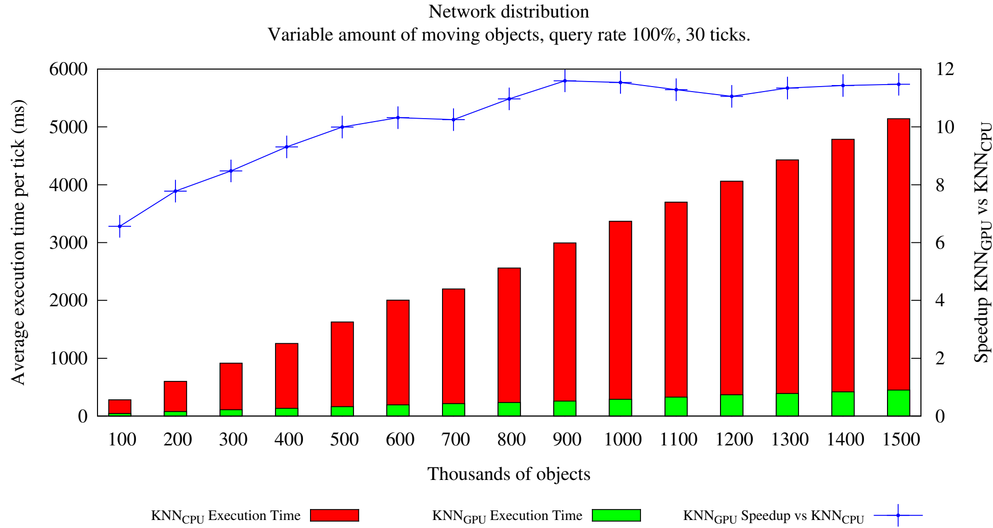


Figure 3.11: K-NN$_{\text{GPU}}$ vs K-NN$_{\text{CPU}}$, variable amount of moving objects, network-based (San Francisco) datasets.

From Figures 3.9, 3.10 and 3.11 we see how, in general, K-NN$_{\text{GPU}}$'s speedups increase whenever the amount of objects increases, since the amount of calculations (in terms of distances to compute) increases, thus making the query processing more and more a compute-intensive task - therefore favouring K-NN$_{\text{GPU}}$. We also notice how improvements become negligible once the amount of objects gets very large.

The other main observation relates to the skewness: speedups achieved with skewed distributions are slightly lower than those achieved with uniform distributions; this fact is expected, since the quadtree-based indexing used by K-NN$_{\text{GPU}}$ cannot totally avoid imbalances between single tasks. Moreover, skewed distributions typically require slightly more computations due to the objects tendency to form clusters.

Varying the neighbours lists size, *k*. In this batch of experiments we want to study how K-NN$_{\text{GPU}}$ compares with respect to K-NN$_{\text{CPU}}$ when varying the neighbours list size k. k represents a key parameter since it directly influences the overall amount of distances to compute, the output size and the average amount of iterations per query (we note that the latter depends on th_{quad} as well).

We consider, again, all the three spatial distributions. This time, however, we use a fixed amount of objects, 1 million, while varying k in the [1,512] range. All other dataset characteristics are set to their defaults, according to Table 3.1.

From Figures 3.12, 3.13 and 3.14 we see how increasing k up to a certain point has positive effects on the performances, since the whole query processing task becomes more and more compute-intensive while GPU caching is still able to cope effectively with the increased amounts of results per query.

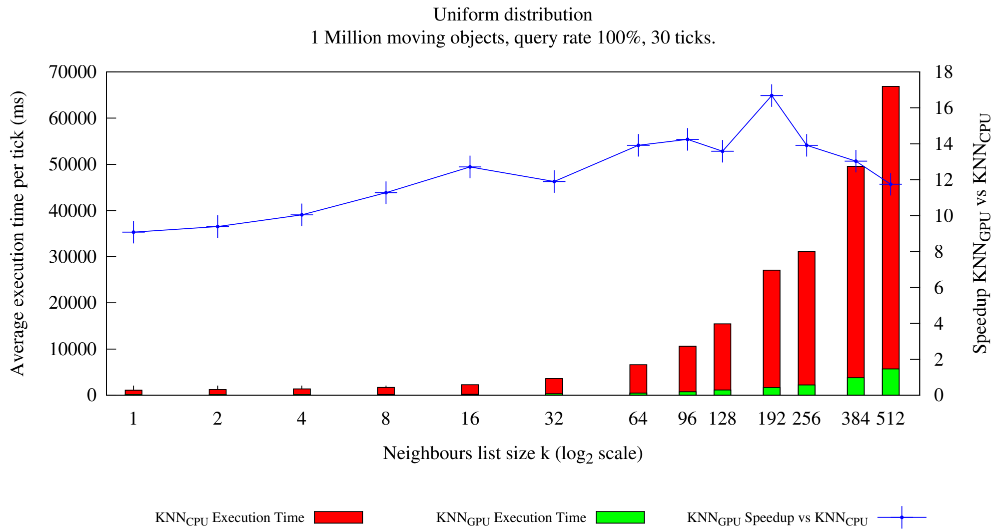


Figure 3.12: K-NN$_{GPU}$ vs K-NN$_{CPU}$, variable neighbours list size k, uniform dataset (1M objects).

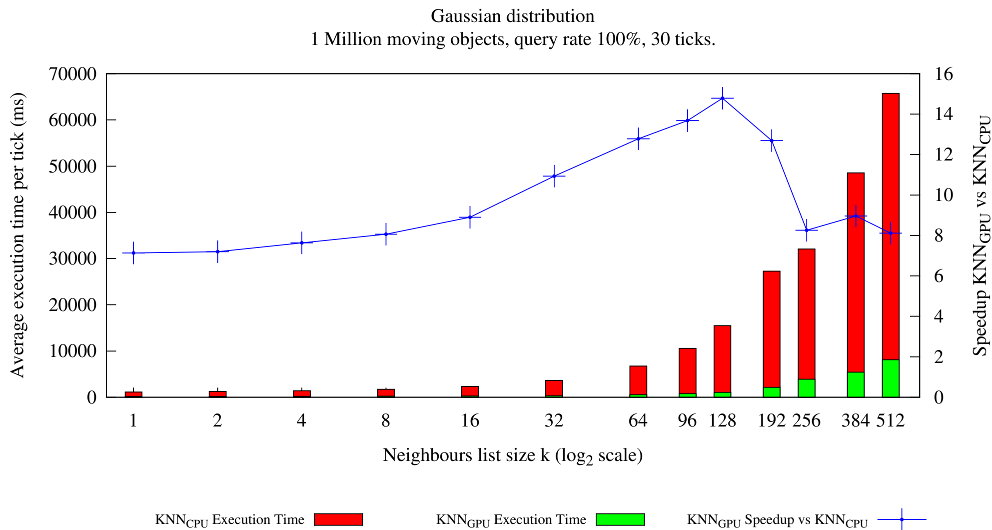


Figure 3.13: K-NN$_{GPU}$ vs K-NN$_{CPU}$, variable neighbours list size k, gaussian dataset (25 hotspots, 1M objects).

However, when k gets very high we observe performance degradation due to a decreased GPU caching efficiency. This is expected, considering the linear layout (Figure 3.1b) used for the queries result set.

For what relates to the skewness, we observe again how it mildly influences negatively K-NN$_{GPU}$'s performances with respect to the execution times observed with uniform distributions.

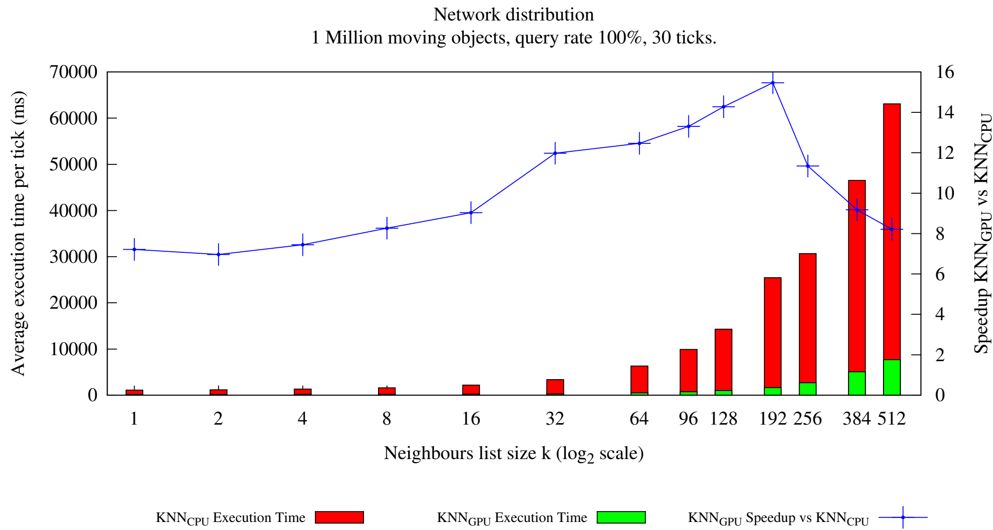


Figure 3.14: K-NN$_{\text{GPU}}$ vs K-NN$_{\text{CPU}}$, variable neighbours list size k, network dataset (1M objects).

Improving the memory throughput – K-NN$_{\text{GPU}}^{COALESCE}$. In the last batch of experiments we saw how K-NN$_{\text{GPU}}^{CACHE}$'s performance heavily degrades whenever k gets high. In the next batch of experiments we want to demonstrate how these cases can be tackled more effectively by using the strategy proposed for K-NN$_{\text{GPU}}^{COALESCE}$. We repeat the very same batch of experiments conducted in Figures 3.12, 3.13, and 3.14, however including and putting the focus on K-NN$_{\text{GPU}}^{COALESCE}$ as well.

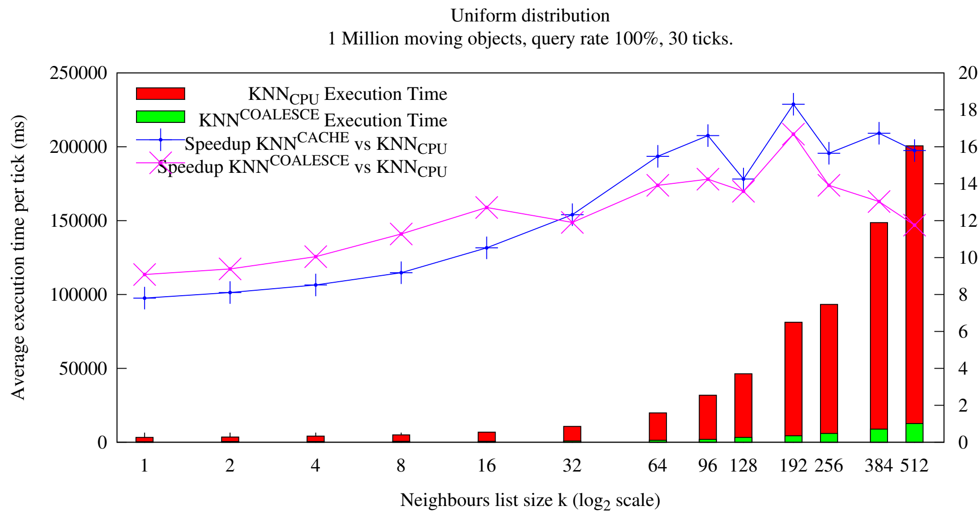


Figure 3.15: K-NN$_{\text{GPU}}^{COALESCE}$ vs K-NN$_{\text{GPU}}^{CACHE}$ vs K-NN$_{\text{CPU}}$, variable neighbours list size k, uniform dataset (1M objects).

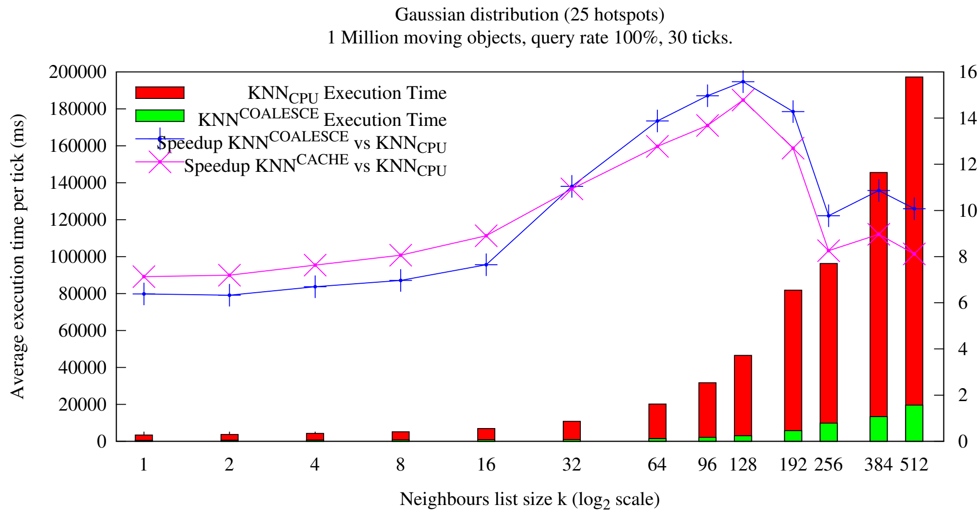


Figure 3.16: K-NN$_{\text{GPU}}^{COALESCE}$ vs K-NN$_{\text{GPU}}^{CACHE}$ vs K-NN$_{\text{CPU}}$, variable neighbours list size k, gaussian dataset (25 hotspots, 1M objects).

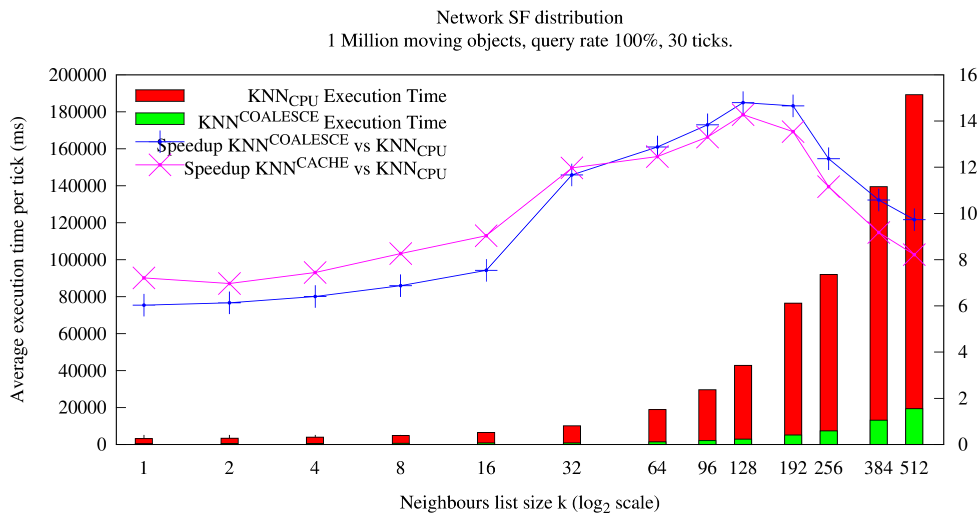


Figure 3.17: K-NN$_{\text{GPU}}^{COALESCE}$ vs K-NN$_{\text{GPU}}^{CACHE}$ vs K-NN$_{\text{CPU}}$, variable neighbours list size k, network dataset (1M objects).

From Figures 3.15, 3.16 and 3.17 we observe how K-NN$_{\text{GPU}}^{COALESCE}$ outperforms K-NN$_{\text{GPU}}^{CACHE}$ whenever k is equal or greater than the size of a warp ($k \geq 32$), while we have the opposite whenever k is lower. [5] These results can be explained by observing some key facts.

[5] We remember once more that in current NVIDIA GPUs the size of a warp is equal to 32.

First, K-NN$_{\text{GPU}}^{COALESCE}$'s strategy starts to become effective once threads inside a warp get fully utilized and the required amount of nearest neighbours per query gets more and more relevant, thus making the cache-only strategy on which K-NN$_{\text{GPU}}^{CACHE}$ relies less effective in terms of memory throughput. Indeed, this combination of factors is reflected in the performance trend observed in all the Figures. Second, in order to exploit coalescing we need a proper access pattern, which in turn requires some computational overhead in order to orchestrate the computations accordingly. Whenever k is low such overhead, coupled with a slight thread underutilization when updating the query lists, slightly penalizes K-NN$_{\text{GPU}}^{COALESCE}$.

In conclusion, K-NN$_{\text{GPU}}^{CACHE}$ better fits those cases where $k < 32$, while K-NN$_{\text{GPU}}^{COALESCE}$ covers the remaining ones. As a consequence, one should resort to the strategy which better adapts to the tackled scenario.

3.4.4 (S4) Bandwidth analysis

The goal of this study is to observe how the bandwidth β of a given system varies with respect to relevant run-time or dataset parameters. In the k-NN queries case these parameters are the *amount* of moving objects, the *query rate*, the amount of nearest neighbours per query k and the *skewness* affecting the spatial distribution.

From lemma 2 in Section 1.1.4 we remember that the system bandwidth β, expressed as the amount of queries processed per time unit (indeed, we use the second), is one of the crucial parameters in order to determine a suitable tick duration Δt, along with a given latency requirement λ and a maximum amount of queries which may occur during Δt, Q_{max}. Since λ and Q_{max} are fixed, the crucial parameter is β.

For what regards K-NN$_{\text{GPU}}$, we set $th_{quad} = 12k$ when $32 \leq k \leq 256$ since this assures, on average, the best performance. In the other cases we use $th_{quad} = 192, k < 32$ and $th_{quad} = 2048, k > 128$.

Figure 3.18 presents the results of the first batch of experiments, where we test the behaviour of β with respect to different amounts of objects and degrees of skewness. In order to conduct these experiments a set of gaussian datasets is considered, each characterized by a specific amount of moving objects and skewness degree.

From the Figure we see how the system bandwidth increases (however, the increase rate flattens out at some point) whenever the amount of objects increases. This can be explained by observing that, even if the action of increasing the amount of objects yields both an increase of the GPU resource usage efficiency and an increase in the amount of computations per query, the former factor mainly determines the behaviour of the observed performance trend.

For what is related to the skewness, we observe how skewed distributions affect negatively the bandwidth since they entail the creation of more uneven GPU workloads. This is similar to the behaviour observed in Figure 2.25 in the range queries case.

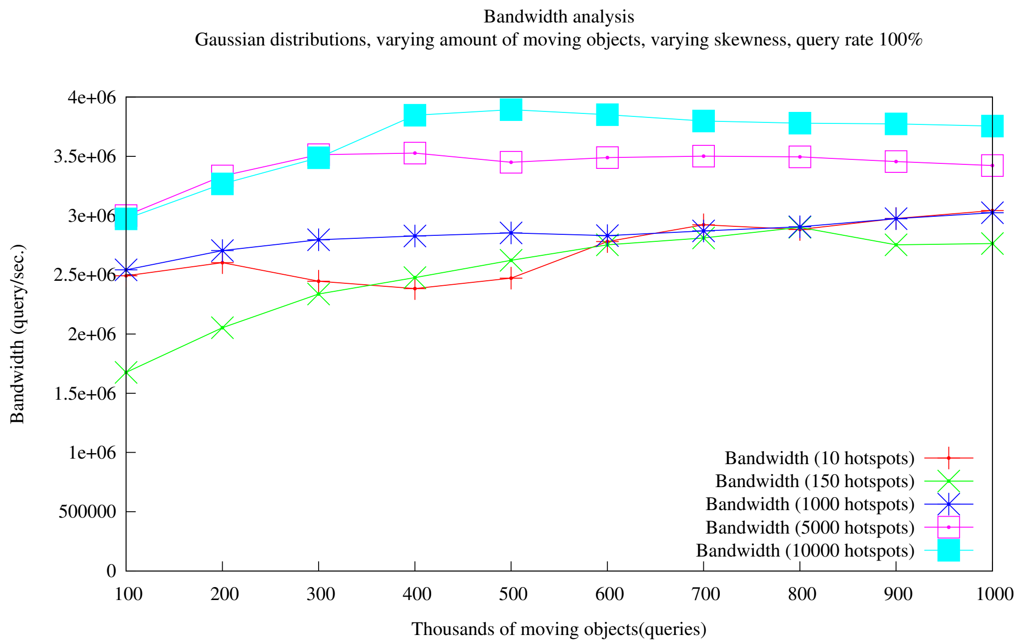


Figure 3.18: System bandwidth analysis when varying the amount of moving objects or the dataset skewness.

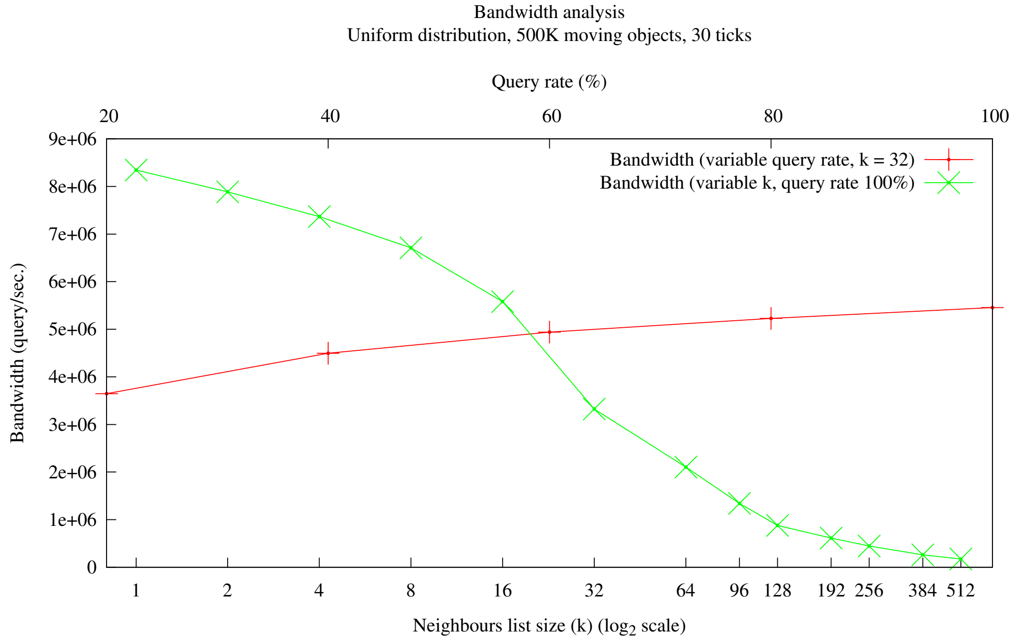


Figure 3.19: System bandwidth analysis when varying the query rate or the neighbours list size k.

Figure 3.19 reports the results related to the second batch of experiments, where we analyze the behaviour of β with respect to the query rate (we observe it corresponds to changing Q_{max}) and the neighours list size k. For what relates the query rate we observe, analogously to the experiment conducted for range queries (Figure 2.26), how the bandwidth increases whenever this parameter is increased. We briefly remember that this phenomenon may seem counter-intuitive at first, since the action of increasing the query rate has the effect of increasing linearly (and not quadratically) the amount of containment tests and results produced. These increases, however, are compensated by an increased efficiency of the system. In other words, GPU resources are more utilized and thus better exploited, in turn increasing the overall bandwidth. We note that this behaviour can be replicated with any spatial distribution.

For what relates the neighbours list size k, we see how increasing this parameter has the effect of decreasing the bandwidth, due to an increased amount of computations needed to compute the queries result set.

II

Second part

4

Detecting avoidance behaviours between moving objects

4.1 Introduction and Motivation

Current advances in mobile technologies have increased the interest in mobility data analysis in several application domains, such as security, smart cities, transportation systems, urban planning, biological studies and so on. As a consequence, several algorithms have been proposed for discovering various types of behaviors in trajectory data, such as T-patterns [50], flocks [51, 52], meet [53], periodic movements [54, 55], anomalous traffic patterns [56], chasing [57], etc. In [58], a taxonomy with different types of trajectory behaviors is proposed, while a summary of the most well known trajectory behaviors (also called patterns) is presented in [4].

In this work we focus on *avoidance behaviors between trajectories*, a new type of pattern which has not been much explored in the literature. For the purposes of this work it is convenient to distinguish between two main classes of works about the general concept of *avoidance* in mobility data: *collision avoidance* and *avoidance detection*. *Collision avoidance* is a well studied and established field where the main objective is to suggest in *real-time* a new route (trajectory) to avoid collisions. There is a vast literature on this topic and we defer further details to Section 4.2. On the other hand, *avoidance detection* tackles the problem of detecting whether a moving object has avoided a static object (area), as shown in Figure 4.1(a), or a moving object has avoided another one, as shown in Figures 4.1(b, c, and d), by analyzing *historical movement traces*. Our contribution belongs to this second class of works, so we focus on it.

Static avoidance detection can be interesting for discovering suspicious behaviors as, for instance, objects avoiding a surveillance camera, a police patrol, or speed controllers. A first treatment of this type of avoidance is proposed in [59]. Avoidance detection *between moving objects* is useful in several application domains. For example, in security applications it may reveal suspicious behaviors among people, such as criminals or terrorists that avoid policemen. In marine surveillance, ships with illicit products or illegal immigrants may avoid coastguard boats. In computer games avoidance behaviors may help to characterize classes of avoided enemies,

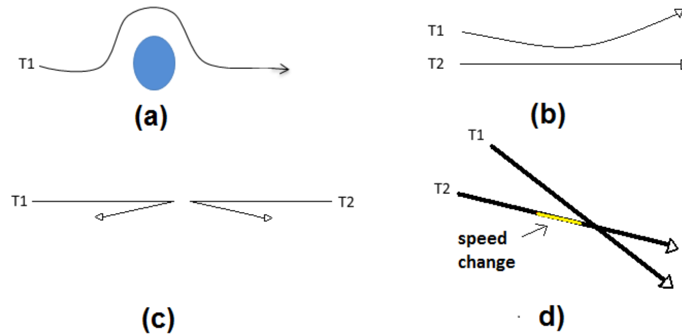


Figure 4.1: Different kinds of avoidance behaviors: avoidance with respect to a static object (a), and avoidance between moving objects: *individual* (b), *mutual* (c), and *individual* induced by a change in speed (d).

while in soccer games it may be useful to analyze players avoiding markers. In zoological studies, avoidance detection may reveal how preys avoid predators (e.g., at which distance, by changing direction or changing speed). The discovery of this type of avoidance is more challenging than detecting avoidances between moving objects and static objects/areas, and the first questions that raises are: what are the main features that characterize an avoidance between two moving objects? Who is avoiding who? At which distance two objects initiate an avoidance?

In this work we want to formalize the concept of trajectory avoidance behavior and to identify every instance of this behavioral pattern in historical movement traces. Figure 4.1 shows three examples of avoidance behavior addressed in this work: Figure 4.1(b) shows an example of trajectory avoidance where T_1 avoids trajectory T_2 by changing its direction. Figure 4.1(c) shows a mutual avoidance behavior, where both T_1 and T_2 avoid each other by changing their direction. In Figure 4.1(d), although both trajectories have a spatial intersection relationship, T_2 avoids T_1 by slowing down the speed in order not to spatio-temporally intersect T_1.

This specific problem has not received much attention in the literature. [60] makes a first attempt to address trajectory avoidance detection looking for attraction and avoidance relationships between pairs of trajectories. In general terms, this work considers the frequency of meetings to define an avoidance. When a pair of objects frequently move close to each other an attraction relationship is characterized, whereas an avoidance relationship occurs in the opposite case. As a result, this method measures the degree of attraction or avoidance between two trajectories.

We claim that an avoidance *between trajectories*, as depicted in Figures 4.1(b,c,d), is characterized by a change of movement behavior exhibited by at least one trajectory. Consequentially, in order to detect such changes we need to produce some kind of forecasts and properly compare these with actual trajectories data.

The contributions presented in this chapter can be summarized as follows: (i) we introduce a framework which defines what is an avoidance between pairs of

trajectories, considering changes of behavior in speed or direction; (ii) we present an algorithm able to automatically detect avoidances between pairs of trajectories and to work on real-world datasets where trajectories may be possibly characterized by different sampling rates; (iii) we propose a criteria to classify any avoidance as *weak*, *mutual* or *individual*, on the basis of factual evidence. Finally, (iv) we introduce the concept of fused detector in order to analyze any dataset with different sets of parameters, allowing to possibly increase the quantity and the quality of the results returned by the algorithm.

The rest of the chapter is organized as follows: Section 4.2 presents the related work. Section 4.3 introduces some basic definitions. Section 4.4 illustrates the new definitions for avoidance detection, while Section 4.5 proposes an algorithm to compute avoidance behavior. The chapter ends by describing the experiments on real trajectories in Section 4.6. We delegate the conclusions and possible directions of future research to the conclusive chapter of the thesis.

4.2 Related Work

As mentioned before, in the domain of mobility data we can distinguish between two main classes of works about the concept of avoidance: *collision avoidance* and *avoidance detection*.

Collision avoidance deals with models, systems, and practices designed to prevent vehicles, such as cars, ships, airplanes and so on, from colliding with other vehicles. Consequentially, the focus is on detecting a future collision and change (or suggest a change) the current route of one or more of the involved vehicles to avoid a collision. All these operations must be carried on in *real-time*. There is a vast literature involving various kinds of vehicles, for example cars ([61], [62], [63], [64]), ships ([65],[66]) and aircrafts ([67], [68], [69]).

In [61] the proposal is to minimize the safety distance error and to regulate the relative speed between two vehicles, so to avoid rear-end collision, using hierarchical longitudinal control. In [62] it is proposed a real-time method for computing a car trajectory towards a safe final state, as soon as an endangering obstacle is detected by a sensor (e.g. radar or lidar). Two scenarios are considered: a car which is overtaking another (slower) car in the same lane and an overtaking car which faces another car coming from the opposite direction. The solution is obtained from a simplified car model based on two control variables (steering velocity and braking force), state variables (speed, yaw angle, yaw angle rate, the center of gravity and direction) and a state dynamics defined by a system of differential equations. [63] presents experimental results for an active control intersection collision avoidance system implemented on modified Lexus test vehicles. The system utilizes vehicle-to-vehicle dedicated short-range communications to share safety critical state information. Safety is achieved in potential collision scenarios by controlling the speed of both vehicles with automatic brake and throttle commands. Another approach for car

collision avoidance considers pedestrians [64], where the use of stereo cameras on board of vehicles supports the detection of pedestrians with the aim of avoiding collisions between cars and pedestrians. In the domain of ships, Liu [65] proposes a fuzzy-neural inference network that learns a set of examples from a set of rules defined by the International Regulations for Preventing Collisions at Sea. Based on the learned examples, the method suggests direction changes of the ship to avoid a possible collision. The main input data are the ships' direction and speed, the distance between them and the type of water area (blue water, coast, ..f rules defined by the International Regulations for Preventing Collisions at Sea. Based on the learned e.). The model only considers cases where an encounter situation is already detected. The output is the set of actions to avoid the collision. In the aircraft domain, [68] considers a set of aircrafts where each one has a known destination and the related trajectory is represented as a straight line going from its current location to the destination. The speed of the aircrafts is known and constant, and it is assumed that aircrafts fly in layers. The contribution of the paper is a linear model to modify aircraft routes in order to avoid a collision when two or more aircrafts become sufficiently close to each other. The model considers real dynamics constraints to be more realistic.

Another domain where collision-avoidance is well studied is robotics. In this domain, when a robot is planning its route (its possible trajectory) it should consider known obstacles and avoid them. Some works include [70], [71], [72], [73]. [72], for instance, uses a dynamical system-based approach to deviate the robot from the obstacles. [73] proposes a behaviour-based multi-robot collision avoidance to efficiently coordinate the simultaneous navigation of large robot teams.

In summary, the *leitmotiv* of collision avoidance is to establish a set of actions, in real-time, to prevent collisions. To this end, the related works take into account both the physical properties and the type(s) of the (moving) objects considered.

For what concerns *avoidance detection*, which is the domain covering our work, the goal is to determine whether a moving object has avoided another object, may be it static or moving. We note that this domain is remarkably different from the former one. Indeed, while in *collision avoidance* the main goal is to change the route of a moving object considering some of its physical properties (e.g. mass, center of gravity, steering angle), in *avoidance detection* the goal is to discover if a trajectory has deviated from another trajectory considering only historical movement traces. In avoidance detection, to the best of our knowledge, there are basically two works: [59] and [60]. The former discovers moving objects that avoid static objects, and does not search for avoidances between moving objects. The latter is closer to our work, in that it looks for avoidance behaviors between moving objects. Still, as mentioned before, [60] searches for general avoidance and attracting relationships based on frequency of meetings, since it proposes a statistical approach based on permutation test: for each pair of moving objects the method outputs a value ranging in the interval $[0, 1]$ which expresses a global estimate of the level of attraction or avoidance between them.

With respect to the aforementioned works our work has a different objective, namely identifying each single occurrence of avoidance behavior between moving objects. Moreover, we distinguish various kinds of avoidances trying to determine who is avoiding who. More precisely, an avoidance occurrence is defined as a situation in which two objects are moving towards the same area, but either one or both change behavior whenever they come close enough to be aware of each other.

4.3 Preliminaries

Moving objects are entities having a time variant position, uniquely determined at each time instant and possibly undefined. A trajectory is a continuous part of the movement of an object [3]. For the sake of simplicity, in the following we will restrict to the 2D Euclidean space, but note that the generalization to higher dimensional spaces is straightforward.

Definition 10. *[Movement and trajectory] The* movement *of an object o is a continuous function $M_o : R^+ \rightarrow R^2$ from the real positive numbers, representing time instants, to 2D space. Given an object o and a time interval* [tBegin, tEnd]*, a trajectory T is the restriction of the movement M_o of the object to the given time interval. The spatio-temporal position of the object at* tBegin *(resp.* tEnd*) is called the* Begin *(resp.* End*) of the trajectory.*

Often, in mobility applications, trajectories are only partially known, usually at specific time instants that correspond to position update actions.

Definition 11 (Trajectory point). *A trajectory point, or trajectory sample, of a trajectory T is a tuple (x, y, t), where $T(t) = (x, y)$ is the object position at time t (called the* timestamp *of the trajectory point).*

The set of known positions of a moving object during the definition interval of a trajectory is named *trajectory track*, or trajectory *sampling*.

Definition 12 (Trajectory track). *Given a temporally ordered sequence $\langle t_1, \ldots, t_n \rangle$ of timestamps, the track of a trajectory T for the given timestamps is the temporally ordered sequence of trajectory points $\langle p_1, \ldots, p_n \rangle$, where $p_i = (x_i, y_i, t_i)$ and $(x_i, y_i) = T(t_i)$.*

A finite sequence of trajectory points can be paired with a finite sequence of interpolation functions that describe in an analytical and continuous way the movement of the object between each pair of consecutive trajectory points in the sequence. This sequence of pairs of trajectory points and interpolation functions is named *continuous trajectory representation.*

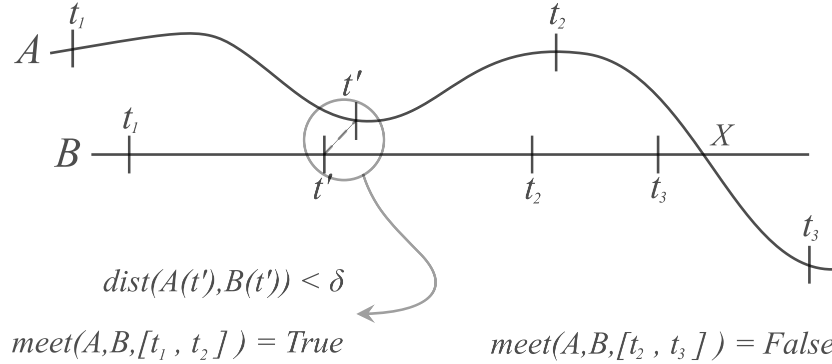


Figure 4.2: *A meets B* (the distance in t' is less than δ) during $[t_1, t_2]$ but not during $[t_2, t_3]$.

4.4 Avoidance

An *avoidance* between two moving objects occurs when both are moving towards the same area at the same time, but either one or both change their behavior when they come close enough to be aware of each other. In the following, with a slight abuse of terms, we will indistinctly use the terms trajectory and object when referring to the avoidance. This is acceptable since the trajectories referring to the same object do not temporally overlap.

In order to define the avoidance concept, we first introduce the predicates *meet* and *will-meet*. The first one expresses the fact that in a certain interval two trajectories become sufficiently close to be considered in contact, whereas the second one states that the *forecast* of these trajectories, determined by some technique on the basis of some observed behavior, will lead to a contact.

Definition 13 (**Meet**)**.** *Given two trajectories* T_a *and* T_b*, a time interval* $[t_1, t_2]$ *and a distance threshold* δ*, we define the predicate* meet_δ *as*

$$\text{meet}_\delta(T_a, T_b, [t_1, t_2]) \equiv \exists t \in [t_1, t_2].\, dist(T_a(t), T_b(t)) < \delta$$

where $dist(p_a, p_b)$ *is the Euclidean distance of the points* p_a *and* p_b*.*

When the predicate is satisfied we also say that T_a *meets* T_b during $[t_1, t_2]$ with threshold δ. In case the threshold δ is evident from the context it will be omitted in the predicate notation, writing *meet* instead of $meet_\delta$.

Figure 4.2 illustrates a pair of trajectories, A and B, during the time interval $[t_1, t_3]$. The predicate *meet* is true for $[t_1, t_2]$, since at time t' the distance of the two trajectories is less than δ. Note that for the interval $[t_2, t_3]$, even if the two trajectories have a spatial intersection X the *meet* predicate is false since the distance

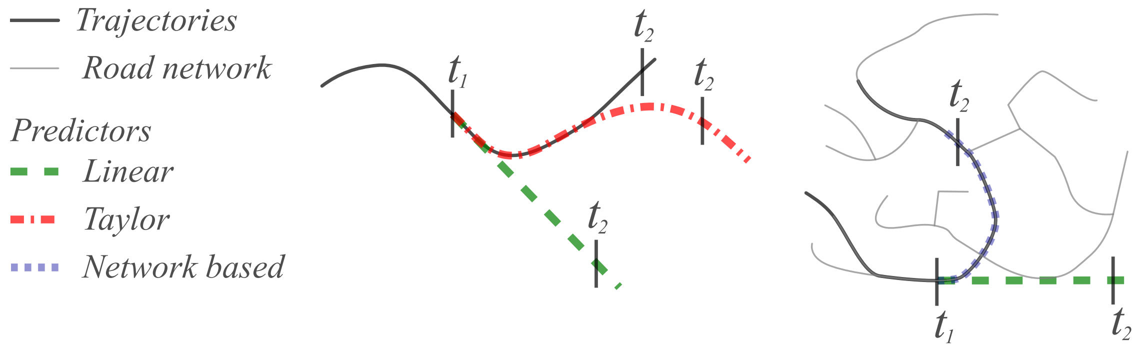


Figure 4.3: Different kinds of predictors based on several interpolations and on movement constraints (road network).

between the two moving objects at any instant $t \in [t_2, t_3]$ is always greater than δ. In fact, A and B cross X at different time instants.

Having a way of predicting the movement of the trajectories on the basis of what happened in the past, we can establish if two trajectories will meet each other or not. We next introduce an abstract notion of *movement predictor* clarifying which are the expected properties.

Definition 14 (**Movement predictor**). *Given a movement domain, consisting of all possible movement functions,* $\mathcal{M} = \{M \mid M : R^+ \to R^2\}$*, and a temporal domain* R^+*, a* movement predictor *is a functional*

$$forecast : \mathcal{M} \times (R^+ \times R^+) \to \mathcal{M} \;\; such \;\; that$$

$$\forall M, M' \in \mathcal{M}.\; M\mid_{]0,t1]} = M'\mid_{]0,t1]} \implies forecast(M, [t1, t2]) = forecast(M', [t1, t2])$$

The functional *forecast* maps a movement function M to its forecast movement into the interval $[t_1, t_2]$ by exploiting only the behavior of M in the past interval $]0, t_1]$. It can be defined in different ways depending on the contexts. Commonly, it is based on the assumption that the behavior does not change with respect to the recent past of the trajectory. For instance, in case of objects that move freely in space, we can use the Taylor series in order to estimate the next positions of the trajectory, and the more terms of these series we compute, the more precise we obtain the approximation, e.g., the first term preserves the direction, the second one the curvature. On the other hand, if the object movement is constrained by a network, we can exploit the network to predict where the object is going.

Figure 4.3 shows several different forecast functionals for the interval $[t_1, t_2]$. The solid line represents the actual trajectory. The dashed, green line models a linear prediction, preserving the direction at time t_1; the dash-dot, red line shows a forecast based on a higher order Taylor series, preserving several derivatives of the trajectory; and the dotted, blue line in the right figure illustrates a prediction based on the knowledge of the road network. In the following examples and in the

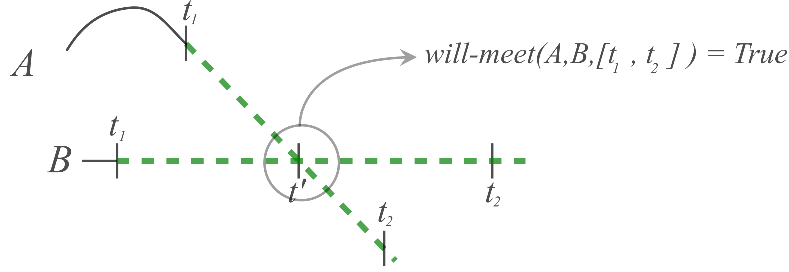


Figure 4.4: According to the forecasts, A *will meet* B (the distance will be less than δ) at some time t' in the time interval $[t_1, t_2]$.

experimental section, we will use a linear movement predictor and we will employ the same notation as in Figure 4.3: solid lines for actual trajectories and dashed lines for the linear forecast.

Definition 15 (**Will-meet**). *Given two trajectories* T_a *and* T_b*, a time interval* $[t_1, t_2]$*, a movement predictor* forecast *and a threshold* δ*, we define the predicate* will-meet$_\delta$ *as*

$$\text{will-meet}_\delta(T_a, T_b, [t_1, t_2]) \equiv \text{meet}_\delta(\text{forecast}(T_a, [t_1, t_2]), \text{forecast}(T_b, [t_1, t_2]), [t_1, t_2])$$

When the predicate is satisfied we also say that T_a is *expected to meet* T_b during $[t_1, t_2]$ with threshold δ. In case the threshold δ is evident from the context it will be omitted in the predicate notation, writing *will-meet* instead of *will-meet*$_\delta$.

Figure 4.4 illustrates an example where the predicate *will-meet* holds: A and B are supposed to meet at time t', provided that they maintain the same speed and direction they had at time t_1 (we used a linear movement predictor).

With these definitions we define an avoidance between trajectories as an event that happens in a time interval where the prediction is a meet between two trajectories but this meet does not occur.

Definition 16 (**Avoid**). *Given two trajectories* T_a *and* T_b*, a timestamp* t*, a time interval* $[t_1, t_2]$*, and a threshold* δ*, we define the predicate* avoid$_\delta$ *as*

$$\text{avoid}_\delta(T_a, T_b, [t_1, t_2]) \equiv \text{will-meet}_\delta(T_a, T_b, [t_1, t_2]) \wedge \neg\text{meet}_\delta(T_a, T_b, [t_1, t_2])$$

When the predicate is satisfied we say that T_a and T_b *avoid* to meet, with threshold δ, during $[t_1, t_2]$. In case the threshold δ is evident from the context it will be omitted in the predicate notation, writing *avoid* instead of *avoid*$_\delta$.

The duration of the time interval $[t_1, t_2]$ is a measure of the future awareness of the moving objects (e.g., a person or an animal), that is the amount of time they are able to reliably forecast all of the involved trajectories to avoid collisions or encounters.

Figure 4.5 shows two different cases of avoidance in which trajectory B exactly fulfills the prediction (dashed green), whereas trajectory A behaves differently with

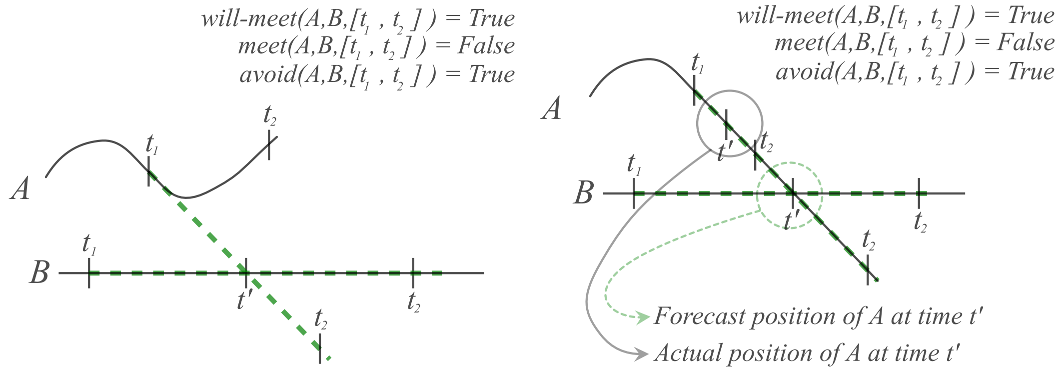


Figure 4.5: A *avoids* B during the time interval $[t_1, t_2]$: meet is expected according to the forecast computed at time t_1 but no actual meet happened during the given time interval.

respect to the forecast (dashed green): on the left, A changes direction after time t_1, and on the right it reduces its speed. As a result, the forecast of the two trajectories (A and B) are expected to meet at time $t' \in [t_1, t_2]$, but the two trajectories do not actually meet since their distance is always larger than δ (omitted for simplicity in Figure 4.5) during $[t_1, t_2]$. Thus, in both cases the *avoid* predicate is true.

4.4.1 Avoidance Classification

The concept of avoidance is related to a change of behavior of one or both the involved objects such that a predicted meet does not occur. We define a new predicate, *change-behavior*, to better characterize the different kinds of avoidance. Informally, we regard the behavior of an object as *changed* during a time interval when there is a difference between the actual trajectory and its forecast that is sufficient to cause missed meets without any change in the other trajectory. Intuitively such a difference should have at least the same magnitude as the meet threshold δ.

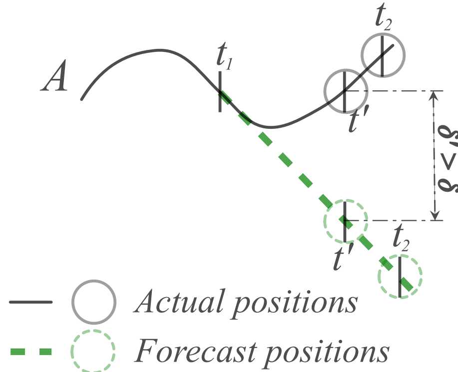
Definition 17 (Change-behavior). *Given a trajectory T, a time interval $[t_1, t_2]$, and a meet distance threshold δ, we define the predicate* change-behavior$_\delta$

$$\text{change-behavior}_\delta(T, [t_1, t_2]) \equiv \exists t \in [t_1, t_2], \text{dist}(\text{forecast}(T, [t_1, t_2])(t), T(t)) > \delta$$

where $\text{forecast}(T, [t_1, t_2])(t)$ *and* $T(t)$ *are respectively the forecast and the actual position for trajectory T at time t.*

When the predicate is satisfied we say that T *changes* its behavior, with threshold δ, during $[t_1, t_2]$. In case the threshold δ is evident from the context it will be omitted in the predicate notation, writing *change-behavior* instead of *change-behavior$_\delta$*.

Figure 4.6 focuses on trajectory A of Figure 4.5, showing how it changes its behavior. On the left it changes direction whereas on the right it reduces the speed with respect to the forecast (green dashed line). In both cases the distance at t'



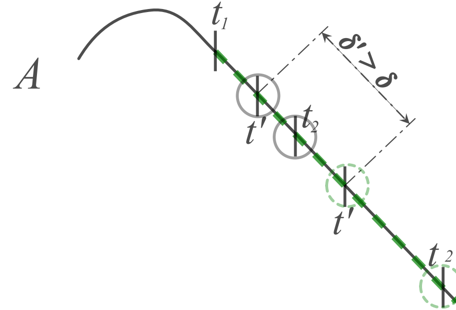


Figure 4.6: Trajectory A *changes behavior*: at some time t' during $[t_1, t_2]$ the distance of the actual position from the forecast position is greater than δ.

between the forecast and the actual trajectory is greater than δ, hence the predicate *change-behavior* holds.

Based on the above definition, we distinguish among *weak*, *individual* and *mutual* avoidance.

Definition 18 (Avoidance classification). *Given two trajectories T_a and T_b, and a time interval $[t_1, t_2]$ such that* $\text{avoid}_\delta(T_a, T_b, [t_1, t_2])$ *is true, we classify that avoidance in the following way:*

$$type_avoid_\delta(T_a, T_b, [t_1, t_2]) = \begin{cases} mutual & \text{if change-behavior}_\delta(T_a, [t_1, t_2]) \wedge \\ & \quad \text{change-behavior}_\delta(T_b, [t_1, t_2]) \\ weak & \text{if } \neg\text{change-behavior}_\delta(T_a, [t_1, t_2]) \wedge \\ & \quad \neg\text{change-behavior}_\delta(T_b, [t_1, t_2]) \\ individual & \text{otherwise} \end{cases}$$

In other words, we have a *mutual avoidance* when there is an evident change of behavior for both trajectories, an *individual avoidance* when only one trajectory significantly changes its behavior, and a *weak avoidance* when there is an avoidance despite the fact that the behavior changes are minimal for both trajectories.

Figure 4.7 shows examples of *mutual* and *weak* avoidance, whereas avoidances in Figure 4.5 are both *individual.* We recall that trajectories are represented in solid black, forecasts in dashed green, and their specific positions at a given time are circled, respectively in solid gray and dashed green. In the first case, on the left, A changes its behavior by reducing its speed and this is detected since there exists a time $t' \in [t_1, t_2]$ such that the distance of the forecast position from the actual position is greater than δ. The same happens for B that changes its direction. Since

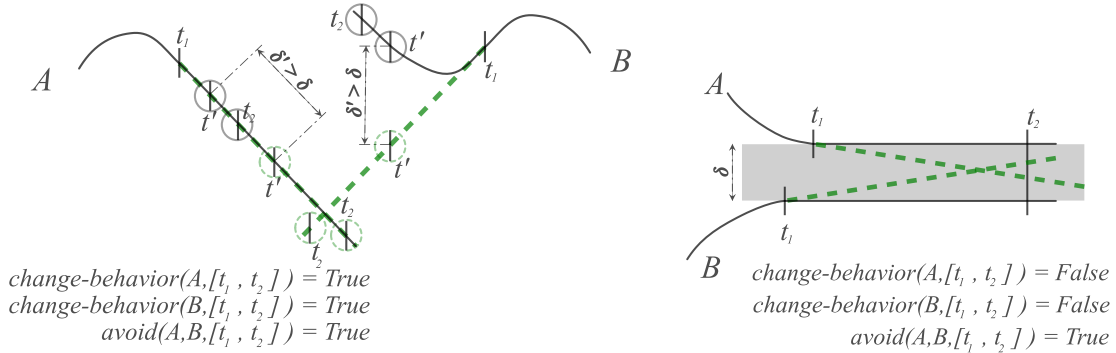


Figure 4.7: On the left, both A and B change behavior to avoid each other (*mutual* avoidance). On the right, according to the forecast A and B should meet but this does not happen even if both did not significantly change behavior (*weak* avoidance).

both trajectories changed behavior during interval $[t_1, t_2]$, this is a *mutual* avoidance. In the second case, on the right, at any time in $[t_1, t_2]$ for both trajectories the distance of the forecast position from the actual position is less than δ (the height of the gray rectangle). Nevertheless, trajectories were expected to meet but did not actually meet (their distance is larger than δ) and thus the *avoid* predicate is true. Since the *change-behavior* predicate is false for both trajectories, this is a *weak* avoidance.

4.4.2 Problem Statement

In this work we address two problems of increasing complexity regarding the detection of avoidance behaviors between moving objects: the *avoidance decision problem* (i.e., decide whether a pair of trajectories in a given temporal interval satisfies the predicate *avoid*) and the *avoidance search problem* (for every possible pair of trajectories find those time intervals such that the predicate *avoid* holds).

We define both problems on trajectory tracks since trajectory datasets usually consist of sets of samplings (one set per trajectory). To obtain the continuous representation of a trajectory we use an interpolation function, namely *interp*, and we denote as *interp*($\mathcal{T}$) the application of *interp* to the trajectory track $\mathcal{T}$. It is worth mentioning that in our experiments presented in Section 4.6 we use linear interpolation.

We check avoidances in intervals starting at the samples of a trajectory and the duration of the interval is based on a *look-ahead* time Δt. As said previously, Δt can be interpreted as a measure of the future awareness of moving objects. Consequently, Δt must be chosen according to their characteristics. For instance, pedestrians have a consistently smaller look-ahead time than big ships (e.g., cargos), since in the latter case, due to the tonnage and the size of the objects, changes of heading or speed are necessarily much slower.

Another parameter which is strictly related to the characteristics of the moving objects under analysis is the meet threshold δ. Indeed, if we consider two completely different classes of moving objects, e.g., pedestrians and ships, it is evident that the spatial extent of the actions needed to change their movements, as well as the distance at which they are considered close to another object of the same type, will be typically different and this fact can and should influence the selection of δ.

The first problem we address is the decision problem.

Definition 19 (**Avoidance decision problem**). *Given two trajectory tracks* $\mathcal{T}_a$,$\mathcal{T}_b$, *the set of their timestamps* $TS = \{timestamp(p) \mid p \in \mathcal{T}_a \vee p \in \mathcal{T}_b\}$, *a time instant* t, *such that* $t \in TS$, *a look-ahead interval* Δt, *and a meet threshold* δ, *the* avoidance decision problem *consists in determining whether*

$$\text{avoid}_\delta(interp(\mathcal{T}_a), interp(\mathcal{T}_b), [t, t + \Delta t]) \textit{ holds.}$$

The second problem we address is a *search* problem: for each pair of trajectories we look for all the timestamps belonging to one of the two trajectory tracks such that the predicate *avoid* holds and we determine the associated type of avoidance. As a first step we formulate this problem for a pair of trajectories in order to find the set of timestamps and the relative avoidance type where an avoidance between such trajectories is detected. Then the notion will be generalized to a set of trajectories.

Definition 20 (**Avoidances set**). *Given two trajectory tracks* $\mathcal{T}_a$,$\mathcal{T}_b$, *the set of their timestamps* $TS = \{timestamp(p) \mid p \in \mathcal{T}_a \vee p \in \mathcal{T}_b\}$, *a look-ahead time* Δt *and a meet threshold* δ, *we define*

$$avoidances_\delta(\mathcal{T}_a, \mathcal{T}_b) = \{(t, type) \mid t \in TS \ \wedge avoid_\delta(interp(\mathcal{T}_a), interp(\mathcal{T}_b), [t, t + \Delta t]) \wedge \\ type_avoid_\delta(interp(\mathcal{T}_a), interp(\mathcal{T}_b), [t, t + \Delta t]) = type\}$$

In words, a pair $(t, type) \in avoidances_\delta$ when t is a timestamp in one of the trajectory tracks and the predicate *avoid* holds for the two trajectories in the interval $[t, t + \Delta t]$. *type* is the kind of the detected avoidance.

Definition 21 (**Avoidance search problem**). *Given a set of trajectory tracks* $\mathcal{D}$, *a look-ahead time* Δt, *and a meet threshold* δ, *the* avoidance search problem *consists in finding the set of tuples*

$$\{(\mathcal{T}_a, \mathcal{T}_b, (t, type)) \mid \mathcal{T}_a \in \mathcal{D} \wedge \mathcal{T}_b \in \mathcal{D} \wedge (t, type) \in avoidances_\delta(\mathcal{T}_a, \mathcal{T}_b)\}$$

each consisting of a pair of trajectory tracks $\mathcal{T}_a, \mathcal{T}_b$, *a time instant* t *and a type type such that* $\mathcal{T}_a$ *and* $\mathcal{T}_b$ *avoid to meet, with threshold* δ, *after* t *for* Δt *time and the kind of avoidance is specified by type.*

In order to get a more compact representation of the result set, we next replace the set of timestamps for a couple of trajectories ($avoidances(\mathcal{T}_a, \mathcal{T}_b)$) with a disjoint set of intervals. The idea is to join two avoidances if the intervals where they are detected overlap. To this aim, we introduce the notion of *repeated avoidance* in an interval.

Definition 22 (**Repeated avoidance**). *Given two trajectory tracks* $\mathcal{T}_a$,$\mathcal{T}_b$*, the set of their timestamps* $TS = \{timestamp(p) \mid p \in \mathcal{T}_a \vee p \in \mathcal{T}_b\}$*, a look-ahead time* Δt*, and* $t_1, t_2 \in TS$ *we say that the interval* $[t_1, t_2]$ *contains a* repeated avoidance, *written* $repeated_avoid_\delta(\mathcal{T}_a, \mathcal{T}_b, [t_1, t_2], type)$*, when for any* $t \in [t_1, t_2]$ *there exists* $t' \in TS$ *such that* $t \in [t', t' + \Delta t]$ *and* $(t', _) \in avoidances_\delta(\mathcal{T}_a, \mathcal{T}_b)\}$*. Moreover,* $type = sup\{type' \mid t' \in [t_1, t_2] \cap TS \ \wedge \ (t', type') \in avoidances_\delta(\mathcal{T}_a, \mathcal{T}_b)\}$ *where*

$$sup(X) = \begin{cases} weak & if\ \forall x \in X\ x = weak \\ mutual & if\ \exists x \in X\ x = mutual \\ individual & otherwise \end{cases}$$

Hence, the compression of the set of avoidances consists of a set of *maximal* intervals satisfying the *repeated_avoid* predicate. The type of a repeated avoidance is the upper bound of the types of the avoidances occurring in the associated maximal interval.

Definition 23 (**Compression of the set of avoidances**). *Given two trajectory tracks* $\mathcal{T}_a$,$\mathcal{T}_b$*, the set of their timestamps* $TS = \{timestamp(p) \mid p \in \mathcal{T}_a \vee p \in \mathcal{T}_b\}$*, a look-ahead time* Δt*, the compression of the set* $avoidances_\delta(\mathcal{T}_a, \mathcal{T}_b)$ *is defined as follows:*

$$\begin{aligned} compressed_avoid_\delta(\mathcal{T}_a, \mathcal{T}_b) = \{([t_1, t_2], type) \mid\ & t_1, t_2 \in TS \\ & \wedge\ repeated_avoid_\delta(\mathcal{T}_a, \mathcal{T}_b, [t_1, t_2], type) \\ & \wedge\ [t_1, t_2] \quad maximal\} \end{aligned}$$

4.5 Algorithmic Framework

In this section, first we present an algorithm for solving the avoidance search problem (Section 4.5.1) and then we make some considerations on the choice of the right parameters to detect avoidances. As a consequence, we propose two different strategies for using our algorithm: the *simple detector* and the *fused detector*. The simple detector consists of a single execution of the algorithm with a fixed set of parameters whereas the fused detector consists of multiple executions of the algorithm with various parameter sets and the different results are merged into a single result set (Section 4.5.2).

4.5.1 An Algorithm for Avoidance Detection

In order to solve the problems posed in Section 4.4.2 we propose Algorithm 11 which, given a finite set of trajectory tracks $\mathcal{D}$, a meet threshold δ and a look-ahead time Δt, returns a set of avoidance behaviors A, consisting of tuples specifying a pair of trajectories, a compressed interval in which the avoidance is detected between such trajectories and the relative type of avoidance.

Algorithm 11: Avoidance behavior detection

Input: Finite set of trajectory tracks $\mathcal{D}$, meet threshold δ, look-ahead time Δt.
Output: Set of avoidance behaviors A.

1 **begin**
2 $A = \emptyset$
3 **foreach** $T_m, T_n \in \mathcal{D}$ *with* $m < n$ **do**
4 $t_m^{curr} = getFirstUsefulSampleTime(T_m, \Delta t)$
5 $t_n^{curr} = getFirstUsefulSampleTime(T_n, \Delta t)$
6 $t_m^{last} = getLastUsefulSampleTime(T_m, \Delta t)$
7 $t_n^{last} = getLastUsefulSampleTime(T_n, \Delta t)$
8 **while** $(t_m^{curr} < t_m^{last} \wedge t_n^{curr} < t_n^{last})$ **do**
9 **if** $(t_m^{curr} < t_n^{curr})$ **then**
10 $t^{curr} = t_m^{curr}$
11 **else**
12 $t^{curr} = t_n^{curr}$
13 **if** $(avoid_\delta(interp(T_m), interp(T_n), [t^{curr}, t^{curr} + \Delta t]))$ **then**
14 $type = weak$
15 **if** $(change_behavior(interp(T_m), [t^{curr}, t^{curr} + \Delta t]) \wedge$ $change_behavior(interp(T_n), [t^{curr}, t^{curr} + \Delta t]))$ **then**
16 $type = mutual$
17 **else if** $change_behavior(interp(T_m), [t^{curr}, t^{curr} + \Delta t]) \vee$ $change_behavior(interp(T_n), [t^{curr}, t^{curr} + \Delta t])$ **then**
18 $type = individual$
19 $([t_1, t_2], typelast) = poplastResult(A, T_m, T_n)$
20 **if** $t_2 + \Delta t >= t^{curr}$ **then**
21 $addToResultSet(A, T_m, T_n, [t_1, t^{curr}], sup\{typelast, type\})$
22 **else**
23 $addToResultSet(A, T_m, T_n, [t_1, t_2], typelast)$
24 $addToResultSet(A, T_m, T_n, [t^{curr}, t^{curr}], type)$
25 **if** $t_m^{curr} < t_n^{curr}$ **then**
26 $t_m^{curr} = nextSampleTime(T_m, t_m^{curr})$
27 **else if** $t_n^{curr} < t_m^{curr}$ **then**
28 $t_n^{curr} = nextSampleTime(T_n, t_n^{curr})$
29 **else**
30 $t_m^{curr} = nextSampleTime(T_m, t_m^{curr})$
31 $t_n^{curr} = nextSampleTime(T_n, t_n^{curr})$
32 **return** A

The algorithm starts by considering every possible pair of trajectory tracks in $\mathcal{T}$ and, for each pair, it determines the first and last *useful* sample timestamps with respect to the look-ahead time Δt, (functions *getFirstUsefulSampleTime* and *getLastUsefulSampleTime*, lines 4-7). More precisely, if t_m^{start} and t_m^{end} denote, respectively, the timestamps of the very first and last samples of a trajectory track T_m, then the timestamps of the first and last *useful* samples with respect to Δt are determined as $getFirstUsefulSampleTime(T_m, \Delta t) = min\{t \mid p \in T_m \ \wedge \ p = (x, y, t) \ \wedge \ t \geq t_m^{start} + \Delta t\}$ and $getLastUsefulSampleTime(T_m, \Delta t) = max\{t \mid p \in T_m \ \wedge \ p = (x, y, t) \ \wedge \ t \leq t_m^{end} - \Delta t\}$.

Once the first and last useful samples of both trajectory tracks are determined (if any), the algorithm starts scanning the trajectories from these samples (line 8). The sample with the smaller timestamp, t^{curr}, is chosen (lines 9-12). Then, the algorithm proceeds with the avoidance search. First, it checks whether there is a *weak* avoidance in the interval $[t^{curr}, t^{curr} + \Delta t]$ (line 13): if there is an avoidance the algorithm determines whether the avoidance is mutual (line 15) or individual (line 17) by properly testing the *change-behavior* predicate. It could happen that there is not enough evidence to further specify the avoidance, hence it remains labeled as a *weak* avoidance.

Once the avoidance has been detected, the algorithm removes the last avoidance, identified by $[t_1, t_2]$ with type *typelast*, for the two trajectories under investigation T_m, T_n (line 19). It checks whether the interval $[t_1, t_2 + \Delta t]$ overlaps the interval $[t^{curr}, t^{curr} + \Delta t]$ and in this case it merges the two avoidances, inserting in the result set the interval $[t_1, t^{curr}]$ associated with the least upper bound between *typelast* and *type* (line 21). The definition of *sup* is given in Definition 22. Otherwise the algorithm inserts back the avoidance $[t_1, t_2]$ with type *typelast* in the result set and it adds also the new avoidance $[t^{curr}, t^{curr}]$ with *type* (line 23-24).

After having processed a sample, the function *nextSampleTime* returns the timestamp of the next sample(s) in the trajectory track(s) (lines 25-31).

As a final consideration we observe that our algorithm can be easily adapted to more complex processing pipelines, where preprocessing phases are allowed to filter out unnecessary information prior to the avoidance behavior detection phase, hence reducing the overall amount of data to be analyzed. Moreover, the algorithm can be also parallelized with respect to the set of trajectory pairs under investigation, since each pair can be processed separately in a core, thus entailing linear speedups with respect to the number of cores used.

4.5.2 Avoidance Detectors

When analyzing a dataset we have to take into consideration the possibility that, depending on the typology of the moving objects (e.g., cars, pedestrians, airplanes) and on the characteristics of their movements, different kinds of avoidance behaviors may require the usage of different δ thresholds in order to be detected. For example, different trajectory pairs having different average speeds may exhibit avoidances

characterized by different δ thresholds. Indeed, faster moving objects usually exhibit avoidances characterized by larger δ thresholds than those observed when considering slower moving objects. Another observation is related to the fact that, even when considering a set of trajectories having the same average speed, different kinds of avoidance behaviors may be exhibited by the trajectories, thus possibly requiring the usage of different δ thresholds in order to be (separately) detected.

In light of these considerations we present two different methods for analyzing a set of trajectories based on the algorithm proposed. We call these methods *simple detector* and *fused detector*.

The *simple detector* consists of a single run of the algorithm using a fixed pair of parameters Δt and δ. Given a pair of trajectory tracks $(\mathcal{T}_a,\mathcal{T}_b)$, a meet threshold δ and a look-ahead time Δt, the simple detector returns the set:

$$\mathcal{RS}^{\delta}_{\Delta t} = \{I_i\}_{i=1,\ldots,m} = \{[t^i_s, t^i_e]\}_{i=1,\ldots,m} \tag{4.1}$$

where t^i_s and t^i_e (with $t^i_s \leq t^i_e$) denote the *starting* and *ending* timestamps of the i*–th* avoidance, $I_i = [t^i_s, t^i_e]$ denotes a temporal interval during which an avoidance occurs, while the set $\{I_i\}_{i=1,\ldots,m}$ consists of *disjoint intervals*, i.e., $\forall i, j,\ I_i \cap I_j = \emptyset$ with $i \neq j$.

The *fused detector* method is motivated by the fact that, as we observed before, different kinds of avoidance behaviors may require the usage of different thresholds δ in order to be detected. This leads to the idea of considering several runs of the algorithm with different parameters and then suitably merging the results. We say that this fusion operation yields a *fused detector*.

Given a fixed look-ahead time Δt and a sequence of meet thresholds $\langle \delta_1, \ldots, \delta_h \rangle$, where $\delta_1 < \ldots \delta_i < \ldots < \delta_h$, we can *fuse* the result sets obtained for each δ_i, still obtaining a disjoint set of temporal intervals. Specifically,

$$\mathcal{RS}^{\langle \delta_1,\ldots,\delta_h \rangle}_{\Delta t} = \biguplus_{j=1,\ldots,h} \mathcal{RS}^{\delta_j}_{\Delta t} \tag{4.2}$$

where the result set $\mathcal{RS}^{\langle \delta_1,\ldots,\delta_h \rangle}_{\Delta t}$ is obtained by *fusing* the interval sets obtained by all the simple detectors with parameters $\delta \in \{\delta_1, \ldots, \delta_h\}$. The operation $\biguplus$ represents a simple set-union of the various intervals, except for the fact that it merges groups of *overlapping intervals*, namely groups of intervals where for each interval I, there is at least another interval I' such that $I \cap I' \neq \emptyset$. Each group of overlapping intervals is replaced by a single larger interval spanning the overlapping intervals. We note that this guarantees that the final fused result set is still composed of disjoint intervals, each one representing a distinct avoidance.

4.6 Experimental Evaluation

The goal of this section is to evaluate the proposed algorithm under different points of view. First, we quantitatively test the *effectiveness* of the algorithm by analyzing

the ability to correctly detect expected avoidance behaviors (Section 4.6.2). To this end we use an ad-hoc annotated dataset, our ground truth, created for the purposes of this work (Section 4.6.1). Second, we assess the ability of the algorithm in highlighting interesting and previously unknown patterns emerging from avoidance behaviors when using datasets for which no prior knowledge related to avoidance behaviors is available (Section 4.6.3). We left out experimental studies tackling performance or scalability issues: first, the aim of this work is to provide an algorithm (Algorithm 11) able to correctly detect avoidance behaviours between moving objects according to the formal framework introduced in Section 4.4; second, we recognize that a preprocessing step, executed before the proposed algorithm, would eliminate lots of useless computations (e.g., pairs of spatially faraway trajectories may be safely ignored). As a consequence, we deem that considering performance and scalability issues would be interesting in presence of some preprocessing step.

4.6.1 Experimental Setup

In order to quantitatively evaluate the effectiveness of the avoidance detectors (simple and fused), we need to study how the quality of result sets produced according to Equations (4.1) and (4.2) change when we adopt different meet thresholds δ and look-ahead times Δt.

To this end, the results obtained by detectors are compared with a *ground truth* of annotated trajectories (Section 4.6.1.1). Moreover, we need to define numerical metrics of quality (Section 4.6.1.2).

4.6.1.1 Ground truth of annotated trajectories

We exploit a *ground truth* of annotated trajectories, explicitly created for this work, whose data derive from real GPS observations of moving objects collected in Florianopolis and Venice. The dataset contains an overall amount of 86 trajectories representing pedestrian movements, for a total of 7,834 samples and an average sampling rate of one second.

Trajectories are logically grouped in pairs (for a total of 43 pairs), where each pair represents a single test case. According to entities behaviors, an(multiple) avoidance(s) may occur or not. When an avoidance occurs, the average distance where the entities start to change their behavior (individually or mutually) is approximately equal to 4 meters, even though few avoidances happen at greater distance.

A total of 24 trajectories (12 pairs) were collected in Florianopolis: among these, 10 pairs are labeled as *positive*, since they exhibit at least one avoidance. Among the positive pairs, one pair exhibits two distinct avoidances while the remaining ones a single avoidance.

As regards data gathered in Venice, a total of 62 trajectories (31 pairs) were collected: among these, 22 pairs are labeled as *positive*. Among the positive pairs, 7 exhibit two avoidances while the remaining ones a single avoidance.

For each positive pair, the set of intervals during which the avoidance(s) occur(s) is reported as well. The positive/negative labels and the temporal intervals referring to single avoidances were given by human assessors. This may result in imprecise annotation of temporal intervals, since their span may depend on the perception of assessors.

We also keep separate data belonging to Florianopolis or Venice, since the algorithm can trivially filter out (spatially or temporally) pairs of trajectories coming from two different sets.

4.6.1.2 Quality Metrics

In order to evaluate the effectiveness of the (simple/fused) detector that solves the *decision problem* (Definition 19), for each pair of trajectories we check the correctness of the detector by verifying if the yes/no answer matches the positive/negative label in the ground truth. The detector returns *yes* if a non-empty set of avoidances is detected, *no* otherwise.

On the other hand, in order to evaluate the quality of the detector that solves the *search problem* (Definitions 21 and 23), for each positive pair correctly detected we also inspect the temporal intervals returned by the detector, by comparing them with the ones associated by the human assessor in the ground truth.

Decision problem. For this study we recur to well-established metrics commonly used in data mining to evaluate the quality of a classifier [74]. Given a set of N pairs of trajectories, we evaluate the results of the detector algorithm by constructing an integer *confusion matrix* (see Table 4.1), a 2×2 table where the number of *true positives* (tp) and *true negatives* (tn) are given in the main diagonal, while the anti-diagonal contains the number of *false positives* (fp) and *false negatives* (fn) detected. Clearly, $N = tp + tn + fp + fn$.

We call *positive* any trajectory pair in the ground truth that is labeled as *avoidance behavior = yes*, while we use the term *negative* otherwise. Hence, tp and tn correspond to the pairs which the detector labels correctly *yes* or *no*, respectively, while fp and fn correspond to mislabeled pairs. Specifically, fp (fn) are pairs that the detector labels as positive (negative), but in fact appear as negative (positive) in the ground truth.

		Detected	
		positive	*negative*
Actual	*positive*	tp	fn
	negative	fp	tn

Table 4.1: Confusion matrix.

Recall and *Precision* are two widely used metrics employed in applications where successful detection of positive cases, i.e., in our case trajectory pairs for which an avoidance behavior is observed, is considered more significant than detection of other behavior. A formal definition of these metrics is given below:

$$\text{Precision}, \ \ p = \frac{tp}{tp + fp} \qquad\qquad \text{Recall}, \ \ r = \frac{tp}{tp + fn}$$

Precision determines the fraction of pairs that actually turn out to be positive in the group the detector algorithm has declared as positive, i.e., pairs that the algorithm detects as exhibiting an avoidance behavior. The higher is the precision, the lower is the number of false positive errors made by the detector algorithm. *Recall* measures the fraction of positive examples correctly detected by the algorithm. Classifiers with large recall have very few positive pairs in the ground truth mis-detected as negative.

Note that the two measures do not directly test the capability of the algorithm in detecting correctly the negative pairs, namely the number tn. However, if tn turns out to be less than the maximum value allowed, corresponding to the pairs in the ground truth that actually do not exhibit any avoidance behavior, as a consequence we should observe the increasing of fp, and thus a reduction of the precision.

Precision p and recall r can be summarized into another metric known as *F-Measure*, defined as follows:

$$\text{F-Measure}, \ \ F = \frac{2rp}{r + p}$$

where $0 \leq F \leq 1$. F represents a harmonic mean between recall and precision. Since the harmonic mean of two numbers tends to be closer to the smaller of the two numbers, we obtain a value of F close to 1 only if a detector performs well in terms of both precision and recall.

Search problem. As previously stated, for positive pairs correctly detected by the simple/fused detector we also inspect the temporal intervals returned by comparing them with the ones associated by human assessor in the ground truth.

Let TP be the set of positive pairs p_i in the ground truth that were correctly identified by the detector, i.e., pairs p_i for which the result set returned by the algorithm is not empty. Assuming that we are using a given combination Δt and δ, for clarity purposes in this context we denote such result set as $\mathcal{RS}^i$, omitting the parameters symbols in the notation. For each pair $p_i \in TP$, we know the set of actual disjoint temporal intervals $\mathcal{G}^i = \{\overline{I}^i_1, \ldots, \overline{I}^i_k\}$ in the ground truth, associated with $k > 0$ avoidances. The detector, either single or fused, also returns a set of m intervals $\mathcal{RS}^i = \{I^i_1, \ldots, I^i_m\}$.

Let $\widehat{\mathcal{G}}^i = \{\overline{I}^i_j \in \mathcal{G}^i \mid \exists! I^i_h \in \mathcal{RS}^i \ s.t. \ (\overline{I}^i_j \cap I^i_h \neq \emptyset) \land (\nexists \overline{I}^i_k \in \mathcal{G}^i, k \neq j \ s.t. \ \overline{I}^i_k \cap I^i_h \neq \emptyset)\}$, $\widehat{\mathcal{G}}^i \subseteq \mathcal{G}^i$, be the set of intervals in $\mathcal{G}^i$ such that each interval overlaps with *only one* interval in $\mathcal{RS}^i$, and the latter does not overlap with any other intervals in $\mathcal{G}^i$.

Finally, we can quantitatively evaluate the quality of the results for all the pairs $p_i \in TP$ by *Q-Measure*:

$$Q\text{-}Measure = \frac{\sum_{p_i \in tp} \frac{|\widehat{\mathcal{G}}^i|}{|\mathcal{G}^i|}}{|tp|}$$

where $0 \leq Q\text{-}Measure \leq 1$. Ideally *Q-Measure* should be equal or close to 1.

4.6.2 Analysis of the Ground Truth Dataset

In this section we evaluate the results of the algorithm when applied to the ground truth of annotated trajectories. First, in Section 4.6.2.1 we present some visual examples of the output of the algorithm. Then, in Sections 4.6.2.2 and 4.6.2.3 we show quantitatively the results obtained by exploiting the (simple/fused) detector that solves the decision or the search problem.

In all the experiments described below, we discuss the various results obtained by changing the two main parameters of our avoidance detection algorithm, namely Δt and δ, whose values are reported in seconds and meters, respectively.

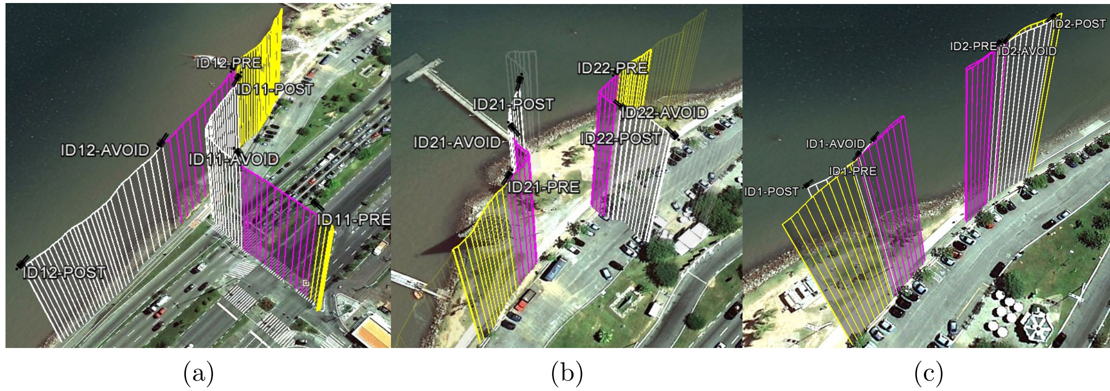
4.6.2.1 Visual inspection of avoidances

In order to visualize some avoidances detected by our algorithm on the ground truth dataset, we conveniently use Google Earth. Specifically, the avoidances shown in Figure 4.8 refer to trajectory pairs collected in Florianopolis. The subset of segments highlighted in *purple* represents the set of samples over which an avoidance is detected. The segments highlighted in *yellow* and *white* represent, respectively, a fixed sequence of samples occurring *before* and *after* the avoidance detected by the algorithm. Figure 4.8(a) represents an individual avoidance by entity ID11 (which moves initially from bottom-right to top-left), slowing down at some point in order to avoid ID12 (moving from top-right to bottom-left). Figure 4.8(b) depicts a mutual avoidance where ID21 (moving from bottom-left to top-right) and ID22 (moving in the opposite direction) change their direction as soon as they get close. Finally, Figure 4.8(c) depicts a mutual avoidance where the two entities invert their direction as soon as they get too close.

4.6.2.2 Results for the Decision Problem

In this section we evaluate the ability of the detectors (simple detector and fused detector) of correctly identifying the trajectory pairs of the ground truth that are positively/negatively labeled (decision problem).

Simple detector. For each pair of parameters Δt and δ, we build the confusion matrix by considering all trajectory pairs in the ground truth, then determine precision/recall, and finally compute the F-Measure scores. Figure 4.9 reports the scores obtained by the algorithm for all combinations of parameters. Specifically,



(a) (b) (c)

Figure 4.8: Examples of three visual inspections performed on three different avoidances returned by the algorithm.

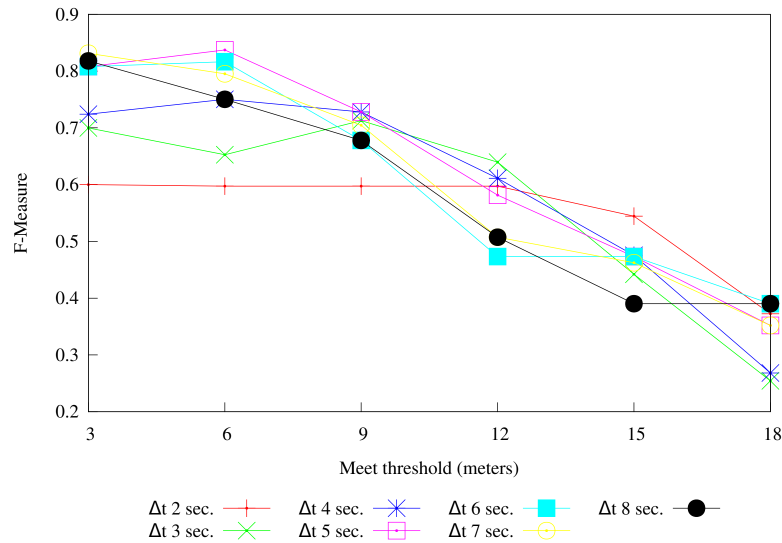


Figure 4.9: Decision problem with simple detector: F-Measure analysis.

each curve in the plot refers to a distinct Δt, and shows the F-Measure score as a function of δ.

For almost all values of Δt, a common optimal value for the meet threshold δ that maximizes the F-measure score falls in the interval $[3, 6]$; for larger values of δ, the score degrades. It is worth noting that these optimal values for δ approximately reflect the average avoidance distance used to physically produce the avoidances for (positive) trajectory pairs included in the ground truth.

If we compare the various curves within this optimal interval of δ values, we observe that the *F-Measure* score gets larger whenever the look-ahead time Δt increases, except for $\Delta t = 7$ and 8, for which the *F-Measure* score starts decreasing. We already noticed that Δt can be interpreted as a measure of the future awareness

of moving objects. Consequently, Δt must be chosen according to their characteristics. From our tests, we observe that Δt values larger than 7 are not suited for the features of the moving objects in our ground truth. The reason is that we forecast object behaviors by considering relatively old and thus scarcely relevant movement data. This induces the detection of some false positive avoidances, which in turn entails loss of precision with negative effects on the F-Measure.

Fused detector. In the following we show how the overall performance of the algorithm can substantially be improved by conveniently *fusing* different result sets of distinct simple detectors, according to the fusion operator defined by Equation (4.2).

Given a Δt, and a sequence of values for δ, e.g., $\langle 3, 6, 9 \rangle$, a pair of trajectories is identified as a positive case, i.e., *avoidance behavior* = *yes*, if at least a simple detector for some δ in the sequence identifies one or more avoidance behaviors. Conversely, if no simple detector is able to recognize any avoidance, the pair is identified as a negative case, i.e., *avoidance behavior* = *no*. Still, for the fused detector we can build the confusion matrix, by considering all trajectory pairs in the ground truth, and finally compute the F-Measure scores.

Figure 4.9 shows how the result sets related to $\Delta t \in \{4, 5, 6\}$ yield the best F-Measures. Thus, we focus on these and, for each Δt, we compute the F-Measure related to the fused result sets $\mathcal{RS}_{\Delta t}^{\langle \delta=3 \rangle}, \mathcal{RS}_{\Delta t}^{\langle \delta=3, \delta=6 \rangle}, ..., \mathcal{RS}_{\Delta t}^{\langle \delta=3, \delta=6, ..., \delta=18 \rangle}$ (each one defined as per Equation 4.2). Results are reported in Figure 4.10, where each fused result set is represented by its upper δ threshold in the X-axis. The figure shows how the fused detector entails substantial improvements in terms of F-Measure (up to 95%), provided that Δt is properly chosen according to the dataset features.

In general, we argue that the opportunity of fusing different result sets depends on the kind of analysis we want to perform. Specifically, it depends on the classes of avoidance behaviors we want to discover (e.g., only values for δ that are relevant for our purposes should be used for the fusion operation), and on the amount of useful information an analyst is interested in extracting at the expense of possible losses in precision (due to the detection of false avoidances).

4.6.2.3 Results for the Search Problem

In this section we assess the quality of the temporal intervals reported by both detectors (simple and fused), for the positive pairs correctly detected, and thus included in the set of trajectory pairs TP, where $tp = |TP|$.

Simple detector. Figure 4.11 reports the Q-Measure scores obtained in the experiments. The experimental findings confirm all the remarks done in Section 4.6.2.2 concerning Δt and δ. In general, we obtain the best Q-Measure score for the same parameters Δt and δ for which we obtained the best F-Measure scores. We also point out that using high values of Δt (where *high* is relative to the dataset features) may erroneously induce the fusion of distinct avoidance behaviors, due to compression,

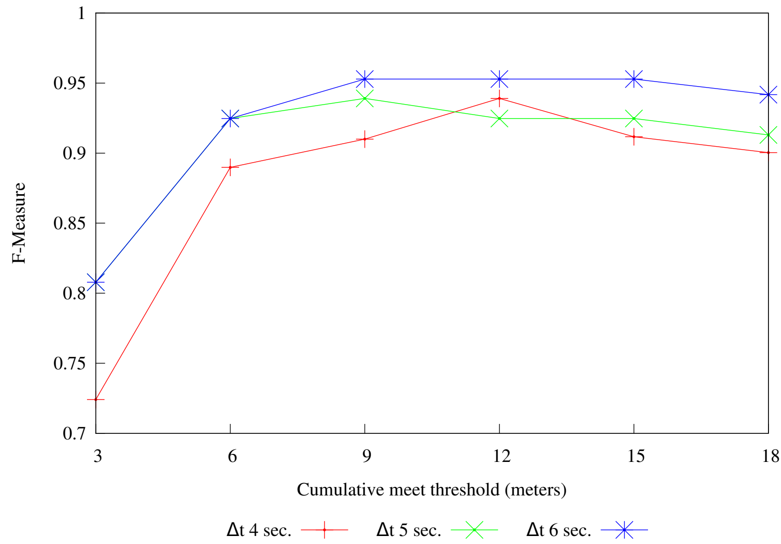


Figure 4.10: Decision problem with fused detector: F-Measure analysis. In X-axis the values are the maximums of the thresholds δ used during the fusion operation, e.g., 9 is the maximum for the set $\{3, 6, 9\}$.

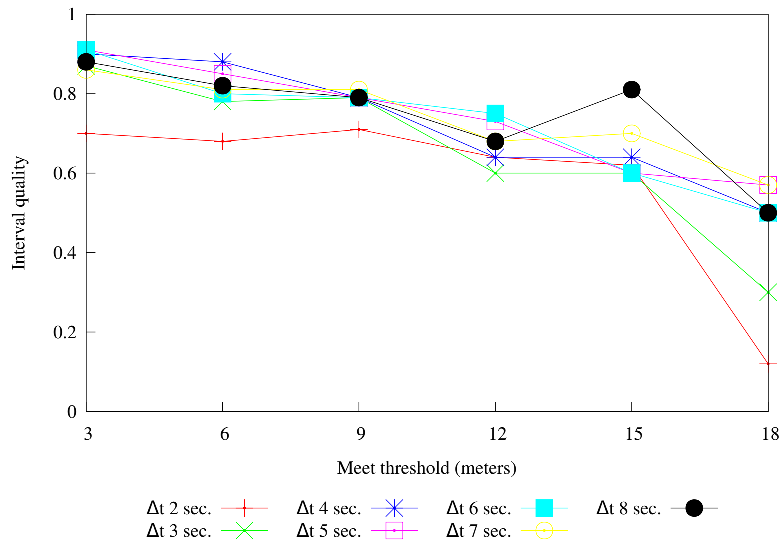


Figure 4.11: Search problem with simple detector: Q-Measure analysis.

thus potentially mapping multiple avoidances occurring between two trajectories in the ground truth to a single detected avoidance. This in turn induces losses in terms of Q-Measure scores.

Fused detector. Also for the fused detector we aim at analyzing the quality of the temporal intervals for the trajectory pairs in TP. To this end, we consider again the fused result sets belonging to $\{\mathcal{RS}_{\Delta t}^{\langle\delta=3\rangle}, \mathcal{RS}_{\Delta t}^{\langle\delta=3,\delta=6\rangle}, ..., \mathcal{RS}_{\Delta t}^{\langle\delta=3,\delta=6,...,\delta=18\rangle}\}$.

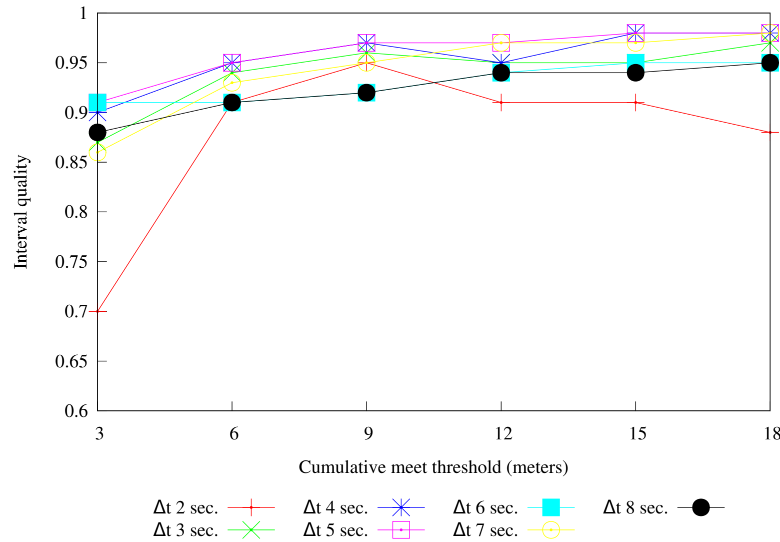


Figure 4.12: Search problem with fused detector: Q-Measure analysis. In X-axis the values are the maximums of the thresholds δ used during the fusion operation, e.g., 9 is the maximum for the set $\{3, 6, 9\}$.

Figure 4.12 reports the performance of the algorithm in terms of Q-Measure score. We can observe how fusing different result sets entails substantial improvements. These improvements are particularly evident whenever the fusion is performed for δ values close to the average distances used for physically producing avoidances ($\delta \in [3, 6]$).

4.6.3 Analysis of a Real World Unannotated Dataset

In this section we consider the AIS Brest dataset, a real world unannotated dataset containing 824 trajectories related to the movements of 824 ships[1] nearby Brest's harbor [75]. Basic statistics reveal that the dataset contains 5.756.438 points, the trajectories move at an average speed of 7.77 km/h and most of the trajectories have an average sampling rate between 1 and 20 seconds. These characteristics make the dataset quite interesting in terms of the precision with which the trajectories are described.

By considering the aforementioned statistics, and after a scrutinization of different meet thresholds and look-ahead times, according to entities' features we chose a meet threshold δ of 30 meters and a look-ahead time Δt of 50 seconds. Indeed, we argue that for this case study this combination of parameters allows us to capture interesting patterns, as we will show further on.

Before performing any avoidance detection we preprocess the dataset to remove points inducing trajectory segments having a speed above 50 km/h (these are nec-

[1]Each trajectory is uniquely associated with a ship.

essarily noisy data).

The algorithm detects a total of 1480 avoidances, among which 321 are mutual, 970 individual and 189 weak. Given this considerable amount of information it is necessary to perform a deeper analysis in order to infer meaningful patterns.

Among the 824 ships, 229 are involved in at least one avoidance. We call this set as the *set of active ships*. If we further look at the number of avoidances in which each active ship is involved, we notice that 8 ships are involved in more than 100 avoidances, while the vast majority - more precisely 196 ships (which constitutes the 85,5% of the active ships set) - are involved in a number of avoidances between 1 and 10. We call the former set as the *set of frequent ships* while the rest of the ships ends up in the set of *infrequent ships*.

The information above suggests that the frequent ships play a very important role in the dataset. If we decompose the total amount of avoidances detected by the algorithm, we find out that the overall amount of avoidances between frequent and infrequent ships are 973 (65,7% of the result set), while the avoidances between frequent ships are 386 (26,1%) and between infrequent ones are 121 (8,2%).

If we look at the MMSI codes of the frequent ships in order to find out their typology (Table 4.2), we have that the top-2 frequent ships are *pilot ships*, while the remaining ones are *passenger ships* and *tugboats*.

MMSI Code	*Type*	Amount of avoidances
227730220	Pilot ship	414
227005550	Pilot ship	364
227635210	Passenger ship	194
227592820	Passenger ship	175
227574020	Passenger ship	174
227612860	Passenger ship	158
227574030	Passenger ship	147
228051000	Tugboat	119

Table 4.2: *Frequent ships* details.

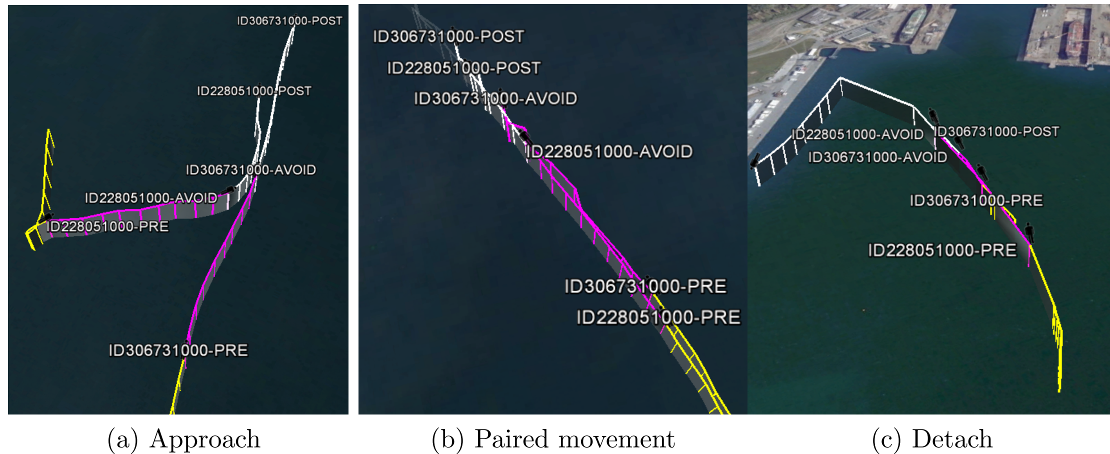
Given these data we want to answer the following questions:

1. Which are the events producing so many avoidances between frequent and infrequent ships?

2. Which are the events producing a considerable amount of avoidances between frequent ships?

3. Is there any kind of recurring pattern causing avoidances between infrequent ships?

When answering Question (1) we notice a dominant pattern (Figure 4.13) that we call *paired movement event*. Through a graphical inspection we observe that



(a) Approach (b) Paired movement (c) Detach

Figure 4.13: Example of a paired movement event involving the frequent ship 228051000. The ships are moving from bottom to top.

almost all these events can be decomposed in 3 phases: during the first phase the two ships approach each other, mostly when they are entering or exiting the harbor (*approach* phase, Figure 4.13a). Then, the ships proceed paired (*paired movement* phase, Figure 4.13b) until they approach the docks or they exit the harbor area. During this intermediate phase some avoidances may emerge or not, depending on the continuous adjustment performed by both ships in order to maintain the relative distance. Finally, the ships separate (*detach* phase), as shown in Figure 4.13c.

Given the typologies of the frequent ships reported in Table 4.2 we argue that the pilot ships and the tugboats produce these events when they have to pilot (or tow, respectively) an incoming (or outgoing) ship. For what concerns the passenger ships, we argue that they adjust their trajectories in order to avoid other infrequent ships nearby; moreover, the amount of avoidances in which they are involved is justified by the fact that they are servicing, and thus repeatedly going through, a fixed route.

In general we expect that these avoidances are mostly distributed in predetermined areas. Indeed, if we plot the avoidances occurring during a two-month window we see that they are approximately distributed on a fixed path going from the harbor's docks to the strait exit (Figure 4.14). This also gives an idea about the most dangerous or trafficked areas (especially near the docks).

Concerning Question (2), we found that many avoidances are produced according to the same pattern observed for Question (1), or when a frequent ship is docking (and therefore slowing down, hence the avoidance) in the harbor nearby already docked ships. The latter pattern is observed between frequent and infrequent ships as well, although with a lesser extent.

Finally, as far as Question (3) is concerned we found out that the second pattern observed when explaining the avoidances related to Question (2) also occurs, i.e.,

Figure 4.14: Subtrajectories related to avoidance behaviors detected in a time interval spanning two months ([20/04/2009, 20/06/2009]).

almost all the avoidances between infrequent ships happen near the docks when one or more ships are docked while one ship is docking nearby.

Conclusions

In this dissertation we contributed to the vast field of mobility data research by addressing two distinct and specific problems. The first contribution, which belongs to the family of *on-line* mobility data analyses, tackles the problem of computing massive amounts of repeated range or k-NN queries over massive moving objects observations (Chapters 1, 2 and 3). In this contribution we present novel methods, relying on scalable grid-based spatial indices and on ad-hoc data structures, capable of addressing efficiently such problem. Since queries are repeatedly issued by the very same moving objects, also queries continuously change their issuing location over time. Our solutions are the first known to exploit GPUs in order to speed-up range and k-NN query processing and, at the same time, effectively contend with skewed spatial distributions of objects and queries. To achieve these goals we introduce two hybrid CPU/GPU range and k-NN query processing pipelines - exploiting query batching, and leverage bitmap-based intermediate data structures (in the range queries case), as well as a PR-quadtree based spatial index, in order to effectively exploit the GPUs architectural features. We extensively test our solutions to study their sensitivity to parameters and data distribution. In such experiments we prove several arguments, above all that our solutions (i) outperform baseline GPU approaches and (ii) achieve significant performance gains with respect to the best known CPU sequential competitors. We also show that (iii) our proposals are able to outperform advanced GPU-based uniform grid-based solutions, since these are unable to fully capture the data skewness - an ability which is needed in order to yield much more uniform workload distributions on GPUs.

As a future direction of research it may be interesting to increase the overall memory throughput yielded during the execution of the various pipelines phases, since this represents the main performance bottleneck. This calls for the introduction of novel, ad-hoc GPU-friendly data-structures coupled with proper memory access patterns, as well as for possible novel, yet GPU-friendly, spatial indices. Partly bounded to this challenge, another interesting direction of research would be to extend our work to spaces having slightly higher dimensionality, which in turn would require to devise proper spatial indices (and data structures) which can be quickly constructed and accessed on GPU.

The second contribution (Chapter 4) belongs to the family of *off-line* analyses used to conduct mobility data *mining*, and positions itself in the ample branch of research dealing with the extraction of mobility patterns from trajectories. Several algorithms have been proposed for mining different types of trajectory patterns. However, an interesting behaviour that has not been much explored in historical trajectories of moving objects is *avoidance*. Our contribution introduces a new

type of trajectory pattern: avoidance between moving objects. We present a set of theoretical definitions and an algorithm which is able to detect such patterns. The discovery of avoidances between moving objects is challenging, since the intent of any moving object may not be immediately apparent from its trajectories. To determine an avoidance, two objects should move towards the same area, but either one or both should change their behavior when they come close enough to be aware of each other. To identify a behaviour change we forecast the movements of both moving objects and compare them with the actual movements. If the forecasts predict a meet but the actual movements do not meet, an avoidance is detected. Each detected avoidance is in turn classified, whenever possible, as individual when only one moving object changes significantly its behaviour, while it is classified as mutual when both objects change their behaviour significantly. It is worth mentioning that a behaviour change is measured through the distance between the forecast movement and the actual movement. Such generalization prevents from using specific features such as direction and speed for detecting a change of behaviour. Besides analyzing the parameters of the algorithm, in our contribution we went one step further by introducing the idea of a fused detector, which merges the result sets of several simple detectors (with different meet thresholds) in order to allow the detection of a possibly broad range of avoidance behaviours.

The algorithm has been evaluated with two real-world datasets. The first dataset is annotated and it contains pedestrian movements; the purpose of analyzing such dataset is to verify that the algorithm is able to detect avoidances which are expected to occur. Indeed, the experiments conducted on this dataset show how the algorithm is able to detect avoidances expected to occur while guaranteeing the quality of the results returned. The second real-world dataset, unannotated, contains ship movements nearby the Brest's harbour. Since no prior avoidance information is available, the purpose of analyzing such dataset is to check whether the algorithm is able to extract interesting evidence from the data. Indeed, by characterizing the avoidances between ships on the basis of their frequency, their spatial distribution and by means of visual inspections on the behaviour of frequent ships, we were able to highlight the most trafficked areas, as well as a frequently recurring event, i.e., the paired movement event. We have not compared the results of our algorithm with other approaches because, to the best of our knowledge, there is no other work in trajectory analysis literature tackling the detection of the pattern we propose.

Future work includes an analysis on the effect of using different forecast functions and the definition of a confidence measure to evaluate any avoidance. Another interesting direction of research is related to the definition of an efficient and fast preprocessing algorithm able to prune out pairs of trajectories which cannot exhibit avoidances (i.e., do not intersect spatially) over a given temporal interval, thus reducing the overall execution time of the algorithm proposed in this work. Finally we note that our pattern, like others in the related research area, may be used as a tool to devise more advanced and complex analyses, especially when it comes to semantic trajectories.

Bibliography

[1] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[2] Nikos Pelekis and Yannis Theodoridis. *Mobility Data Management and Exploration*. Springer, 2014.

[3] Chiara Renso, Stefano Spaccapietra, and Esteban Zimanyi, editors. *Mobility Data: Modeling, Management, and Understanding*. Cambridge University Press, Cambridge, UK, 2013.

[4] Christine Parent, Stefano Spaccapietra, Chiara Renso, Gennady Andrienko, Natalia Andrienko, Vania Bogorny, Maria Luisa Damiani, Aris Gkoulalas-Divanis, Jose Macedo, Nikos Pelekis, Yannis Theodoridis, and Zhixian Yan. Semantic trajectories modeling and analysis. *ACM Computing Surveys*, 40, 2013.

[5] Claudio Silvestri, Francesco Lettich, Salvatore Orlando, and Christian S Jensen. Gpu-based computing of repeated range queries over moving objects. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pages 640–647. IEEE, 2014.

[6] Francesco Lettich, Salvatore Orlando, Claudio Silvestri, and Christian S. Jensen. Manycore processing of repeated range queries over massive moving objects observations. Under review.

[7] Francesco Lettich, Luis Otavio Alvares, Vania Bogorny, Salvatore Orlando, Claudio Silvestri, and Alessandra Raffaetà. Detecting avoidance behaviors between moving objects. Under review.

[8] Dittrich J., Blunschi L., and Vaz Salles M.A. MOVIES: indexing moving objects by shooting index images. *Geoinformatica*, 15(4):727–767, 2011.

[9] Gray J. and Reuter A. *Transaction processing: concepts and techniques*. Morgan Kaufmann Publishers, 1993.

[10] Šidlauskas D., Šaltenis S., and Jensen C.S. Parallel main-memory indexing for moving-object query and update workloads. In *Proc. of ACM SIGMOD Conf.*, pages 37–48, 2012.

[11] Kornacker M., Mohan C., and Hellerstein J.M. Concurrency and recovery in generalized search trees. In *Proc. of ACM SIGMOD Conf.*, pages 62–72, 1997.

[12] Hong S. and Kim H. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. *ACM SIGARCH Computer Architecture News*, 37(3):152–163, 2009.

[13] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 451–460. ACM, 2010.

[14] Hennessy J.L. and Patterson D.A. *Computer Architecture - A Quantitative Approach (5. ed.).* Morgan Kaufmann, 2012.

[15] Sun C., Agrawal D., and El Abbadi A. Hardware acceleration for spatial selections and joins. In *Proc. of ACM SIGMOD Conf.*, pages 455–466, 2003.

[16] Benjamin Sowell, Marcos Vaz Salles, Tuan Cao, Alan Demers, and Johannes Gehrke. An experimental analysis of iterated spatial joins in main memory. *Proc. VLDB Endow.*, 6(14):1882–1893, September 2013.

[17] Šidlauskas D., Ross K.A., Jensen C.S., and Šaltenis S. Thread-level parallel indexing of update intensive moving-object workloads. In *Proc. of SSTD Conf.*, pages 186–204, 2011.

[18] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. Fast bvh construction on gpus. In *Computer Graphics Forum*, volume 28, pages 375–384. Wiley Online Library, 2009.

[19] Luo L., Wong M.D.F., and Leong L. Parallel implementation of r-trees on the gpu. In *IEEE Conf. on Design Automation*, pages 353–358, 2012.

[20] Yu B., Kim H., Choi W., and Kwon D. Parallel range query processing on R-tree with graphics processing unit. In *IEEE Conf. on Dependable, Autonomic and Secure Computing*, pages 1235–1242, 2011.

[21] Tero Karras. Maximizing parallelism in the construction of bvhs, octrees, and k-d trees. In *High Performance Graphics*, pages 33–37, 2012.

[22] Sariel Har-Peled. *Geometric approximation algorithms.* Number 173. American Mathematical Soc., 2011.

[23] Rajeev Raman and David S Wise. Converting to and from dilated integers. *Computers, IEEE Transactions on*, 57(4):567–573, 2008.

[24] Zhang J. and You S. Speeding up large-scale point-in-polygon test based spatial join on GPUs. In *Proc. of ACM SIGSPATIAL Intl. Wksp. on Analytics for Big Geospatial Data*, pages 23–32, 2012.

[25] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[26] Jerome H Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):209–226, 1977.

[27] Dinesh P. Mehta and Sartaj Sahni. *Handbook Of Data Structures And Applications (Chapman & Hall/Crc Computer and Information Science Series.)*. Chapman & Hall/CRC, 2004.

[28] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. Searching in metric spaces. *ACM Comput. Surv.*, 33(3):273–321, September 2001.

[29] Gísli R. Hjaltason and Hanan Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, June 1999.

[30] Yoav Freund, Sanjoy Dasgupta, Mayank Kabra, and Nakul Verma. Learning the structure of manifolds using random projections. In *Advances in Neural Information Processing Systems*, pages 473–480, 2007.

[31] Anand Rajaraman and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge University Press, 2011.

[32] Ella Bingham and Heikki Mannila. Random projection in dimensionality reduction: Applications to image and text data. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '01, pages 245–250, New York, NY, USA, 2001. ACM.

[33] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 604–613, New York, NY, USA, 1998. ACM.

[34] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, January 2008.

[35] Vincent Garcia, Eric Debreuve, and Michel Barlaud. Fast k nearest neighbor search using gpu. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*, pages 1–6. IEEE, 2008.

[36] Justin Heinermann, Oliver Kramer, Kai Lars Polsterer, and Fabian Gieseke. On gpu-based nearest neighbor queries for large-scale photometric catalogs in astronomy. In *KI 2013: Advances in Artificial Intelligence*, pages 86–97. Springer, 2013.

[37] Naohito Nakasato. Implementation of a parallel tree method on a gpu. *Journal of Computational Science*, 3(3):132–141, 2012.

[38] Deyuan Qiu, Stefan May, and Andreas Nüchter. Gpu-accelerated nearest neighbor search for 3d registration. In *Computer Vision Systems*, pages 194–203. Springer, 2009.

[39] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless kd-tree traversal for high performance gpu ray tracing. In *Computer Graphics Forum*, volume 26, pages 415–424. Wiley Online Library, 2007.

[40] Fabian Gieseke, Justin Heinermann, Cosmin Oancea, and Christian Igel. Buffer kd trees: processing massive nearest neighbor queries on gpus. In *Proceedings of The 31st International Conference on Machine Learning*, pages 172–180, 2014.

[41] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient processing of spatial joins using r-trees. *SIGMOD Rec.*, 22(2):237–246, June 1993.

[42] Sengupta S., Harris M., Zhang Y., and Owens J.D. Scan primitives for GPU computing. In *Proc. of ACM SIGGRAPH Symposium on Graphics Hardware*, pages 97–106, 2007.

[43] Merrill D. and Grimshaw A.S. High performance and scalable radix sorting: a case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters*, 21(2):245–272, 2011.

[44] Matt Pharr and Randima Fernando. *Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation.* Addison-Wesley Professional, 2005.

[45] Daniel Cederman and Philippas Tsigas. On dynamic load balancing on graphics processors. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 57–64. Eurographics Association, 2008.

[46] Sowell B., Vaz Salles M., Cao T., Demers A., and Gehrke J. Indexing framework. http://www.cs.cornell.edu/~sowell/indexing/.

[47] Thomas Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6(2):153–180, 2002.

[48] Tolu Alabi, Jeffrey D Blanchard, Bradley Gordon, and Russel Steinbach. Fast k-selection algorithms for graphics processing units. *Journal of Experimental Algorithmics (JEA)*, 17:4–2, 2012.

[49] Jia Pan and Dinesh Manocha. Fast gpu-based locality sensitive hashing for k-nearest neighbor computation. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '11, pages 211–220, New York, NY, USA, 2011. ACM.

[50] Fosca Giannotti, Mirco Nanni, Fabio Pinelli, and Dino Pedreschi. Trajectory pattern mining. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 330–339. ACM, 2007.

[51] Patrick Laube, Stephan Imfeld, and Robert Weibel. Discovering relative motion patterns in groups of moving point objects. *International Journal of Geographical Information Science*, 19(6):639–668, 2005.

[52] Monica Wachowicz, Rebecca Ong, Chiara Renso, and Mirco Nanni. Finding moving flock patterns among pedestrians through collective coherence. *International Journal of Geographical Information Science*, 25(11):1849–1864, 2011.

[53] Joachim Gudmundsson and Marc J. van Kreveld. Computing longest duration flocks in trajectory data. In Rolf A. de By and Silvia Nittel, editors, *14th ACM International Symposium on Geographic Information Systems, ACM-GIS 2006, November 10-11, 2006, Arlington, Virginia, USA, Proceedings*, pages 35–42. ACM, 2006.

[54] Zhenhui Li, Bolin Ding, Jiawei Han, Roland Kays, and Peter Nye. Mining periodic behaviors for moving objects. In Bharat Rao, Balaji Krishnapuram, Andrew Tomkins, and Qiang Yang, editors, *KDD*, pages 1099–1108. ACM, 2010.

[55] Roberto Trasarti, Fabio Pinelli, Mirco Nanni, and Fosca Giannotti. Mining mobility user profiles for car pooling. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1190–1198. ACM, 2011.

[56] Linsey Xiaolin Pang, Sanjay Chawla, Wei Liu, and Yu Zheng. On detection of emerging anomalous traffic patterns using GPS data. *Data Knowl. Eng.*, 87:357–373, 2013.

[57] Fernando de Lucca Siqueira and Vania Bogorny. Discovering chasing behavior in moving object trajectories. *Transactions in GIS*, 15(5):667–688, 2011.

[58] Somayeh Dodge, Robert Weibel, and Anna-Katharina Lautenschütz. Towards a taxonomy of movement patterns. *Information Visualization*, 7(3-4):240–252, 2008.

[59] Luis Otávio Alvares, Alisson Moscato Loy, Chiara Renso, and Vania Bogorny. An algorithm to identify avoidance behavior in moving object trajectories. *J. Braz. Comp. Soc.*, 17(3):193–203, 2011.

[60] Zhenhui Li, Bolin Ding, Fei Wu, Tobias Kin Hou Lei, Roland Kays, and Margaret Crofoot. Attraction and avoidance detection from movements. *PVLDB*, 7(3):157–168, 2013.

[61] Dae-Jin Kim, Kwang-Hyun Park, and Zeungnam Bien. Hierarchical longitudinal controller for rear-end collision avoidance. *Industrial Electronics, IEEE Transactions on*, 54(2):805–817, 2007.

[62] Ilaria Xausa, Robert Baier, Matthias Gerdts, Mark Gonter, and Christian Wegwerth. Avoidance trajectories for driver assistance systems via solvers for optimal control problems. In *International Symposium on Mathematical Theory of Networks and Systems*, pages 1–8. Springer, 2012.

[63] Michael R. Hafner, Drew Cunningham, Lorenzo Caminiti, and Domitilla Del Vecchio. Cooperative collision avoidance at intersections: Algorithms and experiments. *IEEE Transactions on Intelligent Transportation Systems*, 14(3):1162–1175, 2013.

[64] Sergiu Nedevschi, Silviu Bota, and Corneliu Tomiuc. Stereo-based pedestrian detection for collision-avoidance applications. *Transactions on Intelligent Transportation Systems*, 10(3):380–391, September 2009.

[65] Yu-Hong Liu and Chao-Jian Shi. A fuzzy-neural inference network for ship collision avoidance. In *Proceedings of 2005 International Conference on Machine Learning and Cybernetics*, pages 4754–4754. IEEE Computer Society, 2005.

[66] Jun Min Mou, Cees van der Tak, and Han Ligteringen. Study on collision avoidance in busy waterways by using ais data. *Ocean Engineering*, 37(5):483–490, 2010.

[67] Surya Shandy and John Valasek. Intelligent agent for aircraft collision avoidance. In *Proceedings of AIAA Guidance, Navigation, and Control Conference*, pages 1–11. American Institute of Aeronautics and Astronautics, 2001.

[68] Arthur Richards and Jonathan P How. Aircraft trajectory planning with collision avoidance using mixed integer linear programming. In *American Control Conference, 2002. Proceedings of the 2002*, volume 3, pages 1936–1941, 2002.

[69] M Pechoucek and D Sislak. Agent-based approach to free-flight planning, control, and simulation. *IEEE Intelligent Systems*, 24(1):14–17, Jan 2009.

[70] Oussama Khatib. Real-time obstacle avoidance for robot manipulator and mobile robots. *International Journal of Robotics Research*, 5(1):90–98, 1986.

[71] J Borenstein and Y Koren. The vector field histogram-fast obstacle avoidance for mobile robots. *IEEE Transactions on Robotics and Automation*, 7(3):278–288, 1991.

[72] S.M. Khansari-Zadeh and A. Billard. Realtime avoidance of fast moving objects: A dynamical system-based approach. Electronic proc. of the Workshop on Robot Motion Planning: Online, Reactive, and in Real-Time [IROS 2012], 2012.

[73] Dali Sun, Alexander Kleiner, and Bernhard Nebel. Behavior-based multi-robot collision avoidance. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1668–1673, 2014.

[74] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining, (First Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[75] Laurent Etienne, Thomas Devogele, and Alain Bouju. Spatio-temporal trajectory analysis of mobile objects following the same itinerary. *Advances in Geo-Spatial Information Science*, 10:47, 2012.